A Storage-Efficient IP Address Lookup Algorithm Based on Offset Addressing (Postprint)

Authors: Huang Kun, Xie Gaogang, Li Yanbiao, Liu Xiangyang

Date: 2017-03-09T00:00:00+00:00

Abstract

The rapid growth of network bandwidth and the advent of emerging technologies such as virtual routers and software routers have urgently necessitated storage-efficient IP address lookup algorithms. Existing practical IP address lookup algorithms are based on space-efficient encodings of multi-way tries, such as Tree Bitmap Trie. However, in these encoding methods, each node maintains multiple pointers as well as multiple associated bitmaps, resulting in substantial storage space overhead for the trie and making it difficult to store the information in high-speed on-chip memory, thereby limiting IP lookup performance. This paper proposes a novel Offset Encoded Trie (OET) to achieve storage-efficient IP address lookup. Each node in the Offset Encoded Trie maintains only one next-hop bitmap and one offset value, without requiring child pointers or next-hop pointers. Each node utilizes the next-hop bitmap and offset value to calculate the storage address of the next node to be searched. During the IP address lookup process, the on-chip Offset Encoded Trie finds the longest matching prefix, while the off-chip prefix hash table looks up the next-hop information associated with that prefix. This paper conducts experimental evaluations using real-world IP prefix rule sets, and the experimental results demonstrate that, compared to existing multi-way trie encoding methods, the Offset Encoded Trie significantly reduces storage space overhead.

Full Text

Preamble

Vol. 8 No. 6
Information Technology Letters

A Storage-Efficient IP Address Lookup Algorithm Based on Offset Addressing
Kun Huang, Gaogang Xie, Yanbiao Li, Xiangyang Liu

Abstract

The rapid growth of network bandwidth and the emergence of new technologies such as virtual routers and software routers have created an urgent need for storage-efficient IP address lookup algorithms. Existing practical IP address lookup algorithms rely on space-efficient encodings of multi-way tries, such as the Tree Bitmap Trie. However, these encoding methods require each node to maintain multiple pointers and associated bitmaps, resulting in substantial storage overhead that prevents the data structure from fitting in high-speed on-chip memory, thereby limiting IP lookup performance. This paper proposes a novel Offset Encoded Trie (OET) for storage-efficient IP address lookup. Each node in an OET maintains only one next-hop bitmap and one offset value, eliminating the need for child pointers and next-hop pointers. Each node uses its next-hop bitmap and offset value to compute the memory address of the next node to search. During IP address lookup, the on-chip OET finds the longest matching prefix, while an off-chip prefix hash table retrieves the next-hop information associated with that prefix. Experimental evaluation using real IP prefix rule sets demonstrates that OET significantly reduces storage overhead compared to existing multi-way trie encoding methods.

Keywords: router, IP address lookup, longest prefix matching, trie

1. Introduction

IP address lookup is a core function of Internet routers, typically implemented using Longest Prefix Matching (LPM), which searches for the longest prefix rule in an IP prefix rule set that matches the destination IP address of an incoming packet [1]. Core routers contain over a million rules, with each IP prefix rule consisting of an IP address prefix and its associated next-hop information, which includes forwarding port numbers and router MAC addresses.

IP routers can operate in either static or dynamic modes. In static mode, routers periodically update their IP prefix rule sets offline, which is suitable for fast IP lookup. In dynamic mode, routers update rule sets online in real-time, which may interrupt lookup operations. Caesar et al. [2] have shown that dynamic update strategies for IP prefix rule sets degrade routing performance. Therefore, this paper focuses on static IP address lookup algorithms that support offline updates to achieve fast packet forwarding.

The explosive growth in network bandwidth and traffic volume has created scalability challenges for IP address lookup algorithms, particularly in meeting the throughput and space requirements for high-speed packet processing. As a compute-intensive operation on the critical data path of routers, IP address lookup has become a performance bottleneck. Recent network technologies and applications urgently demand storage-efficient IP address lookup algorithms for several reasons:

- Internet backbone link bandwidth has increased from 40 Gbps to 100 Gbps

[3-4], requiring reduced storage space for IP lookup algorithms to enable line-rate processing.

