Memory-Efficient IPv4/v6 Lookup on FPGAs Using Distance-Bounded Path Compression

Hoang Le
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Los Angeles, CA 90089, USA
Email: hoangle@usc.edu

Weirong Jiang
Juniper Networks Inc.
Sunnyvale, CA, USA
Email: weirongj@acm.org

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Los Angeles, CA 90089, USA
Email: prasanna@usc.edu

***Abstract*—Memory efficiency with compact data structures for Internet Protocol (IP) lookup has recently regained much interest in the research community. In this paper, we revisit the classic trie-based approach for solving the longest prefix matching (LPM) problem used in IP lookup. In particular, we target our solutions for a class of large and sparsely-distributed routing tables, such as those potentially arising in the next-generation IPv6 routing protocol. Due to longer prefix lengths and much larger address space, straight-forward implementation of trie-based LPM can significantly increase the number of nodes and/or memory required for IP lookup. Additionally, due to the available on-chip memory and the number of I/O pins of Field Programmable Gate Arrays (FPGAs), state-of-the-art designs cannot support large IPv6 routing tables consisting of over 300K prefixes.**

We propose two algorithms to compress the uni-bit-trie representation of a given routing table: (1) *single-prefix distance-bounded path compression* and (2) *multiple-prefix distance-bounded path compression*. These algorithms determine the optimal maximum *skip distance* at each node of the trie to minimize the total memory requirement. Our algorithms demonstrates substantial reduction in the memory footprint compared with the uni-bit-trie algorithm ($1.86\times$ for IPv4 and $6.16\times$ for IPv6), and with the original path compression algorithm ($1.77\times$ for IPv4 and $1.53\times$ for IPv6). Furthermore, implementation on a state-of-the-art FPGA device shows that our algorithms achieve 466 million lookups per second and are well suited for 100Gbps lookup. This implementation also scales to support larger routing tables and longer prefix length when we go from IPv4 to IPv6.

I. Introduction

State-of-the-art FPGAs offer high operating frequency, unprecedented logic density and a host of other features. Additionally, FPGAs are programmed specifically for the problem to be solved rather than to solve all problems, they can achieve higher performance than the general-purpose processors. The advantage is more beneficial for applications with a regular structure and abundant parallelism. Thus, FPGA is a promising implementation technology for many areas [3], [8], [10], [12].

With the rapid growth of the Internet, IP-lookup becomes the bottle-neck in network traffic management. Therefore, the design of high speed IP routers has been a major area of research. High link rates demand that packet forwarding in IP routers must be performed in hardware [17]. For instance, a 100 Gbps link requires a throughput of over 150 million lookups per second (MLPS). Such line rates demand fast lookup algorithms with *compact data structures*.

Supported by the U.S. National Science Foundation under grant No. CCF-1018801. Equipment grant from Xilinx is gratefully acknowledged.

At the core routers, the size of the routing table also increases at the rate of 25-50K prefixes per year [7]. Additionally, IPv6 extends the length of the IP address from 32 bits in IPv4 to 128 bits. This increased length allows for a broader range of addressing hierarchies and a much larger number of addressable nodes. According to the Internet Architecture Board (IAB), an IPv6 address consists of two parts: a 64-bit network/sub-network ID followed by a 64-bit host ID. At the core router, only the 64-bit network/sub-network ID is relevant in making the forwarding decisions. The increase in the size of routing table and the extension of the prefix length necessitate high *memory-efficient* lookup algorithms to reduce the size of the storage memory. Moreover, the use of FPGAs in network router requires compact memory footprints that can fit in the limited on-chip memory to ensure high throughput.

Our analysis of the synthetic IPv6 tables generated using [23] suggests that current solutions for IP lookups do not actually scale very well to support IPv6. The most popular IP lookup algorithms are trie-based. However, their performance decreases linearly as the tree depth increases. Additionally, the explosion in the number of trie nodes when doubling the prefix length makes the straight-forward trie-based solutions less attractive. Existing compression techniques ([16], [19]) can *increase the total memory requirement* and the computational complexity at the nodes, while *reducing the total number of nodes*. Some of these solutions can reduce the memory footprint with the requirement of back-tracking, which is not desirable from hardware implementation standpoint.

Therefore, the focus of this paper is on achieving significant reduction in memory requirements for the longest prefix-match operation needed in IPv4/v6 lookups. Fast and efficient IP lookup has been well-studied in research community, and numerous algorithms have been proposed (more details in Section II-B). However, IPv6 lookup has not been well studied. There are only few FPGA designs for large-scale IPv6 lookup [20], [21], [25]. Some characteristics (e.g sparsity) of IPv6 routing tables have not been exploited. Thus, it is still possible to achieve substantial reduction in memory usage, especially in IPv6.

