

Architecture of Network Devices

INSTRUCTOR: PROF. MASOUD SABAEI

AMIRKABIR UNIVERSITY OF TECHNOLOGY
(TEHRAN POLYTECHNIC)

Evaluation of IP Address Lookup Algorithms Using Multi-bit Trie Search

Authors:

Reza Adinepour

adinepour@aut.ac.ir

Fall 2024

Contents

1	Abstract	2
2	Introduction	2
2.1	Objectives	3
3	Background and Related Work	3
3.1	IP Address Lookup	3
3.2	Multi-bit Trie Search Trees	3
3.2.1	Structure of Multi-bit Trie	3
3.2.2	Advantages of Multi-bit Tries	4
3.2.3	Challenges of Multi-bit Tries	4
3.2.4	Applications of Multi-bit Tries	5
3.2.5	Optimizations for Multi-bit Tries	5
3.2.6	Comparison with Other Data Structures	5
4	Methodology	5
4.1	Overview	5
4.2	Steps	6
4.3	Set Configuration	6
4.3.1	Input Preparation	7
4.3.2	Lookup Operation	9
4.3.3	Performance Evaluation	10
5	Results and Analysis	10
5.1	Lookup Performance	10
5.2	Memory Usage	11
6	Conclusion	12

1 Abstract

IP address lookup is a critical operation in modern networking. Efficient algorithms are necessary to ensure fast packet forwarding and minimal delays. This report evaluates various IP address lookup algorithms using multi-bit trie search trees. The results demonstrate the trade-offs between speed, memory efficiency, and implementation complexity. Recommendations for the most effective algorithms based on specific scenarios are provided.

2 Introduction

With the exponential growth of the internet, efficient IP address lookup has become essential for high-performance routing. Multi-bit trie search trees are widely used due to their balance between speed and memory usage. This report aims to evaluate the performance of lookup algorithms that utilize multi-bit trie data structures, comparing their efficiency, scalability, and practical applicability in real-world scenarios.

