Offset Addressing Approach to Memory-Efficient IP Address Lookup

Kun Huang, Gaogang Xie
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
{huangkun09, xie}@ict.ac.cn

Yanbiao Li
School of Computer and Communication
Hunan University
Changsha, China
liyanbiao2010@gmail.com

Alex X. Liu
Department of CSE
Michigan State University
East Lansing, U.S.A
alexliu@cse.msu.edu

Abstract—**This paper presents a novel offset encoding scheme for memory-efficient IP address lookup, called Offset Encoded Trie (OET). Each node in the OET contains only a next hop bitmap and an offset value, without the child pointers and the next hop pointers. Each traversal node uses the next hop bitmap and the offset value as two offsets to determine the location address of the next node to be searched. The on-chip OET is searched to find the longest matching prefix, and then the prefix is used as a key to retrieve the corresponding next hop from an off-chip prefix hash table. Experiments on real IP forwarding tables show that the OET outperforms previous multi-bit trie schemes in terms of the memory consumption. The OET facilitates the far more effective use of on-chip memory for faster IP address lookup.**

I. INTRODUCTION

IP address lookup algorithms with longest prefix matching have received significant attention in the literature over the past several years [1]. Recently, there has been much renewed research interest in compact data structures for high-speed IP address lookup in a large-scale forwarding table. There are several driving factors that motivate the use of memory-efficient data structures for IP address lookup. First, to keep up with 100Gbps transmission rates [2-3], Internet routers require the use of low memory-footprint data structures for line-speed IP lookup. Second, a core router contains about 310K prefix rules [4], and forwarding tables are expanding exponentially. It necessitates compressing the forwarding data structures to fit in small high-speed memory on line cards. Third, virtual routers [5-6] are emerging as a promising technology, where a common physical platform acts as multiple virtual routers. Each virtual router maintains its own forwarding table. To achieve good scaling in the number of virtual routers, it is critical for each virtual router to minimize the memory usage of its forwarding data structure. Finally, software routers [7-8] have been developed to perform fast IP lookup by leveraging the powerful parallelism of multi-core processors. To achieve high-speeds, it is crucial to store the entire forwarding data structure in the on-chip cache. Thus, we need to use memory-efficient data structures for IP lookup.

There are three major categories of techniques for performing IP address lookup: TCAM-based, hash-based, and trie-based schemes. TCAM-based schemes [9-11] can provide deterministic and high-speed IP lookup, but they suffer from high cost and large power consumption. Hash-based schemes [2-3, 12-17] have been proposed to accelerate the IP lookup, but they require prohibitive amount of high-bandwidth memory that impedes their use in practice. Trie-based schemes [18-26] have been widely used in high-speed Internet routers due to their efficient data structures. Nowadays virtual routers and software routers require efficient trie-based schemes for the performance and scalability.

However, previous trie-based algorithms [18-22] still suffer from high memory consumption. Typically, multi-bit tries are used to inspect a stride of several bits at a time to improve the IP lookup speed, at the cost of less efficient use of memory. Some of the practical algorithms such as Tree Bitmap Trie [21] are based on space-efficient encoding schemes of multi-bit tries, achieving significant reduction of memory usage. In these encoding schemes, each trie node contains the child pointers, the next hop pointers, and the associated bitmaps. A child pointer is of $\log_2 n$ bits, and a next hop pointer is of $\log_2 m$ bits, and a bitmap is of $O(2^s)$, where n denotes the total number of nodes, m denotes the total number of next hops, and s denotes the bit size of a stride. As both n and m increase with the number of prefix rules, these pointers require larger memory usage, which leads to poor scalability of these trie-based algorithms, limiting the lookup performance.