- IP prefix rule sets grow exponentially with network size; current core routers contain approximately 310,000 IP prefix rules [5]. Compressing the forwarding data structure is essential for storing and searching entire rule sets in on-chip memory.
- The increasingly popular virtual router technology is an emerging programmable networking technology that runs multiple virtual routers concurrently on a single physical hardware platform, with each maintaining its own IP prefix rule table. Scalable virtual routers require minimizing per-router storage overhead to support more concurrent instances.
- Widely adopted software router technology leverages the parallel computing power of multi-core processors to implement high-speed routers in software. High-performance software routers require storing and searching entire IP prefix rule sets in on-chip cache to accelerate IP lookup on multi-core processors.

These trends have motivated researchers to revisit compact forwarding data structures for IP address lookup algorithms to achieve fast and scalable packet forwarding.

IP address lookup algorithms can be classified into three categories: TCAM-based, hash-based, and trie-based. TCAM-based algorithms [10-12] provide deterministic, high-speed lookup in one clock cycle but suffer from high cost and power consumption. Hash-based algorithms [3-4, 13-18] use hash tables to accelerate lookup but require high memory bandwidth. Trie-based algorithms [19-27] are widely used in high-speed routers, firewalls, and NIDS/NIPS due to the efficiency of tries as a data structure. Current virtual and software routers primarily employ trie-based algorithms to meet scalability and programmability requirements.

However, existing trie-based IP address lookup algorithms [19-23] still suffer from large storage overhead. These algorithms typically use multi-way tries to represent IP prefix rule sets, examining multiple bits of the IP address at once to increase throughput at the cost of higher space consumption. Many practical implementations use space-efficient encoding methods for multi-way tries, such as Tree Bitmap Trie [22], to reduce storage requirements. In these encodings, each trie node maintains child pointers, next-hop pointers, and associated bitmaps. Child pointers (size: 2 log n bits) point to child nodes, while next-hop pointers (size: O(log m) bits) point to next-hop information, where n is the number of nodes and m is the number of next-hop entries. The bitmap size is 2^s bits, where s is the stride length in bits. As n and m grow with the number of prefix rules, the pointers consume increasing storage space, limiting scalability. For example, in a Tree Bitmap Trie with 310K IP prefix rules, each node maintains a 2-bit external bitmap with a 3-bit child pointer and a 3-bit internal bitmap with a 3-bit next-hop pointer. The resulting binary tree bitmap trie contains 219K nodes and occupies 15 Mb of storage—too large for modern

10 Mb on-chip SRAM [4], thus limiting lookup performance.

[Figure 1: see original paper]

To address these limitations, this paper proposes a storage-efficient IP address lookup algorithm based on offset addressing. The algorithm uses an Offset Encoded Trie (OET) to represent IP prefix rule sets, dramatically reducing storage requirements. Each OET node maintains only one next-hop bitmap and one offset value, eliminating child and next-hop pointers. The next-hop bitmap records the index offset between child nodes and the leftmost non-leaf child, while the offset value records the identifier distance between each node and its leftmost non-leaf child. Each node uses these two values to compute the memory address of the next search node. During lookup, the on-chip OET finds the longest matching prefix, while an off-chip prefix hash table retrieves the associated next-hop information. For example, Figure 1 shows a binary trie with seven nodes. While a Tree Bitmap Trie node requires a 2-bit external bitmap with a 3-bit child pointer and a 3-bit internal bitmap with a 3-bit next-hop pointer, an OET node requires only a 2-bit next-hop bitmap and a 1-bit offset value—just 3 bits per node. Experimental results show that OET significantly reduces storage overhead compared to existing multi-way trie encoding methods, achieving 76% and 63% reductions for real IPv4 and IPv6 prefix rule sets, respectively, compared to Tree Bitmap Trie.

The remainder of this paper is organized as follows: Section 2 reviews related work on IP address lookup. Section 3 details the proposed storage-efficient IP address lookup algorithm based on offset addressing. Section 4 presents experimental results, and Section 5 concludes the paper.