TABLE I
A SAMPLE ROUTING TABLE (MAXIMUM PREFIX LENGTH = 8)

	Prefix	Next Hop		Prefix	Next Hop
P_0	*	0	P_5	11111*	5
P_1	000*	1	P_6	010*	6
P_2	01000*	2	P_7	1*	7
P_3	01011*	3	P_8	0*	8
P_4	11010*	4			

We propose the *distance-bounded path compression* algorithms for trie-based IP lookup. These algorithms are realized on a *scalable high-throughput, SRAM-based* linear pipeline architecture. This design utilizes the dual-ported feature of on-chip memory on FPGAs to achieve high throughput. Moreover, external SRAMs can be used to overcome the limitation in the amount of the on-chip memory and to carry larger routing tables.

This paper makes the following contributions:

1) *Single-prefix distance-bounded path compression* algorithm that can compress the trie of a given routing table to significantly reduce the total memory requirement, while keeping the data structure simple (Section III-B).
2) *Multiple-prefix distance-bounded path compression* algorithm that generalizes the previous algorithm to balance the total number of nodes and the total memory requirement (Section III-C).
3) A design that support over $330K$ IPv4/v6 prefixes and sustains a high throughput of 466 million lookups per second on a state-of-the-art FPGA device (Section V).

The rest of the paper is organized as follows. Section II covers the background and related work. Section III introduces the proposed distance-bounded path compression algorithms. Section IV describes the architecture and its implementation on FPGA. Section V presents implementation results. Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

A. IP Lookup

A sample routing table with the maximum prefix length of 8 is illustrated in Table I. This sample table will be used throughout the paper. Note that the next hop values need not be in any particular order. In this routing table, binary prefix P_3 (01011$*$) matches all destination addresses that begin with 01011. Similarly, prefix P_4 matches all destination addresses that begin with 11010. The 8-bit destination address **IP**$=$ 01000000 is matched by the prefixes P_0, P_2, P_6, and P_8. Since $|P_0| = 0$, $|P_2| = 5$, $|P_6| = 3$, $|P_8| = 1$, P_2 is the longest prefix that matches **IP** ($|P|$ is defined as the length of prefix P). In longest-prefix routing, the next hop index for a packet is given by the table entry corresponding to the longest prefix that matches its destination IP address. Thus, in the example above, the next hop of 2 is returned.

B. Related Work

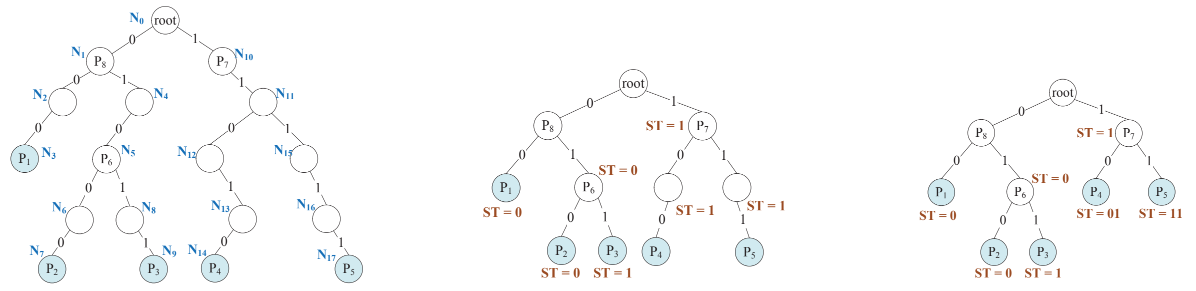
In general, algorithms for IP-lookup can be classified into the following categories: trie-based, scalar-tree-based, range-tree-based, and hash-based approaches. Each has its own advantages and disadvantages.

In *trie-based approaches* ([4], [5], [13], [21], [22]), IP-lookup is performed by simply traversing the trie according to the bits in the IP address. These designs have a *compact data structure*, but *large* number of trie nodes; hence, moderate memory efficiency. Furthermore, the latency of these trie-based solutions is proportional to the prefix length, making them less attractive for IPv6 protocol, unless multi-bit trie is used, as in [21]. This architecture employs a method called *shape graph* to achieve good memory efficiency and high throughput. However, it requires more complex preprocessing. The optimization also makes it difficult to update. Additionally, the use of hash table for next-hop lookup demands a highly efficient hash function to avoid collisions, which can dramatically reduce the overall performance. The uni-bit trie can be compressed as in [2], [16], [19] to reduce the number of nodes. Yet, these compression techniques can greatly increase the total memory requirement and the computational complexity at the nodes. Some of them also require back-tracking, which is not suitable for hardware implementation.

