Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

IP address lookup using bit-shuffled trie

Derek Pao [*], Ziyan Lu, Yat Hang Poon

Electronic Engineering Department, City University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 8 December 2012
Received in revised form 14 January 2014
Accepted 9 April 2014
Available online 18 April 2014

Keywords:
IPv4 and IPv6 address lookup
Packet forwarding
Pipelined architecture

ABSTRACT

An algorithmic RAM-based IP address lookup method called *bit-shuffled trie* is presented. By rearranging the bits of the prefixes, memory efficient index tables can be constructed to support IP address lookup. The address lookup engine can be implemented using pipelined architecture with simple processing logic. The proposed method has superior memory efficiency. The memory cost for a 474 K prefixes IPv4 routing table is only 1.1 MB, and the memory cost for a 215 K 64-bit prefixes IPv6 routing table is about 1.7 MB. The exceptional memory efficiency of the proposed method allows us to implement the IP address lookup engine for both IPv4 and IPv6 on a single FPGA device. Incremental updates to the routing table can be handled efficiently. On average, about 8 memory-write operations to the data structures are required to process an insertion or deletion.

1. Introduction

The rapid growth of the Internet in terms of the number of users and the communication line speed have posed great challenges to the design of real-time packet forwarding hardware, in particular the IP address lookup engine. The routing table in today's core routers can have half a million IPv4 prefixes. To support 100 Gbps line speed, the IP address lookup engine needs to perform 300 million lookup per second (MLPS).

Ternary content addressable memory (TCAM) had been a popular candidate for implementing lookup tables [24,26] in high-speed switches/routers, e.g. Cisco Catalyst 6500 series. The TCAM device is used as a co-processor to relieve the workload of the network processor from performing IP header lookups. A typical TCAM cell contains 16 transistors, and it is much more expensive than SRAM. Power dissipation of TCAM is another concern. Hence, the industry and academia are eager to research on more cost-effective algorithmic RAM-based IP address lookup methods. For example, the IBM PowerEdge network processor is equipped with on-chip hardware accelerator to support wire-speed IP address lookup and packet classification [3,17].

The migration from IPv4 to IPv6 adds one more dimension to the problem, i.e. the address length is increased from 32 bits to 128 bits. The transition from IPv4 to IPv6 will likely be a very long process. The current IPv4 network will be in use and continues to grow, and the IPv4 and IPv6 networks can co-exist for many years [20]. Hence, the next generation IP address lookup method should be applicable to both IPv4 and IPv6.

In this paper a new IP address lookup method called *bit-shuffled trie* is presented. The proposed method has superior memory efficiency compared to existing methods [25]. The memory cost for an IPv4 routing table with 474 K prefixes is as low as 1.1 MB, and the memory cost for a 215 K IPv6 routing table with 64-bit prefixes is about 1.7 MB. The exceptional memory efficiency of the proposed method allows us to implement the IP address lookup engine for both IPv4 and IPv6 in a single FPGA device, and achieve a throughput of 350 MLPS. Power dissipation of our method is only 1.7 W and 4.2 W for IPv4 and IPv6 address lookup, respectively. The lookup engine is composed of several linear pipelines with simple processing logic. Incremental updates to the routing table can be supported. On average, about 8 memory writes are required to perform an insertion/deletion to the routing table.

1.1. General background and motivation

The IP lookup problem can be modeled as a searching problem over a binary-trie. The system traverses the trie along the path that corresponds to the packet's destination IP address. The last prefix seen along the given path is the *longest matching prefix* (LMP), and the packet will be forwarded using the next-hop field associated with the LMP found. One way to speed up the traversal of the trie is to use multibit-trie [8,30], where multiple bits of the input address are processed in one step. The number of bits processed in a step is called the *stride* width.

The method of [8] exemplifies the simplicity of RAM-based IP address lookup. The lookup engine is consisted of two levels of

* Corresponding author.
E-mail addresses: d.pao@cityu.edu.hk (D. Pao), ziyanlu2-c@my.cityu.edu.hk (Z. Lu), garypoon014@gmail.com (Y.H. Poon).

http://dx.doi.org/10.1016/j.comcom.2014.04.006
0140-3664/© 2014 Elsevier B.V. All rights reserved.

index tables in the basic configuration. The first level table (called *TBL*24) has 2^{24} entries, and the second level table (called *TBLlong*) contains multiple 256-entry blocks. Prefixes with up to 24 bits are expanded to 24 bits, and are stored in *TBL*24. Prefixes with 25 bits or more are expanded to 32 bits and are stored in *TBLlong*. This configuration is referred to as the 24–8 multibit-trie. To find the LMP for an input address the system will first take the most significant 24 bits, i.e. bits 1 to 24, of the input address to access *TBL*24. The information stored in *TBL*24 can be either the next-hop data (if the LMP for the input address is no longer than 24 bits), or a reference address for accessing *TBLlong* (if the LMP may be longer than 24 bits). For the latter case, the system will take the last 8 bits of the input address, i.e. bits 25 to 32, as the offset value. The system computes the effective address for accessing *TBLlong* by adding the 8-bit offset to the reference address obtained from *TBL*24. The IP address lookup operation can be completed in at most two memory accesses.