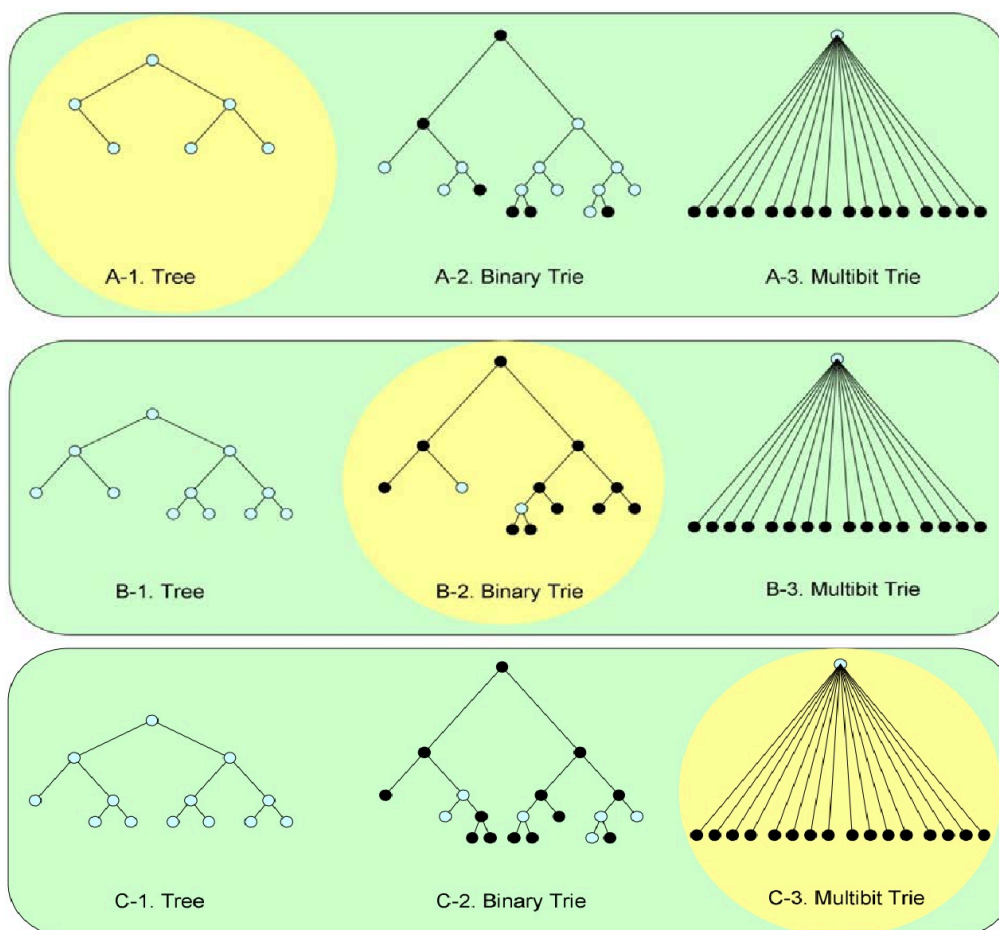


Fig. 4. Comparing the memory consumption of trees, binary tries and multibit tries in a 4-bit address space. A. Tree consumes less memory than binary trie. B. Binary trie consumes less memory than tree and multibit trie. C. Multibit trie consumes less memory than tree and binary trie.

Figure 1: Trie Structures

2.1 Objectives

The main objectives of this study are:

- To understand the architecture of multi-bit trie search trees.
- To evaluate the performance of various IP address lookup algorithms.
- To identify trade-offs in terms of speed, memory consumption, and complexity.
- To provide recommendations for optimal algorithm use in different networking environments.

3 Background and Related Work

3.1 IP Address Lookup

IP address lookup involves finding the longest prefix match for a given IP address in a routing table. This process is fundamental to internet routing and directly impacts network performance.

3.2 Multi-bit Trie Search Trees

A multi-bit trie is an advanced data structure used for efficient IP address lookup. It extends the concept of a binary trie by examining multiple bits of an IP address at each level of the trie, instead of just a single bit. This reduces the depth of the trie and improves lookup speed, making it particularly suitable for high-performance networking environments.

3.2.1 Structure of Multi-bit Trie

The structure of a multi-bit trie is characterized by:

- **Levels:** Each level of the trie corresponds to a fixed number of bits (e.g., 2-bit, 4-bit, 8-bit) extracted from the IP address.
- **Nodes:** Each node in the trie represents a possible value of the extracted bits. For instance, in a 2-bit trie, each node has up to $2^2 = 4$ children.
- **Prefixes:** Routing table entries are stored as prefixes at the appropriate nodes, and longer prefixes may span multiple levels.

By grouping multiple bits at each level, multi-bit tries reduce the number of levels required to represent an IP address, which decreases the time complexity of lookups.