2. Related Work

IP address lookup is critical to router performance and has been extensively studied. Recent proposals can be classified into three categories: TCAM-based, hash-based, and trie-based algorithms.

TCAM (Ternary Content-Addressable Memory) enables content-based address lookup. High-end routers primarily use TCAM-based IP lookup algorithms, which complete one lookup per clock cycle, providing deterministic, high-speed search. However, TCAM suffers from high power consumption, cost, and low density, limiting large-scale deployment. Recent work has proposed energy- and storage-efficient TCAM algorithms [10-12] to reduce power and storage overhead. Compared to TCAM, SRAM offers superior access speed, storage density, and energy efficiency, making SRAM-based algorithms an attractive alternative for low-power, high-speed lookup.

Hash-based IP lookup algorithms [3-4, 13-18] use on-chip SRAM hash tables for line-rate processing. Researchers have employed Bloom filters and their variants [3, 14, 16, 18] to reduce off-chip DRAM accesses, accelerating hash-based lookup to 100 Gbps. However, these algorithms require high memory

bandwidth, necessitating expensive multi-port memories for parallel processing.

Trie-based IP lookup algorithms are the most popular packet processing algorithms, widely deployed in switches/routers, firewalls, and NIDS/NIPS. Recent proposals [19-24] have improved storage efficiency and lookup throughput. However, their performance degrades linearly with trie depth, preventing line-rate processing. To increase throughput, researchers have proposed memory pipelining [25-27] to complete one lookup per clock cycle. Since on-chip memory is small and expensive, each pipeline stage requires space-efficient trie structures to enable storage-efficient, load-balanced multi-pipeline designs.

Multi-core processors integrate multiple parallel CPU cores on a single chip to enhance computational power. To improve router scalability and programmability, multi-core technology is widely used in virtual and software routers, requiring storage-efficient trie structures for high-speed software-based IP lookup. Recent work has proposed trie overlay and trie braiding methods to build compact, shared trie structures for multiple virtual routers. Complementing these efforts, our offset encoding method compresses the trie data structure for a single virtual router, reducing overall storage overhead for multiple virtual routers and enabling scalable virtualization.

3. Offset-Based IP Address Lookup

3.1 Existing Trie Encoding Methods

A binary trie is a fundamental tree data structure for IP address lookup. A trie represents a set of IP prefixes, with each prefix represented by a prefix node on a path from the root. During lookup, the trie is traversed from the root until the longest matching prefix is found. In binary tries, each bit of the IP address determines the traversal path: 0 for the left child, 1 for the right child. Figure 2 shows a prefix table and its binary trie, where shaded nodes represent prefix nodes. Each node contains left and right child pointers and a next-hop pointer. When searching for IP address 1101, the shaded node matching prefix 1* is found initially, but subsequent traversal matches the longer prefix 1101* at shaded node 10, returning next-hop information 6P.

[Figure 2: see original paper]

Leaf-Pushed Trie [20] is a popular trie variant that pushes prefixes from internal nodes down to leaf nodes, ensuring all prefix nodes are leaves. The right side of Figure 2 shows the leaf-pushed version, where internal node 3' s prefix 1* is pushed to leaf nodes 6 and 12, expanding into prefixes 10* and 1100*. In a leaf-pushed trie with stride s, each node contains 2^s pointer entries, each being either a child pointer or a next-hop pointer. Figure 3(a) shows a leaf-pushed trie with stride 2, where each node has four pointer entries. Child pointers are 2 bits, and next-hop pointers are 3 bits. The root node' s entries (01) and (10) contain next-hop pointers to 3P and 2P, while other entries contain child pointers. Since each entry holds either a child or next-hop pointer, leaf-pushed

tries reduce storage overhead by approximately half compared to original tries.