In *range-tree-based approaches* ([15], [24]), each prefix is treated as a *range*. In [15], a B-tree data structure is used to support both prefixes and non-intersecting ranges as prefixes are converted to ranges. The number of ranges is twice the number of prefixes in the worst case. The main advantage of this approach is that each prefix is stored in $O(1)$ B-tree nodes per B-tree level; hence, updates can be performed in $O(\log_m n)$, where m is the order of the B-tree and n is number of prefixes in the routing table. In [24], IP lookup is performed using a multi-way range tree (a type of B-tree) algorithm, which achieves worst-case search and update time of $O(\log n)$. The proposed approach achieves the optimal lookup time of binary search, and can be updated in logarithmic time. Another advantage of these tree-based approaches is that the lookup latency does not depend on the prefix length, but is proportional to $\log$ of the number of prefixes; thus, they are highly suitable for IPv6. Note that B-tree of order greater than 3 is very difficult to be efficiently implemented on hardware due to (1) the poor uniformity in the data structure and (2) the wide-memory access required for each fat node.

The *scalar-tree-based approaches* were also proposed ([6], [14]). In [14], each prefix is treated as a search key and the entire routing table is stored into multiple binary search trees. This approach achieved low memory footprint at the cost of high memory bandwidth. In [6], the authors presented the “Scalar Prefix Search”, which interprets each prefix as a number without any encoding. Using keys and match vectors, it can store several prefixes in one key. This approach reduced the number of prefixes while doubling the memory bandwidth. The memory requirement also increases for routing tables with high percentage of distinct prefixes.

In *hash-based approaches* ([9], [18], [21]), hashing functions or bloom filters are used. The architecture in [9] takes advantage of the benefit of both a traditional hashing scheme and reconfigurable hardware. However, these architectures



(a) A sample trie (b) $skip\text{-}distance = 1$ (c) $skip\text{-}distance = 2$

Fig. 1. A sample trie and its compressed tries for different $skip\text{-}distances$ (ST = *skip-string*)

only result in moderate memory efficiency, and can only be updated using partial reconfiguration when adding or removing prefixes. It is also unclear as how to scale these designs to support larger routing tables and/or longer prefix lengths. Additionally, they suffer from non-deterministic performance due to hash collisions.

III. Algorithms

A. Definitions and Notations

Fig. 1(a) shows the corresponding trie representation of Table I. Given a routing table and the corresponding binary trie representation, the following terms are defined:

Definition Any node, for which the path from the root of the trie to that node represents a prefix in the routing table, is called a ***prefix-node*** (e.g. N_1, N_3).

Definition A path connecting two nodes of a trie is called a ***non-branching path*** if all the nodes along the path (except the end node) have exactly **one** child-node (e.g. $N_4\_N_5$, $N_{12}\_N_{13}\_N_{14}$).

Definition The ***skip-string*** of each node is defined as the non-branching path of its **single** child-node, if any. If the node has 2 children, its skip-string is **empty** (ϕ). The ***skip distance*** is defined as the length of the skip-string.

Definition Nodes with an empty skip-string are called ***single-node***. Otherwise, they are called ***super-node***.

Definition The memory footprint is defined as the size of the memory required to store the entire routing table. The terms ***storage***, ***memory requirement***, ***memory footprint***, and ***storage memory*** are used interchangeably in this paper.

B. Algorithm 1: Single-Prefix Distance-Bounded Path Compression (SP-DBPC)

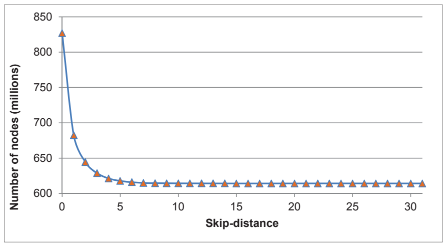
Trie search algorithm is a good choice for IP forwarding engine due to its simple search at each node. However, in a sparse trie such as one found in IPv6, the number of nodes in the trie drastically expands as the prefix length increases from 32 bits to 64 bits. Our analysis for real-life and synthetic routing tables (more detail in Section V) shows that the number of nodes of a core IPv6 table is almost $10\times$ more than that of an IPv4 table, each with the same number of prefixes. Therefore, a straight-forward implementation of trie-based approach for an IPv6 routing table is not desirable due to high memory requirement. Path compression techniques work well to reduce the number of trie nodes ([2], [16], [19]). Yet, reducing the total number of nodes does not necessarily lead to the reduction of the memory footprint. The ultimate goal is to reduce the total required storage. In these existing path compression solutions, the total memory requirement is not minimal due to the large size of each node in the compressed trie.