In this paper, we present a novel offset encoding scheme for memory-efficient IP address lookup, called Offset Encoded Trie (OET). In the OET, each node is equivalently compacted by eliminating the child pointers and the next hop pointers. So it contains only a next hop bitmap and an offset value. The next hop bitmap is used to count the index offset between a child and the leftmost non-leaf child, while the offset value is used to compute the identifier distance between a node and its leftmost non-leaf child. The two offsets are leveraged to compute the location address of the next node to be searched. During the lookup procedure, an on-chip OET is traversed to find the longest matching prefix, and then the prefix is used as a key to retrieve the associated next hop from an off-chip prefix hash table. Experiments on real IPv4 and IPv6 prefix tables show that compared to the Tree Bitmap Trie, the OET reduces the memory consumption by up to 76% and 63%, respectively.

II. RELATED WORK

Due to its essential role in Internet routers, IP address lookup is a well-studied problem. A broad spectrum of IP

This work is supported by the National Basic Research Program of China (973) under Grant 2007CB310702, and the National Science Foundation of China under Grant 60873242 and Grant 60903208, and the China Postdoctoral Science Foundation under Grant 20100470023.

address lookup techniques has been proposed in the literature, involving three major categories: TCAM-based, hash-based, and trie-based schemes.

TCAM can perform an IP address lookup in one clock cycle. Most of high-speed routers employ the TCAMs to achieve deterministic and high-speed IP address lookup. But TCAM-based schemes suffer from the excessive power consumption, high cost, and low density. Recently, power-efficient and memory-efficient TCAM-based schemes have been proposed to improve the lookup performance [9-11]. As a SRAM outperforms a TCAM with respect to speed, density, and power consumption, the algorithmic solutions using SRAM/DRAM for IP address lookup are very popular alternatives.

Hash-based schemes [2-3, 12-17] have been proposed for line-speed IP address lookup. These algorithms use the hashing schemes in fast on-chip memory to greatly enhance the lookup throughput. In addition, Bloom filters and their variants [2, 12, 15, 17] have been used to accelerate the hashing process for 100Gbps IP lookup. However, hash-based schemes have the limitation of higher memory bandwidth. The reason is that they often require many expensive multi-port memories to support the parallelism of the IP address lookup.

Trie-based schemes have been the most popular IP address lookup algorithms. While these trie-based algorithms [18-23] become more memory efficient and allow faster lookup, the performance still decreases linearly as the tree depth increases. This makes these algorithms not keep up with the high speeds. To improve the lookup throughput of trie-based algorithms, memory pipelines [24-26] have been proposed to produce one lookup result per clock cycle. As on-chip memory is still small and expensive, the pipeline stages require the compact trie data structure. This facilitates memory efficiency and load balance across multiple stages and multiple pipelines.

Recently, the advent of multi-core processors requires better compact trie data structures in software to scale virtual routers and software routes. The Trie Overlay scheme [27] and Trie Braiding scheme [28] have been proposed to build a compact shared trie data structure for a large number of virtual routers. Orthogonal to these studies, our offset encoding scheme can be used to compress the trie data structure of a single virtual router. So we can use the OET to build scalable virtual routers by reducing the overall memory requirements.

III. OFFSET ENCODING OF TRIES

A. Prior Encoding Schemes of Tries

Binary tries are a natural tree-based data structure for IP address lookup. The trie data structure is used to store a set of prefixes, where a prefix is represented by a node called prefix node. A given IP address is searched by traversing the trie from the root node to match the longest prefix. Figure 1 shows a prefix table and its binary trie on the left. Each node in the trie contains a left and right child pointer as well as a next hop pointer.

The Leaf-Pushed Trie [19] is a popular variant of tries, where prefixes in the internal nodes are pushed down to the leaf nodes. Figure 1 shows a binary Leaf-Pushed Trie on the right. In a s -stride Leaf-Pushed Trie, each node has a list of 2^s pointer entries, each containing either a child pointer or a next hop pointer.