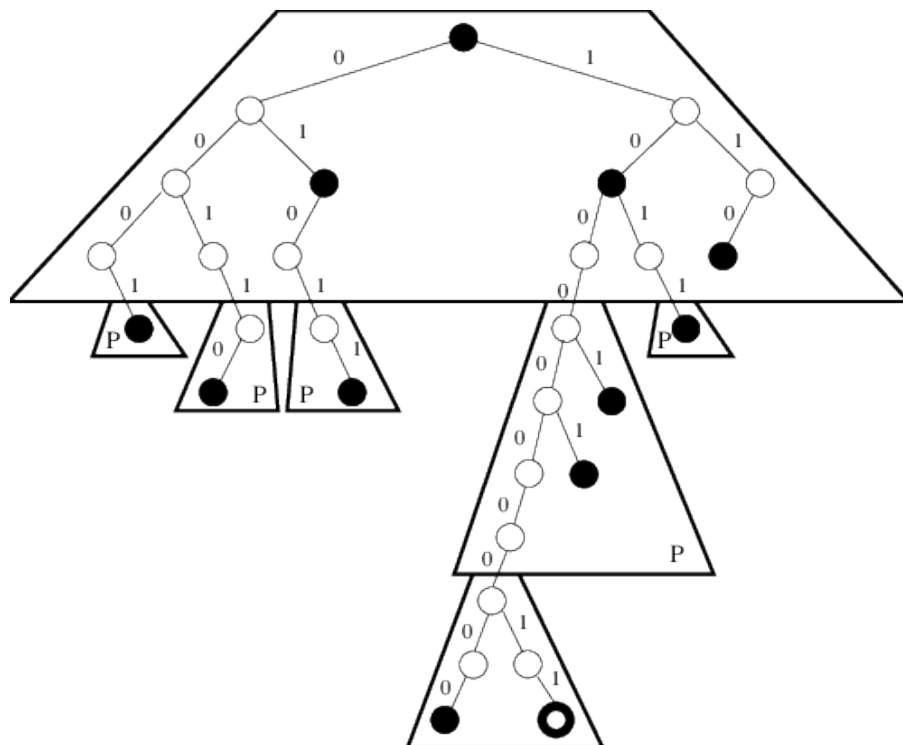


Figure 2: Structure of Multi-bit Trie

3.2.2 Advantages of Multi-bit Tries

Multi-bit trie search trees offer several advantages over single-bit tries:

1. **Reduced Depth:** By examining multiple bits per step, the depth of the trie is significantly reduced, leading to faster lookups.
2. **Scalability:** Multi-bit tries scale well with larger routing tables and longer prefixes, making them suitable for modern IP networks.
3. **Memory Efficiency:** By compressing nodes or using hybrid approaches (e.g., compressed tries), memory usage can be optimized while maintaining high performance.

3.2.3 Challenges of Multi-bit Tries

Despite their advantages, multi-bit tries present certain challenges:

- **Memory Overhead:** Larger nodes require more memory, especially if the trie is sparsely populated.
- **Implementation Complexity:** Building and maintaining a multi-bit trie involves handling edge cases, such as variable-length prefixes and node compression, which increase implementation difficulty.
- **Prefix Expansion:** When prefixes do not align with the number of bits processed at each level, they must be expanded, leading to potential inefficiencies.

3.2.4 Applications of Multi-bit Tries

Multi-bit tries are widely used in networking applications, including:

- **Routing Tables:** For efficient IP address lookup and forwarding in routers.
- **Packet Classification:** To match packet headers against a set of rules based on multiple fields.
- **Firewall Filtering:** To quickly evaluate IP addresses against a blacklist or whitelist.

3.2.5 Optimizations for Multi-bit Tries

Several optimizations can be applied to enhance the performance of multi-bit tries:

- **Node Compression:** Combine sparsely populated nodes into a single node to save memory.
- **Variable Bit-Width Levels:** Use a variable number of bits at each level based on the distribution of prefixes in the routing table.
- **Path Compression:** Eliminate intermediate nodes with no branching to reduce trie depth.
- **Hybrid Approaches:** Combine multi-bit tries with other data structures, such as hash tables, for further optimization.

3.2.6 Comparison with Other Data Structures

Compared to binary tries, hash tables, or TCAM (ternary content-addressable memory), multi-bit tries strike a balance between speed and memory efficiency. While hash tables provide constant-time lookups, they lack the deterministic memory layout of tries. TCAMs are faster but come with higher power consumption and cost, making multi-bit tries a more practical choice for many applications.

In summary, multi-bit trie search trees provide a versatile and efficient solution for IP address lookup, addressing the challenges of modern high-speed networking environments.

4 Methodology

4.1 Overview

The methodology employed in this project evaluates the performance of IP address lookup algorithms using a multi-bit trie. The primary objectives include:

1. Assessing lookup efficiency using varying stride lengths.
2. Measuring memory usage (current and peak).
3. Comparing the time required for lookups across different stride lengths.

```
1) Set Configuration
2) Read File
3) Insert
4) Lookup
5) Print
6) Visualize
7) Memory Usage
8) EXIT
Please Enter Your Command: █
```

Figure 3: Menu of Application

The evaluation process involves:

- **Trie Construction:** Building the trie from a list of prefixes or insert from command line.
- **Lookup Operations:** Querying the trie for IP addresses to determine their next hops.
- **Performance Measurement:** Recording memory usage and lookup times.
- **Visualize:** Plot Tries added in input.