[Figure 3: see original paper]

Lulea Trie [21] and Tree Bitmap Trie [22] are space-efficient multi-way trie encoding methods. Lulea Trie is a leaf-pushed variant that uses a bitmap to represent sequences of repeated pointers, eliminating redundancy. Figure 3(b) shows a Lulea Trie with stride 2, where each node contains a 4-bit bitmap: 1 indicates a unique pointer, 0 indicates a repeated pointer. The lower-left node contains only two next-hop pointers (to 1P and 4P) and a bitmap 1010. Tree Bitmap Trie is a non-leaf-pushed variant. With stride s, each node contains a 2^s-bit external bitmap (EBMP) with a child pointer and a (2^s - 1)-bit internal bitmap (IBMP) with a next-hop pointer. Figure 3(c) shows a Tree Bitmap Trie with stride 2, where each node has a 4-bit external bitmap with a 2-bit child pointer and a 3-bit internal bitmap with a 3-bit next-hop pointer. The root node contains a next-hop head pointer to a linked list of 1P and 2P, a child head pointer to a list of consecutive child nodes, and sets the external and internal bitmaps to 1101 and 011, respectively.

[Figure 4: see original paper]

Unlike these methods, our design separates fast and slow paths: using on-chip high-speed memory to reduce off-chip memory accesses for high-speed lookup. Existing methods typically use on-chip pointers to off-chip next-hop information, requiring large on-chip storage. We replace next-hop pointers with a 1-bit flag indicating next-hop presence, significantly reducing trie storage overhead so the entire structure fits in on-chip memory, thereby increasing lookup throughput.

We propose an offset-addressing-based IP lookup architecture consisting of an on-chip OET and an off-chip prefix hash table. The OET, stored in on-chip SRAM, finds the longest matching prefix for an IP address, while the off-chip prefix hash table, stored in DRAM, retrieves the associated next-hop information. Figure 4 illustrates this architecture. To lookup IP address 0010, the on-chip OET is traversed to find the longest matching prefix 001. *Using 001 as the key, a hash function searches the off-chip prefix hash table to return the next-hop information 4P associated with 001.*

3.2 Binary Offset Encoded Trie

Offset Encoded Trie (OET) is a compact encoding of leaf-pushed tries. Before constructing an OET, we propose a node naming scheme using a "top-down-left-right" order to assign node identifiers. After leaf-pushing, the root is assigned identifier 1, followed by its non-leaf children from left to right, recursively. Leaf nodes need not be named since they are embedded in their parent structures. This breadth-first naming reduces offset value sizes. Figure 4 shows an example of leaf-pushed trie node naming, where non-leaf nodes are named top-down-left-right and leaf nodes are named by their next-hop information.

In a binary leaf-pushed trie, each node contains three fields: left child node iden-

tifier, right child node identifier, and left/right child next-hop flags indicating whether each child is a prefix node. A flag value of 1 indicates a prefix node; 0 indicates otherwise. Figure 5 (left) shows the node data structure. Root node 1 contains left child identifier 2, right child identifier 3, and flags 00. With six non-leaf nodes total (Figure 4), child identifiers require 3 bits, making each node occupy 8 bits of on-chip storage. Note that all prefix node next-hop information is stored off-chip.

[Figure 5: see original paper]

Based on this leaf-pushed trie, we construct a basic binary OET using offset encoding. Each node contains four fields: left child flag, right child flag, offset value, and left/right child next-hop flags. Child flags replace child identifiers to indicate child presence. The offset value represents the identifier distance between a node and its leftmost non-leaf child. Figure 5 (dashed box) shows the basic binary OET node structure. Root node 1 sets both child flags to 1, offset to 1, and next-hop flags to 00. With a maximum offset of 2 (Figure 4), the offset value requires 2 bits, making each basic binary OET node occupy only 6 bits.

To further reduce storage, we construct an advanced binary OET where each node contains just two fields: left/right child next-hop flags and offset value. In leaf-pushed tries, every non-leaf node has both left and right children, while leaf nodes have none. We observe that in the basic binary OET, the next-hop flags are complementary to the child flags—next-hop flags alone can indicate child presence. Thus, the advanced binary OET eliminates child flags, retaining only next-hop flags and offset. Figure 5 (shaded box) shows this structure. Root node 1 has next-hop flags 00 and offset 1. When the search bit is 0, root node 1' s left child flag 0 indicates a non-prefix left child exists, so traversal continues to left child node 2 using offset 1. When the bit is 1, the right child flag 0 indicates a non-prefix right child, leading to node 3. Each advanced binary OET node occupies only 4 bits—far less than the 8 bits of a leaf-pushed trie node.