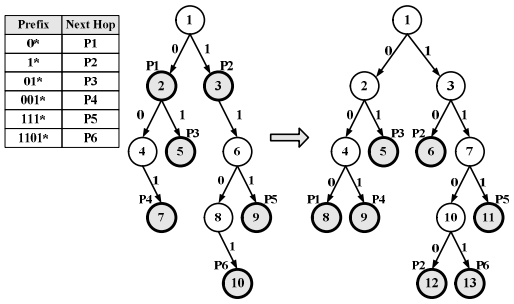


Figure 1. A prefix table and its binary tries

The Lulea Trie [20] and the Tre Bitmap Trie [21] are space-efficient encoding schemes of multi-bit tries. The Lulea Trie is a variant of the Leaf-Pushed Trie, where each node uses a bitmap to indicate the consecutive duplicated pointers, dramatically reducing the number of pointers. The Tree Bitmap Trie is a variant of the non-leaf-pushed multi-bit trie with a stride of s bits. Each node uses an external bitmap (EBMP) of 2^s bits with a single child pointer, and an internal bitmap (IBMP) of $2^s - 1$ bits with a single next hop pointer.

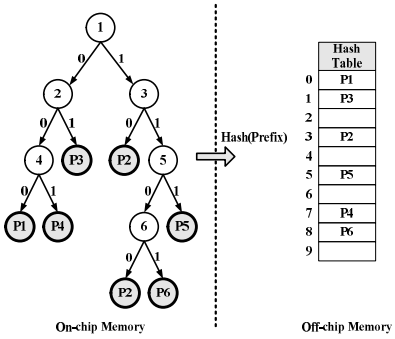


Figure 2. IP address lookup architecture using offset addressing

Beyond these above encoding schemes, we leverage the separation of fast and slow path to achieve high-speed IP lookup with smaller on-chip memory usage. Prior encoding schemes often use an on-chip pointer to point to an off-chip next hop, which leads to larger on-chip memory requirements. Our scheme uses a 1-bit flag to replace the next hop pointer, indicating whether a next hop exists or not. This allows the OET to significantly reduce the memory usage, which can fit in small on-chip memory to improve the IP lookup throughput.

We propose an IP address lookup architecture to accelerate the performance. Figure 2 illustrates our IP address lookup architecture using offset addressing. In this architecture, an OET is stored in the fast path like on-chip SRAM, and a prefix hash table is stored in the slow path like off-chip DRAM. When searching a given IP address, we traverse the on-chip OET to find the longest matching prefix on the address, and then the prefix is used as a key to retrieve the next hop from the

off-chip prefix hash table. Here, we assume that all next hops associated with prefix nodes are stored in off-chip memory.

B. Offset Encoding of Binary Tries

The OET is compact encoding of the Leaf-Pushed Trie. Before constructing the OET, we propose a specific scheme for labeling the trie. This scheme adopts the principle of top-down-left-right labeling. In the Leaf-Pushed Trie, the identifier of the root node is labeled first, and then the identifiers of its non-leaf child nodes are recursively labeled from left to right, and so on. Figure 2 shows an example of labeling the binary Leaf-Pushed Trie. We use this scheme to reduce the offset value size of each node in the OET.

In a binary Leaf-Pushed Trie, each node has 3-tuple fields consisting of the identifiers of its left and right child pointers, as well as the left and right next hop flags. The left or right next hop flag indicates whether the left or right child is a prefix node. If the left or right child is a prefix node, the left or right next hop flag is set as 1; otherwise, the corresponding next hop flag is set as 0. Figure 3 shows node data structures of the binary Leaf-Pushed Trie on the left. The root node 1 contains the left child's identifier 2 and the right child's identifier 3, and sets the left and right next hop flags as 00. As there are 6 non-leaf nodes as shown in Figure 2, a node's identifier has the size of 3 bits. So each node in the Leaf-Pushed Trie requires the total on-chip memory of 8 bits.

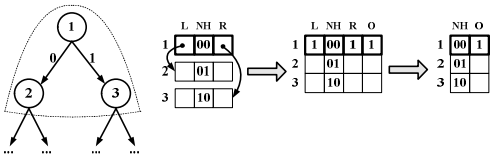


Figure 3. Trasformation of node data structures in the binary trie

We now use the offset encoding scheme to construct a basic binary OET. Each node in a basic binary OET has 4-tuple fields consisting of the left and right child flags, an offset value, and the left and right next hop flags. We use the child flags instead of the child pointers to indicate whether the left or right child exists or not. The offset value is the distance between a node's identifier and the identifier of its non-leaf child. Figure 3 shows node data structures of the basic binary OET in the middle. The root node 1 sets both the left and right child flags as 1, sets the offset value as 1, and sets the left and right next hop flags as 00. As the maximal offset value is 2 as shown in Figure 2, the offset value has the size of 2 bits. Hence, each node in the basic OET requires less memory of 6 bits.