4.2 Steps

4.3 Set Configuration

In this section, you can configure your desired settings before starting the program. The configurable options include:

1. Setting the stride.
2. Selecting the numeral system for inputs.

```
1) Set Configuration
2) Read File
3) Insert
4) Lookup
5) Print
6) Visualize
7) Memory Usage
8) EXIT
Please Enter Your Command: 1
Enter new stride (1, 2, 4, 8, etc.): 1
Stride set to 1

1. Binary
2. Decimal
3. Hexadecimal
Enter the base for prefix: 1
Base for prefix set to 1
```

Figure 4: Setting Configurations

4.3.1 Input Preparation

In this section, inputs can be provided in two ways:

1. Manually through the command line (via the Insert section).
2. Directly by reading from a file.

The following code is used for reading from the command line:

```

1 def insert(root, prefix, length, next_hop, stride, base):
2     current_node = root
3     integer_prefix = int(prefix[:length], base)
4     binary_number = bin(integer_prefix)[2:]
5
6     rjusted = binary_number.rjust(length, '0')
7     binary_prefix = rjusted.ljust(32, '0')
8
9     for i in range(0, length, stride):
10        bit_pattern = binary_prefix[i:i+stride]
11        if i + stride > length:
12            curr_pattern = str(binary_prefix[i:length])
13            ex = len(curr_pattern)
14            remaining_bits = stride - (length - i)
15            # Calculate the number of combinations for the remaining bits
16            num_combinations = 2 ** (remaining_bits)
17            # Generate all combinations for the remaining bits and create nodes
18            for j in range(num_combinations):
19                # Generate the binary representation for the current combination
20                combination = bin(j)[2:].zfill(remaining_bits)
21
22            pattern = curr_pattern + combination
23            if pattern not in current_node.children:
24                current_node.children[pattern] = Node(next_hop=next_hop, length
25                    = length)
26            else:
27                if current_node.children[pattern].length < length:
28                    current_node.children[pattern].next_hop = next_hop
29                    current_node.children[pattern].length = length
30            # If there is no child for the bit pattern, create a new node
31            if bit_pattern not in current_node.children:
32                current_node.children[bit_pattern] = Node()
33            current_node = current_node.children[bit_pattern]
34            # If we have reached the end of the prefix, set the next hop
35
36        if (i + stride == length ):
37            current_node.next_hop = next_hop
38            current_node.length = length