[Figure 6: see original paper]

Figure 6 compares the structures. The basic binary OET has 6 nodes $\times$ 6 bits = 36 bits total, while the advanced version has 6 nodes $\times$ 4 bits = 24 bits. The original leaf-pushed trie requires 6 nodes $\times$ 8 bits = 48 bits. Thus, the advanced binary OET reduces storage by 50% compared to leaf-pushed tries and by 33% compared to the basic OET.

To lookup IP address 0101 in the advanced binary OET (Figure 6): For the first bit 0, root node 1' s left next-hop flag 0 indicates a non-prefix left child exists. Using offset 1 and location 1, the left child' s address is calculated as 2, and traversal continues to node 2. For the second bit 1, node 2' s right next-hop flag 1 indicates a prefix node, terminating the search and returning the longest matching prefix 01*.

3.3 Multi-Bit Offset Encoded Trie

Internet routers typically use multi-way tries to accelerate lookup throughput. However, node size grows exponentially with stride length, rapidly increasing total storage. When stride length becomes too large, the node size increase outweighs node count reduction, degrading performance. Dynamic programming algorithms [20] have been proposed to minimize storage for multi-way leaf-pushed tries. Figure 7 shows the expansion from a binary leaf-pushed trie to a stride-2 multi-way leaf-pushed trie, where prefix nodes 1P, 4P, and 5P become leaf nodes.

[Figure 7: see original paper]

Similar to the binary case, we construct a multi-way OET from a multi-way leaf-pushed trie using offset encoding. In the basic multi-way OET, each node contains a child bitmap (CBMP), offset value, and next-hop bitmap (NBMP). The offset represents the identifier distance to the leftmost non-leaf child. In the advanced multi-way OET, each node contains only an NBMP and offset value because the CBMP and NBMP are complementary—the NBMP alone can indicate child presence.

[Figure 8: see original paper]

Figure 8 shows both structures. In the basic version, each node has NBMP, CBMP, and offset; in the advanced version, only NBMP and offset remain. With 4-bit NBMP/CBMP and 1-bit offset, the basic multi-way OET requires 3 nodes $\times$ 9 bits = 27 bits, while the advanced version needs only 3 nodes $\times$ 5 bits = 15 bits.

To lookup IP address 1110 in the stride-2 advanced multi-way OET (Figure 8): For the first two bits 11, root node 1' s NBMP is 0, indicating a child search is needed. In the NBMP, counting zeros before bit (11) = 3 yields 1 zero. With offset 1 and location 1, the child address is 1 + 1 + 1 = 3. For the last two bits 10, node 3' s NBMP is 1, indicating a successful match. The longest matching prefix 111* is returned, and using 111 as the hash key, the off-chip prefix hash table returns next-hop information 5P.

[Figure 9: see original paper]

The OET lookup process resembles Tree Bitmap Trie traversal, iterating from the root until reaching a leaf. At each step, s bits of the IP address are read to determine if the OET node matches and to decide the search path. If NBMP is 1, the search succeeds and returns the longest matching prefix; otherwise, it continues to the 2^s-th child. Given a node at location *Location*, the address of the 2^s-th child is calculated as *Location + Offset + Index*, where *Index* is the count of zeros before the 2^s-th bit in NBMP. Figure 9 shows the pseudocode for multi-bit OET lookup.

3.4 Variable-Stride Offset Encoded Trie

We extend the fixed-stride OET to variable-stride OET. For a given prefix rule set, Srinivasan et al. [20] proposed storage minimization methods for both fixed- and variable-stride leaf-pushed tries. Variable-stride OET is a compact encoding of variable-stride leaf-pushed tries, inheriting their advantages while further compressing storage. The construction process mirrors that of fixed-stride OET, but each node contains a variable-size NBMP and offset value. Figure 10 shows a variable-stride OET with maximum stride 2 and its node structures. Root node 1 and node 3 maintain 4-bit NBMPs, while node 2 has a 2-bit NBMP.