We construct an advanced binary OET in order to further reduce the memory usage. In an advanced binary OET, each node has 2-tuple fields consisting of the left and right next hop flags, and an offset value. Due to the nature of leaf pushing, a non-leaf node has both left and right child, while a leaf node has no any child. We can eliminate both the left and right child flags, so each node maintains only the left and right next hop flags. This is due to the facts that the left and right next hop flags exactly complement the left and right child flags. The left and right next hop flags are enough to indicate whether the left or right child exists or not.

Figure 3 shows node data structures of the advanced binary OET on the right. The root node 1 sets the offset value as 1, and sets the left and right next hop flags as 00. If the input bit is 0, the root node checks to see that the left next hop is 0, indicating that the left child exists but it is not a prefix node, and then the search continues to the left child node 2. If the input bit is 1, the root node checks the next hop flags, and then the search continues to the right child node 3. Hence, each node in the advanced binary OET only requires the memory of 4 bits instead of 8 bits.

Addr	Left Flag	Right Flag	Offset	NH Flag
1	1	1	1	00
2	1	0	2	01
3	0	1	2	10
4	0	0	0	11
5	1	0	1	01
6	0	0	0	11

⇒

Addr	Offset	NH Flag
1	1	00
2	2	01
3	2	10
4	0	11
5	1	01
6	0	11

Figure 4. Binary offset encoded tries

Figure 4 shows all node data structures in both the basic and advanced binary OETs. The left is the basic binary OET, while the right is the advanced binary OET. In the basic binary OET, each node has 4-tuple fields with the size of 6 bits, so the total memory is of $6 \times 6 = 36$ bits. In the advanced binary OET, each node has 2-tuple fields of 4 bits, so the total memory is of $6 \times 4 = 24$ bits. The binary Leaf-Pushed Trie in Figure 3 requires the total memory of $6 \times 8 = 48$ bits. Hence, compared to the binary Leaf-Pushed Trie and the basic binary OET, the advanced OET achieves significant reduction of memory usage.

Suppose that we search an IP address starting with 0101 in the advanced binary OET as shown in Figure 4. The search starts with the root node 1. For the first bit 0 of the address, the root node 1 checks to see that the left next hop flag is 0, and then uses the offset value 1 plus its location address 1 to compute the location address of the left child. The search continues to the child node 2. For the second bit 1, the node 2 checks to see that the right next hop flag is 1, indicating that the right child is a prefix node, and then the search terminates, producing the longest matching prefix 01*.

C. Offset Encoding of Multi-bit Tries

A multi-bit trie with a stride of s is generally used to boost the lookup throughput of a binary trie by a factor of s. However, the node size grows exponentially with the stride size, which leads to the rapidly increasing memory usage of the multi-bit trie. When the stride size is too large, the increase in node size significantly outpaces the reduction in the number of nodes. Recently, the dynamic programming technique [19] has been used to minimize the memory usage of a multi-bit Leaf-Pushed Trie. Figure 5 shows an example of expanding multi-bit tries. The left is the binary Leaf-Pushed Trie, while the right is the multi-bit Leaf-Pushed Trie with a stride of 2.

We can construct a multi-bit OET in a manner similar to the construction of the binary OET. In practice, a multi-bit OET is derived from a multi-bit Leaf-Pushed Trie. Initially, a binary Leaf-Pushed Trie is expanded to a multi-bit Leaf-Pushed Trie with a stride of s. In a basic multi-bit OET, each node contains a child bitmap called CBMP, an offset value, and a next hop bitmap called NBMP. The offset value means the distance between a node's identifier and the identifier of its

leftmost non-leaf child due to multiple child nodes. In an advanced multi-bit OET, each node contains only a NBMP and an offset value. This is due to the facts that the CBMP exactly complements the NBMP.