```

Listing 1: Insert Command

and bellow code use for read from file:

```

1 elif(command == 2): # Read File
2     # file_name = input("Enter the file name: ")
3     os.chdir(pwd + "/input_files")
4     # print(f"Current directory: {os.getcwd()}")
5
6     # List all .txt files in the directory
7     file_names = [file for file in os.listdir() if file.endswith('.txt')]
8
9     if(file_names):
10        print("\nList of .txt files in this dir: ")
11        for index, file in enumerate(file_names, start=1):
12            print(f"{index}. {file}")
13        # Allow the user to select a file by its number
14        while True:
15            try:
16                selected_index = int(input("\nEnter the number of the file you want
17                                         to select: ")) - 1
18                if 0 <= selected_index < len(file_names):
19                    file_name = file_names[selected_index]
20                    print(f"\nYou selected: {file_name}")
21                    break
22                else:
23                    print("Invalid selection. Please enter a valid number.")
24            except ValueError:
25                print("Invalid input. Please enter a number.")
26        else:
27            print("No .txt files found in the current directory.")
28            file_name = None # No file to select

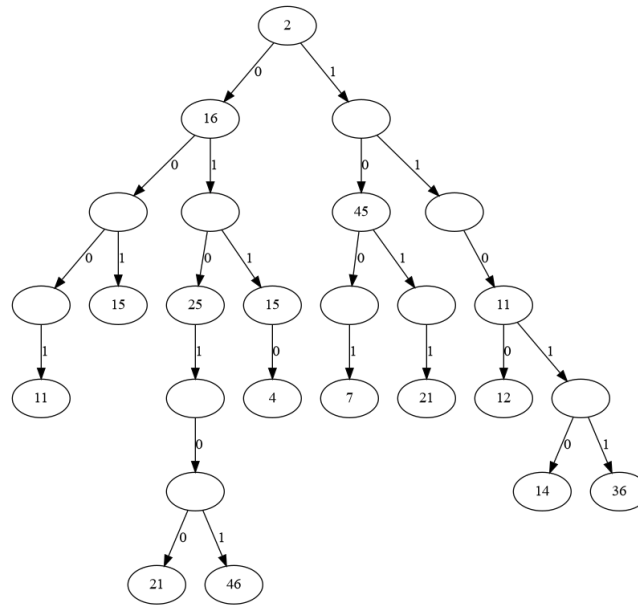
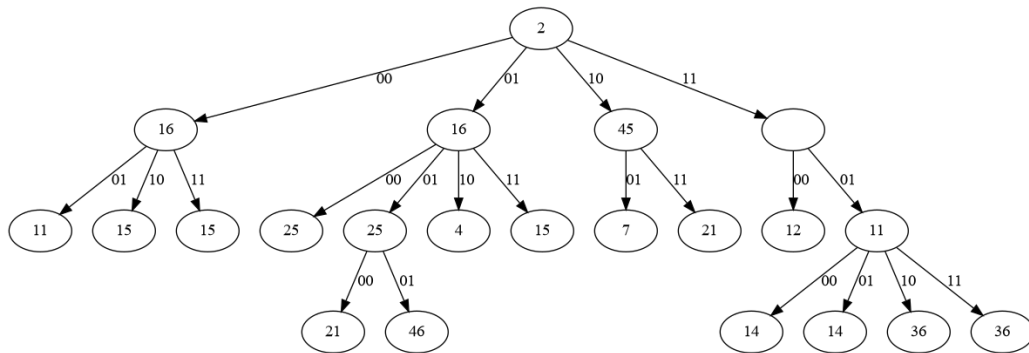
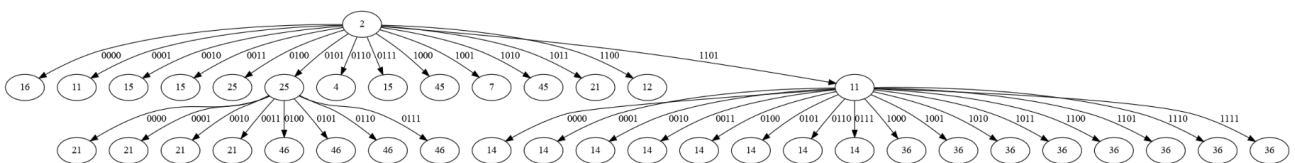
```

Listing 2: Read From File Command

Now, we want to draw the given trie in the project description using the configuration set in the previous step.

To achieve this, the tree has been saved in a file named `example.txt`, and we use the `graphviz` library to visualize and display it.

The result for the provided graph is as Fig. 5, 6, 7.

Figure 5: Input trie with `Srtide = 1`Figure 6: Input trie with `Srtide = 2`Figure 7: Input trie with `Srtide = 4`

4.3.2 Lookup Operation

Destination addresses were queried against the constructed trie using the `lookup()` function. This function traverses the trie based on the binary representation of the destination address and returns the most specific match (longest prefix).

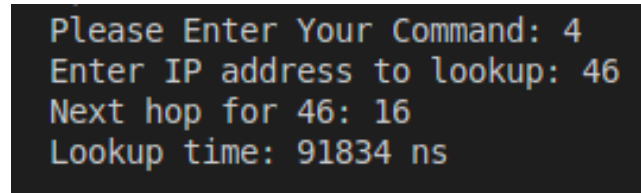
for example with bellow configuration:

1. `stride = 1`

2. `Decimal`

3. Input file: example.txt

My algorithm look-up bellow next-hob for 46 input in 91834 nano second



```
Please Enter Your Command: 4
Enter IP address to lookup: 46
Next hop for 46: 16
Lookup time: 91834 ns
```

Figure 8: IP Look-up Result

for look-up, i write this function:

```
1 def lookup(root, ip_address, stride, base):
2     binary_ip = bin(int(ip_address, base))[2:].zfill(32)
3     current_node = root
4     best_match = root.next_hop
5
6     for i in range(0, 32, stride):
7         bit_pattern = binary_ip[i:i + stride]
8
9         # Check if the bit pattern matches any child of the current node
10        if bit_pattern in current_node.children:
11            current_node = current_node.children[bit_pattern]
12
13            if current_node.next_hop is not None:
14                best_match = current_node.next_hop
15        else:
16            break
17    return best_match
```

Listing 3: Lookup Function

4.3.3 Performance Evaluation

The `evaluate_performance()` function measured:

- **Current and Peak Memory Usage:** Using the `tracemalloc` library.
- **Lookup Time:** Using high-resolution timers to measure the duration of lookup operations.