[Figure 10: see original paper]

The lookup process is similar to fixed-stride OET. For IP address 0010: The first two bits 00 yield NBMP 0 at the root, so using offset 1 and location 1, the next address is 1 + 1 + 0 = 2. The next bit 1 at node 2 gives NBMP 1, indicating a successful match with longest prefix 001*. Using 001 as the hash key, the off-chip prefix hash table returns next-hop information 4P.

3.5 Prefix Hash Table Construction

In our architecture, the off-chip prefix hash table retrieves next-hop information for the longest matching prefix. Its performance directly impacts lookup throughput and storage overhead. Recent work has proposed efficient hash tables such as Fast Hash [30], Peacock Hash [31], Cuckoo Hash [32], and One-Move Hash [33] to reduce storage and collision probability. These can be applied to our prefix hash table design.

We adopt a simple yet efficient multi-choice hashing method widely used in hash-based IP lookup [3, 13-14]. The multi-choice hash table uses k 2 independent hash functions, with each bucket storing n elements of the form <prefix, next-hop>. During insertion, the prefix is mapped to k buckets using the hash functions, and the least-loaded bucket is chosen. During lookup, the same k hash functions map the prefix to k buckets, which are searched in parallel. Hardware such as multi-port memories or parallel memory modules accelerates this process. Similar to prior work [3, 14], we partition prefix rule sets by length into groups, each stored in a separate multi-choice hash table. Unlike [3, 14], our on-chip OET produces exactly one longest matching prefix, so only one off-chip hash table needs to be accessed.

3.6 Offline Updates for OET

Network topology changes and transient routing failures cause frequent IP routing table updates. The control plane computes new prefixes and next-hop information to update line card lookup data structures. Caesar et al. [2] showed that dynamic real-time updates consume excessive resources and can cause update storms, disrupting normal forwarding. Therefore, we adopt offline updates:

the entire on-chip OET is updated in backup mode without interrupting packet forwarding or causing routing errors.

Incremental updates include next-hop modifications and prefix changes. Next-hop updates only modify the off-chip prefix hash table, leaving the on-chip OET unchanged. Prefix insertion and deletion require building a backup off-chip leaf-pushed trie and updating the on-chip OET and off-chip hash table as needed. Deletion only updates the off-chip hash table. Insertion is more complex: if no new leaf-pushed trie node is created, only the off-chip hash table is updated; if new nodes are created, the backup trie is updated first, then the entire on-chip OET is rebuilt, and the off-chip hash table is updated with expanded prefixes. This offline update approach ensures uninterrupted, correct IP lookup. Determining optimal offline update intervals remains future work.

4. Experimental Evaluation

We evaluate the OET-based IP lookup algorithm using real IP prefix rule sets. We implemented four existing multi-way trie encoding methods in C/C++: original trie, leaf-pushed trie, Lulea trie, and Tree Bitmap Trie. Our comparison focuses on storage overhead.

For evaluation, we selected four typical real-world IP prefix rule sets [5]: AS6447, AS65000, AS2, and AS1221. AS6447 and AS65000 are large-scale IPv4 prefix sets from BGP routers, containing approximately 31K and 21K rules, respectively. AS2 and AS1221 are smaller IPv6 prefix sets, containing about 2K and 900 rules, respectively. Table 1 shows the prefix counts and trie node counts for these sets.

4.1 IPv4 Prefix Rule Sets

The IPv4 sets AS6447 and AS65000 contain 310,344 and 21,795 prefixes, respectively. Figure 11 shows their prefix length distributions, which range from 8 to 32 bits. Prefix length 24 dominates, accounting for 51% of AS6447 and 35% of AS65000.

[Figure 11: see original paper]

For IPv4, we compare storage overhead across fixed strides. Table 1 shows node counts decreasing as stride increases from 1 to 6: AS6447 nodes drop from 500K to 41K, while AS65000 nodes drop from 348K to 36K. Similar results hold for variable strides.

Figure 12 shows storage overhead for IPv4 sets. For AS6447 (Figure 12a), OET requires only 2.51–9.08 Mb across strides 1–6, representing reductions of 75–96.4%, 52.5–93.8%, 54.8–76.6%, and 59.3–77.1% compared to original trie, leaf-pushed trie, Lulea trie, and Tree Bitmap Trie, respectively. For AS65000 (Figure 12b), OET requires 2.03–5.98 Mb, achieving reductions of 75.7–96.4%, 55–93.5%, 57.1–73%, and 61.1–74.9%, respectively.