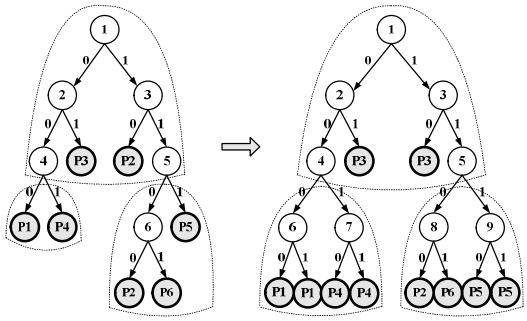


Figure 5. Expanding multi-bit tries

Figure 6 shows all node data structures in both the basic and advanced multi-bit OETs. The left is the basic multi-bit OET, while the right is the advanced multi-bit OET. Each node on the left contains a NBMP, a CBMP, and an offset value, while each node on the right contains only a NBMP and an offset value. Both the NBMP and CBMP have the size of 4 bits, and the offset value has the size of 1 bit. So the basic multi-bit OET requires the total memory of $3 \times 9 = 27$ bits, while the advanced multi-bit OET only requires the total memory of $3 \times 5 = 15$ bits. Hence, the advanced multi-bit OET reduces less half memory than the basic multi-bit OET.

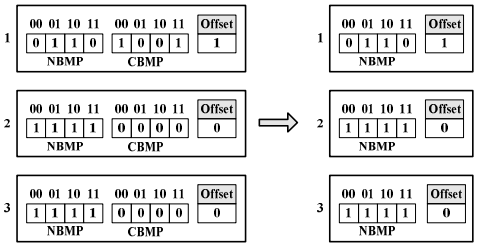


Figure 6. Multi-bit offset encoded tries

Suppose that we search an IP address starting with 1110 in the 2-stride OET as shown in Figure 6. For the first two bits 11 of the address, the root node 1 checks to see that *NBMP*[11] is 0, indicating a child node to be searched. We count the number of 0s before the index $(11)_2 = 3$ in the NBMP as 1, and then use the offset value 1 plus its location address 1 to compute the location address of the child node as $1+1+1=3$. The search continues to the child node 3. For the second two bits 10, the node 3 checks to see that *NBMP*[10] is 1, indicating a match, and then the search terminates, producing the longest matching prefix 111*. Finally, we use '111' as a key to retrieve the corresponding next hop *P*5 from an off-chip prefix hash table.

The lookup in the OET is similar to that in the Tree Bitmap Trie [21]. The search proceeds recursively, starting from the root, until to the leaf. Each traversal node uses the NBMP to determine whether the search terminates or continues to the next node. The lookup algorithm in the multi-bit OET is given as follows:

Algorithm: Multi-bit Offset Encoded Trie Lookup

```
1: Search Offset Encoded Trie (ip_addr)
2: oet is the multi-bit Offset-Encoded Trie;
3: s is the stride size;
4: current_node = oet.getroot(); // get the root node
5: for (i = 1; i <= len(ip_addr)/s; i++) do
6:   bit_substr = ip_addr.getbits(i, s); // get the bit substring
7:   j = 2^bit_substr-1; // calculate the index value
8:   if (current_node.nbmp[j] == 1) do
9:     match_len = i*s; // calculate the length of matching prefix
10:    return ip_addr[match_len];
11:  else // it continues to search a child node
12:    index_offset = popcount0s(current_node.nbmp, j);
13:    child_location = current_node.location
                        + current_node.offset + index_offset;
14:    current_node = oet.getnode(child_location);
15:  end do
16: end do
```