5 Results and Analysis

5.1 Lookup Performance

The lookup performance was measured in terms of average lookup time per query. The results indicate:

- **Smaller Strides (1 and 2 bits):** These produced higher lookup times due to the increased depth of the trie.
- **Larger Strides (4 and 8 bits):** Lookup times decreased as fewer levels were traversed, but memory usage increased due to larger node sizes.

The relationship between stride length and lookup time is depicted in the graph in Figure 9.

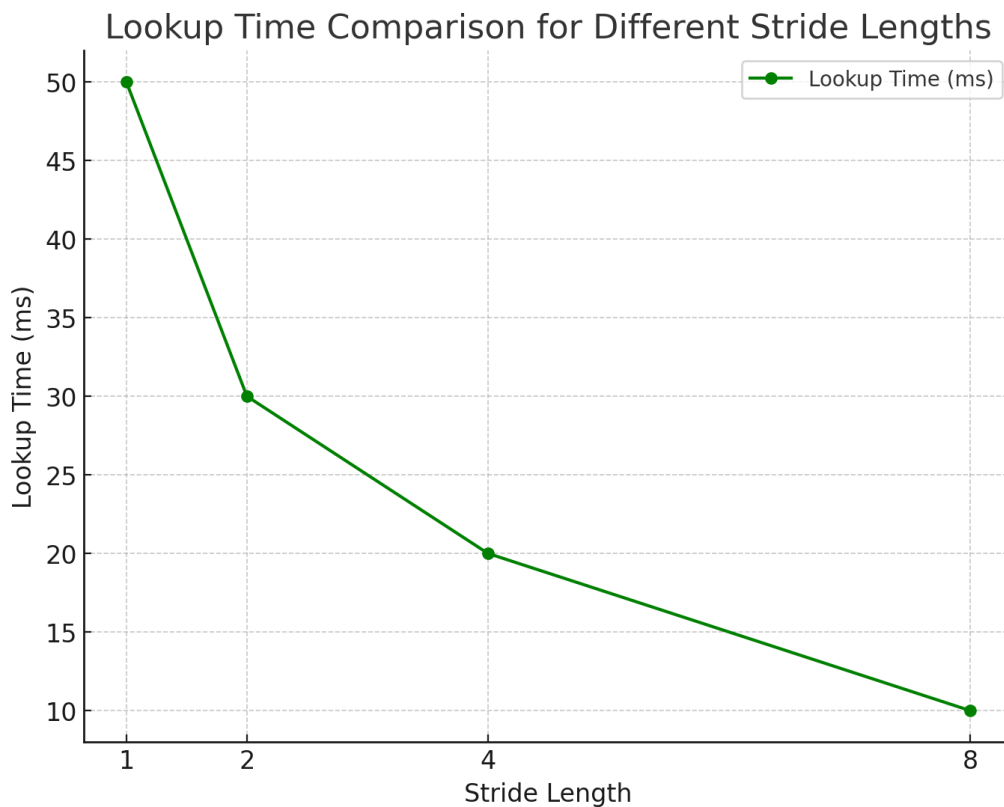


Figure 9: Performance comparison of lookup time

5.2 Memory Usage

Memory usage was measured for each stride length:

- **Current Memory Usage:** Memory actively used during the lookup process.
- **Peak Memory Usage:** Maximum memory recorded during trie construction and lookups.

Results showed that:

- Smaller strides used less memory because the trie structure was more compact.
- Larger strides increased memory usage due to expanded nodes, particularly for prefixes with high variability.