Simplicity is the major advantage of the above approach. The major disadvantage is the high memory cost. According to the evaluation of [8] using a relatively small routing table (less than 50 K prefixes), a 21-3-8 multibit-trie consumes 9 MB, and a 16-8-8 multibit-trie consumes 105 MB. A major reason for the high memory cost is that a significant proportion of the index tables are either empty or being used to store duplicated prefixes due to prefix expansion.

There have been attempts to reduce the memory cost of trie-based method using bit-map representation [7,23,32]. The overall binary-trie is divided into a number of subtries, and the locations of prefix nodes in a subtrie can be represented by a bit-vector. In addition to the bit-vector, the data structures also need to provide information for the associated next-hop data and the reference pointers for accessing the next level subtries. If the *stride* width is equal to *s*, the length of the bit-vector is equal to 2^{s+1}–1. The stride width is restricted to a small value, e.g. no more than 8, otherwise the logic circuits for processing the bit-vector will be too complex. One can also see that most of the storages are in fact consumed by the reference pointers. The memory cost of [7] is about 11 to 13 bytes per IPv4 prefix.

From the above discussion, we can make the following observations. First, the advantages of multibit-trie are the speed and simplicity. However, the memory efficiency is not satisfactory. Many of the index table entries are storing null values or duplicated prefixes due to the needs for prefix expansion. Second, the memory efficiency of the bit-map approach can be further improved if we can do away with the needs of the reference pointers in the data structure. This can be possible if the bit-map approach is used to represent subtries at the bottom level, i.e. there is no more next level subtrie.

Let's consider the case where we are searching for the LMP of an input address against a set of prefixes with a minimum length L_{min} and maximum length L_{max}. In the conventional multibit-trie approach, a prefix with $L_{max} - k$ bits will likely be expanded to 2^k entries, and this will lead to low memory efficiency. In our method, we try to minimize the memory cost of the IP address lookup engine by the following strategies.

(1) Prefixes in the routing table are divided into 6 groups based on the prefix length, such that $L_{max} - L_{min}$ is equal to 3 in a group. By doing so, the bottom level subtrie can be represented by a 15-bit vector. Prefix expansion can be eliminated, and no reference pointers are required in the bit-map data structures.
(2) All the prefixes in a group have a minimum length of L_{min} bits. The leading L_{min} bits of the prefixes are called the *fixed-length portion*, and the remaining $L_{max} - L_{min}$ bits are called the *extension bits*. A memory efficient search engine based on a novel concept called the *bit-shuffled trie* is used to find the matching fixed-length portion for the input address. Once the matching fixed-length portion is found, the hardware can process the 3 extension bits and the associated bit-map to determine the LMP in the group. The logic circuit for handling 15-bit vector is relatively simple.
(3) We shall exploit the parallelism offered by hardware implementations. Separate hardware pipelines are used to search the 6 groups of prefixes in parallel. A priority selection module is then used to determine the overall LMP of the input address.

We illustrate the general idea of the bit-shuffled trie with a simple example. Consider a group of 5-bit prefixes listed in Table I. The organization of the trie for this set of prefixes and the lookup tables for the 3–2 multibit-trie is shown in Fig. 1. The first 3 bits of the input address is used to access table T_1 with 8 entrires. If a reference address is found in T_1, the last 2 bits of the input address are added to the reference address to access T_2. We can see in this example that 3 out of 8 entries in T_1 are not used, and 12 out of 20 entries in T_2 are not used.

If we rearrange (shuffle) the address bits by swapping the rightmost 2 bits with their left-hand-side neighboring bits, we shall obtain the revised trie structure shown in Fig. 2. Note the changes in the locations of the prefix nodes. We can see that all the nodes on level-3 of the bit-shuffled trie have exactly one prefix in the corresponding subtrie. Hence, if the search key is also shuffled in the same way, we can use the first 3 bits of the bit-shuffled key to index into the first level table T_1. There is no *null* entry in T_1. In this example, the second level index table T_2 is not required. The last 2 bits of the bit-shuffled prefix (residue prefix) can be stored in T_1, and the system only needs to compare the last two bits of the bit-shuffled search key with the stored value to determine if we have a match.

Bit-shuffled trie is motivated by multibit-trie. However, there are three major differences between the two methods.

(1) In the conventional multibit-trie, bits of the input address are processed from left to right. In bit-shuffled trie, bits of the input address can be processed in some other order.
(2) In the conventional multibit-trie, the same group of bits in the input address are processed in a given level. In the bit-shuffled trie, different subsets of bits of the input address can be processed in a given level depending on the search path.
(3) Prefix expansion is required in multibit-trie, where prefixes are expanded to some predefined discrete lengths. In bit-shuffled trie, the prefixes are divided into several length-groups, and we use the bit-map approach to handle the variations in prefix length withing a length-group such that prefix expansion can be avoided.

Organization of the remaining parts of this paper is as follow. We shall give a review of related work in Section 2. Readers who

Table I
Sample set of prefixes.

Prefix	Original value	Bit-shuffled value
A	00001	00100
B	00010	01000
C	01000	00010
D	01011	01110
E	10011	11100
F	10100	10001
G	10110	11001
H	11001	10110