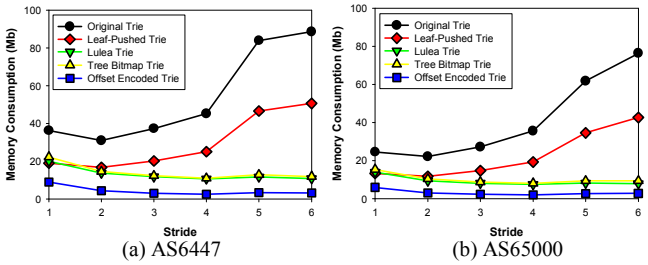
D. Construction of Prefix Hash Table

In our IP lookup architecture, an off-chip prefix hash table is used to retrieve the next hop associated with the longest matching prefix. The performance of the prefix hash table has direct impact on the IP lookup throughput and the memory consumption. In this paper, we use a simple and efficient multiple-choice hashing scheme proposed in [2, 12-13]. In this scheme, there are $k \geq 2$ independent hash functions, and each table bucket has n cells that contain up to n pairs of $\{prefix, next\ hop\}$. Each prefix is hashed k times into k candidate table buckets, and only one table bucket with the lowest load are chosen to store the prefix. A prefix lookup needs to access k table buckets using the same k hash functions in parallel. In each of the k table buckets, all prefix pairs are sequentially searched to find the next hop associated with the matching prefix. Multi-port memories or multiple parallel memory modules can be used to improve the access bandwidth of our prefix hash table.

IV. Experimental Results

We conduct simulation experiments on real IP prefix tables to evaluate the performance of OET-based IP address lookup algorithm, in terms of the memory consumption. For evaluation purposes, we obtain four reprehensive real-world prefix tables [4]. AS6447 and AS65000 are large-scale IPv4 BGP tables that contain about 310K and 217K prefixes respectively. AS2 and AS1221 are small-scale IPv6 BGP tables that contain about 2K and 0.9K prefixes respectively.

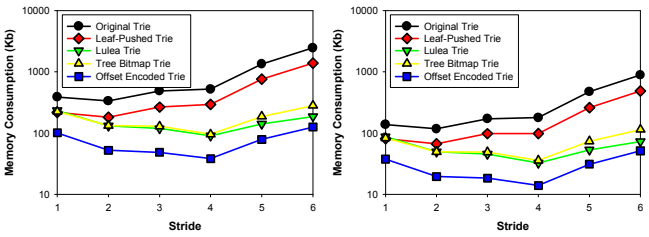
Figure 7 depicts the memory consumption on two IPv4 prefix tables. As shown in Figure 7(a), when the stride size increases from 1 to 6, the OET requires only the memory usage of 2.51Mbits-9.08Mbits. Experiments on the table AS6447 show that compared to the Original Trie, Leaf-Pushed Trie, Lulea Trie, and Tree Bitmap Trie, the OET reduces the memory consumption by 75-96.4%, 52.5-93.8%, 54.8-76.6%, and 59.3-77.1%, respectively. As shown in Figure 7(b), the OET requires only the memory usage of 2.03Mbits-5.98Mbits. Experiments on the table AS65000 show that compared to the Original Trie, Leaf-Pushed Trie, Lulea Trie, and Tree Bitmap Trie, the OET reduces the memory consumption by 75.7-96.4%, 55-93.5%, 57.1-73%, and 61.1-74.9%, respectively.



(a) AS6447 (b) AS65000

Figure 7. Memory consumption on two IPv4 prefix tables

Figure 8 depicts the memory consumption on two IPv6 prefix tables. As shown in Figure 8(a), when the stride size increases from 1 to 6, the OET requires only the memory usage of 38Kbits-125Kbits. Experiments on the table AS2 show that compared to the Original Trie, Leaf-Pushed Trie, Lulea Trie, and Tree Bitmap Trie, the OET reduces the memory consumption by 74-94.9%, 53.6-91%, 32.9-59.9%, and 55.4-62.4%, respectively. As shown in Figure 8(b), the OET requires only the memory usage of 14Kbits-51Kbits. Experiments on the table AS1221 show that compared to the Original Trie, Leaf-Pushed Trie, Lulea Trie, and Tree Bitmap Trie, the OET reduces the memory consumption by 72.7-94.2%, 53.8-89.5%, 29.9-60.5%, and 55-62.5%, respectively.