[Figure 12: see original paper]

In summary, OET significantly reduces storage overhead across all stride lengths for real IPv4 prefix sets.

4.2 IPv6 Prefix Rule Sets

Current IPv6 networks are smaller; AS2 and AS1221 contain only 2,259 and 932 prefixes, respectively. Figure 13 shows their prefix length distributions, ranging from 16 to 64 bits. Prefix length 32 dominates, accounting for 63% of AS2 and 66% of AS1221.

[Figure 13: see original paper]

For IPv6, we also compare fixed-stride storage overhead. Table 1 shows node counts: AS2 nodes decrease from 8K to 1K, while AS1221 nodes decrease from 3K to 0.7K as stride increases from 1 to 6. Variable strides yield similar benefits.

Figure 14 shows IPv6 storage overhead. For AS2 (Figure 14a), OET requires 38–125 Kb across strides 1–6, reducing storage by 74–94.9%, 53.6–91%, 32.9–59.9%, and 55.4–62.4% compared to the four baseline methods. For AS1221 (Figure 14b), OET requires 14–51 Kb, achieving reductions of 72.7–94.2%, 53.8–89.5%, 29.9–60.5%, and 55–62.5%, respectively.

[Figure 14: see original paper]

Overall, OET significantly reduces storage overhead for IPv6 sets, achieving average reductions of 73–95%, 53–90%, 31–60%, and 55–63% compared to original trie, leaf-pushed trie, Lulea trie, and Tree Bitmap Trie, respectively.

5. Conclusion

This paper proposes a storage-efficient IP address lookup algorithm based on offset addressing. The Offset Encoded Trie (OET) represents trie data structures compactly: each node maintains only one next-hop bitmap and one offset value, eliminating child and next-hop pointers. During lookup, each node uses its next-hop bitmap and offset to determine search success and compute the next node' s address. OET is essentially a compact equivalent of multi-way tries. In operation, the on-chip OET finds the longest matching prefix, which is used as a hash key to retrieve associated next-hop information from an off-chip prefix hash table. The off-chip table consists of multiple multi-choice hash tables, each storing prefixes of equal length. OET uses backup-based offline updates to ensure uninterrupted lookup.

Experimental evaluation using real IP prefix rule sets demonstrates that OET significantly reduces storage overhead compared to existing multi-way trie encoding methods. For example, OET reduces storage by 60–76% for IPv4 and 55–63% for IPv6 compared to Tree Bitmap Trie. As a storage-efficient data structure that fits entirely in on-chip memory, OET enables high-speed IP address lookup, meeting the scalability requirements of virtual and software routers.

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] M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker, "Dynamic route computation considered harmful," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 66-71, 2010.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

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

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

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

[26] 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.

[27] 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.

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

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

[30] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: an aid to network processing," in *ACM SIGCOMM*, 2005, pp. 181-192.

[31] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: deterministic and updatable hashing for high performance networking," in *IEEE INFOCOM*, 2008, pp. 101-105.

[32] R. Pagh and F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122-144, 2004.

[33] A. Kirsch and M. Mitzenmacher, "The power of one move: hashing schemes for hardware," in *IEEE INFOCOM*, 2008, pp. 106-110.

Author Biographies

Kun Huang: Postdoctoral Researcher, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences. huangkun09@ict.ac.cn

Gaogang Xie: Director and Professor, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences. Ph.D. advisor.

Yanbiao Li: Master's Student, College of Information Science and Engineering, Hunan University.

Xiangyang Liu: Assistant Professor, Department of Computer Science and Engineering, Michigan State University.

Footnotes:

1. Media Access Controller, media storage controller
2. Ternary Content-Addressable Memory
3. Network Intrusion Detection System/Network Intrusion Prevention System
4. Static Random Access Memory
5. Dynamic Random Access Memory

Note: Figure translations are in progress. See original paper for figures.