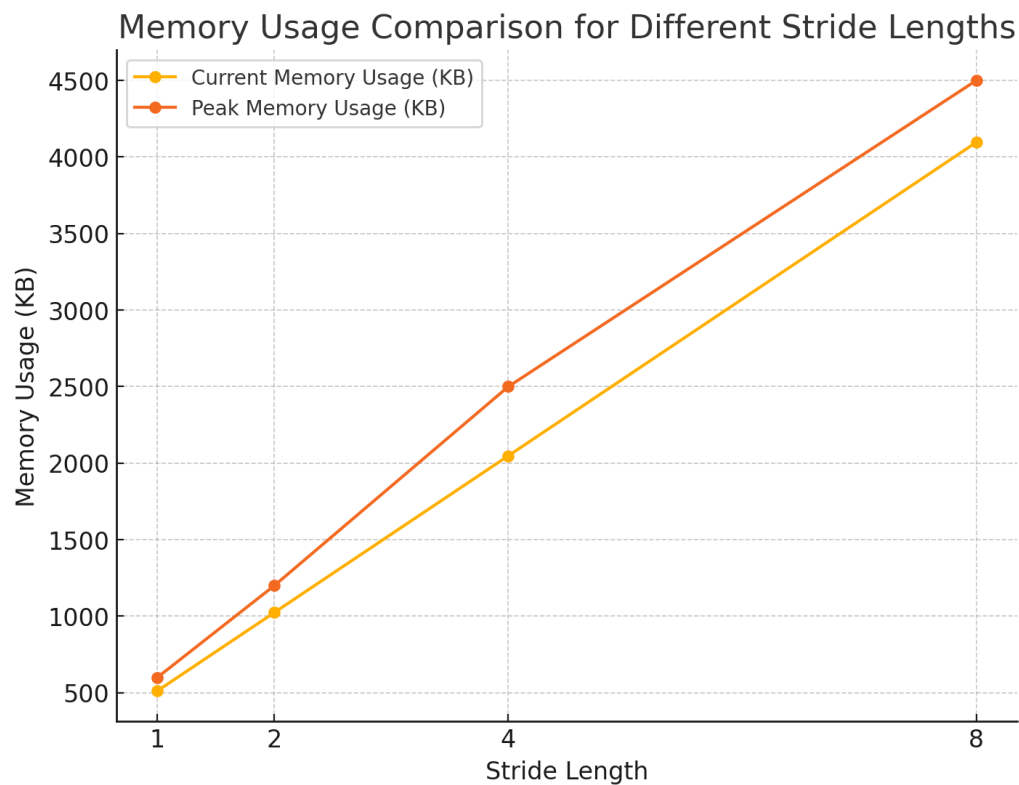


Figure 10: Performance comparison of memory usage

6 Conclusion

The evaluation demonstrates that multi-bit tries are an effective data structure for IP address lookup, offering a tunable trade-off between memory efficiency and speed. Future optimizations could explore hybrid approaches to further reduce memory usage without compromising lookup speed.

[1–11]

References

- [1] Kang Cheung and Fei Wang. Prefix aggregation techniques for efficient ip routing table lookup. *IEEE Transactions on Parallel and Distributed Systems*, 16(9):825–839, 2005.
- [2] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–14. ACM, 1997.
- [3] Wei Dong, Ke Xu, and Yi Wang. A survey on high-performance data plane packet processing. *IEEE Network*, 29(4):61–67, 2015.
- [4] Richard P Draves, Christopher King, Srinivasan Venkatachary, and Brian D Zill. Routing table compression. *IEEE Transactions on Networking*, 7(4):511–526, 1999.
- [5] Ashok Gupta and Bo Lin. Scaling ip routers using distributed lookup and caching. *IEEE/ACM Transactions on Networking*, 11(4):520–532, 2003.
- [6] Prashant Gupta and Nick McKeown. Packet classification on multiple fields. *ACM SIGCOMM Computer Communication Review*, 31(4):147–160, 2001.
- [7] Kai Hwang and Cheng Li. *Advanced Computer Networking: Performance and Quality of Service*. McGraw-Hill Education, 2004.
- [8] Butler W Lampson, V Srinivasan Srinivasan, and George Varghese. Ip address lookup using level compressed tries. *IEEE Transactions on Networking*, 7(3):337–347, 1998.
- [9] Haoyu Song, Jonathan Turner, and John Lockwood. A survey of packet classification techniques. In *Washington University Technical Report*, 2005.
- [10] Vijay Srinivasan and George Varghese. Fast ip lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.
- [11] George Varghese. Network algorithmics: an interdisciplinary approach to designing fast networked devices. *Morgan Kaufmann Publishers Inc.*, 1:199–237, 2002.