(a) AS2 (b) AS1221

Figure 8. Memory consumption on two IPv6 prefix tables

V. CONCLUSIONS

In this paper, we propose an Offset Encoded Trie (OET) for memory-efficient IP address lookup. Each node in the OET contains only a next hop bitmap and an offset value, without the child pointers and the next hop pointers. When performing a lookup in the OET, each node uses the next hop bitmap and the offset value as two offsets to determine whether the search terminates or compute the location address of the next node to be searched. The performance evaluation on real-world IP prefix tables shows that the OET requires significantly less memory usage than previous multi-bit trie schemes. For instance, experiments on real IPv4 and IPv6 prefix tables show that compared to the Tree Bitmap Trie, the OET reduces 60-76% and 55-63% of the memory consumption, respectively.

REFERENCES

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network, vol.15, no.2, pp.8-23, 2001.

[2] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced Bloom filters for 100Gbps core router line cards," in IEEE INFOCOM, 2009, pp.2518-2526.

[3] M. Bando and H. J. Chao. "FlashTrie: hash-based prefrix-compressed trie for IP route lookup beyond 100Gbps," in IEEE INFOCOM, 2010, pp.1-9.

[4] BGP table, http://bgp.potaroo.net.

[5] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In VINI veritas: realistic and controlled network experimentation," in ACM SIGCOMM, 2006, pp.3-14.

[6] M. B. Anwer and N. Feamster, "Building a fast, virtualized data plane with programmable hardware," ACM SIGCOMM Computer Communications Review, vol.40, no.1, pp.75-82, 2010.

[7] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers," in ACM SOSP, 2009, pp.15-28.

[8] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in ACM SIGCOMM, 2010.

[9] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: power-efficient TCAMs for forwarding engines," in IEEE INFOCOM, 2003, pp.42-52.

[10] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," IEEE/ACM Transactions on Networking, vol.14, no.4, pp.863-875, 2006.

[11] W. Lu and S. Sahni, "Low power TCAMs for very large forwarding tables," IEEE Transactions on Networking, vol.18, no.3, pp.948-959, 2010.

[12] A. Border and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in IEEE INFOCOM, 2001, pp.1454-1463.

[13] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using Bloom filters," in ACM SIGCOMM, 2003, pp.201-212.

[14] J. T. M. Waldvogel, G. Varghese, and B. Plattner, "Scalable high speed IP routing lookups," in ACM SIGCOMM, 1997, pp.25-36.

[15] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: a storage-efficient collision-free hash-based network processing architecture," in ISCA, 2006, pp.203-215.

[16] S. Kumar, J. Tuner, P. Crowley, and M. Mitzenmacher, "HEXA: compact data structures for faster packet processing," in IEEE ICNP, 2007, pp.246-255.

[17] H. Yu, R. Mahapatra, and L. Bhuyan, "A hash-based scalable IP lookup using Bloom and fingerprint filters," in IEEE ICNP, 2009, pp.264-273.

[18] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in IEEE INFOCOM, 1998, pp.1240-1247.

[19] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," in ACM SIGMETRICS, 1998, pp.1-11.

[20] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in ACM SIGCOMM, 1997, pp.3-14.

[21] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," SIGCOMM Computer Communications Review, vol.34, no.2, pp.97-122, 2004.

[22] H. Song, J. Turner, and J. Lockwood, "Shape shift tries for faster IP lookup," in IEEE ICNP, 2005, pp.358-367.

[23] H. Song, M Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using shape graphs," in IEEE ICNP, 2009, pp.73-82.

[24] J. Hasan and T. N. Vijaykumar, "Dynamic pipelining: making IP lookup truly scalable," in ACM SIGCOMM, 2005, pp.205-216.

[25] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in ISCA, 2005, pp.123-133.

[26] W. Jiang and V. K. Prasanna, "Beyond TCAM: an SRAM-based multi-pipeline architecture for terabit IP lookup," in IEEE INFOCOM, 2008, pp.1786-1794.

[27] J. Fu and J. Rexford, "Efficient IP address lookup with a shared forwarding table for multiple virtual routers," in ACM CoNEXT, 2008.

[28] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Building scalable virtual routers with trie braiding," in IEEE INFOCOM, 2010, pp.1-9.