In this paper, the focus is on the classic trie-based approach for IP lookup. We observe that the length of a non-branching path in a trie varies from 1 to the maximum depth of the trie. Hence, using maximum length for the skip-string in the path compression algorithm leads to a larger wastage in the memory footprint. The examples in Fig. 1 are used to illustrate the point. In Fig. 1(b), with the maximum skip-distance of 1, the total number of nodes is 11. Whereas in Fig. 1(c), the total number of nodes is 9 with the maximum skip-distance of 2. In hardware implementation, all the nodes have a uniform size. Therefore, the memory wastage comes from the single-nodes and super-nodes whose skip-distance is shorter than the maximum distance. In the above examples, the memory wastage is 4 bits in the first case and 9 bits in the second case.

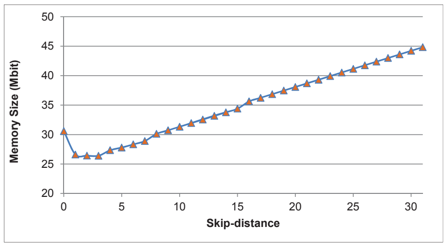
Long skip-distance increases not only the memory consumption, but also the comparison complexity at each node. Therefore, a memory-efficient implementation of path-compressed trie on hardware requires: (1) the skip-distance to be bounded and (2) the *optimal* maximum skip-distance D to be determined to minimize the memory requirement.

We propose the *single-prefix distance-bounded path compression (SP-DBPC)* algorithm to compress the trie (Algorithm 1). In this algorithm, the number of prefixes per super-node is fixed at 1. The algorithm works as follows. A unit-bit trie is built from a given routing table. At each node of the trie (starting from the root), *SP-DBPC* finds the non-branching path P and calculates the skip-distance d for the current node. Let m denote the number of nodes following the current node on path P that can be merged with the current node. m can be calculated as $\min(d, D)$. The skip-string of the current node is updated. The child-nodes of the last merged node become the child-nodes of the current node. The *SP-DBPC* algorithm is then executed recursively until the leaf-node is reached. Once the algorithm completes, the total number of nodes is calculated and returned.

In hardware implementation, each node of the *SP-DBPC* trie has the following data fields: (1) 2 child pointers, (2) skip-

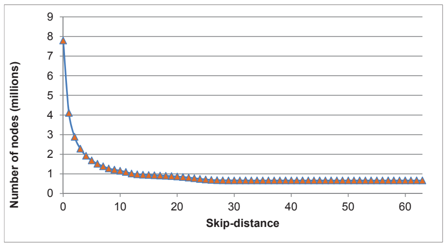


(a) Number of nodes vs. skip-distance

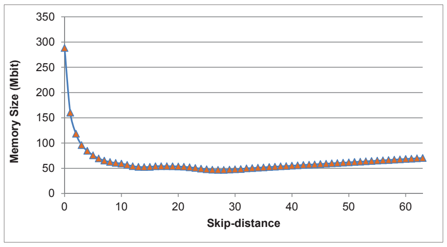


(b) Memory requirement vs. skip-distance

Fig. 2. Number of nodes and memory requirement of the compressed IPv4 trie for different skip-distances



(a) Number of nodes vs. skip-distance



(b) Memory requirement vs. skip-distance

Fig. 3. Number of nodes and memory requirement of the compressed IPv6 trie for different skip-distances

string, (3) length of actual skip-string, (4) position of prefix bit, and (5) next hop information. The following notations are used in the analysis:

- N: the total number of nodes.
- A: the size of one child pointer in each node
- D: the maximum skip-distance.
- H: the size of the next hop information.
- M: the total memory requirement.
- L_P: the maximum prefix length (32 for core IPv4 and 64 for core IPv6 routing tables).

Algorithm 1 SP-DBPC

```
Input: Current trie node $n$, maximum skip-distance $D$
Output: Compressed trie
 1: $m = \#$ of trie nodes to be merged, $d =$ skip-distance
 2: Find the non-branching path $P$ of the current node, calculate $d$
 3: if $d \leq D$ then
 4:     $m = d$
 5: else
 6:     $m = D$
 7: end if
 8: Merge $m$ nodes of $P$ to the current node
 9: Update the skip-string of the current node
10: Update children of the super-node
11: SP-DBPC ($n \rightarrow left\_child, D$)
12: SP-DBPC ($n \rightarrow right\_child, D$)
```

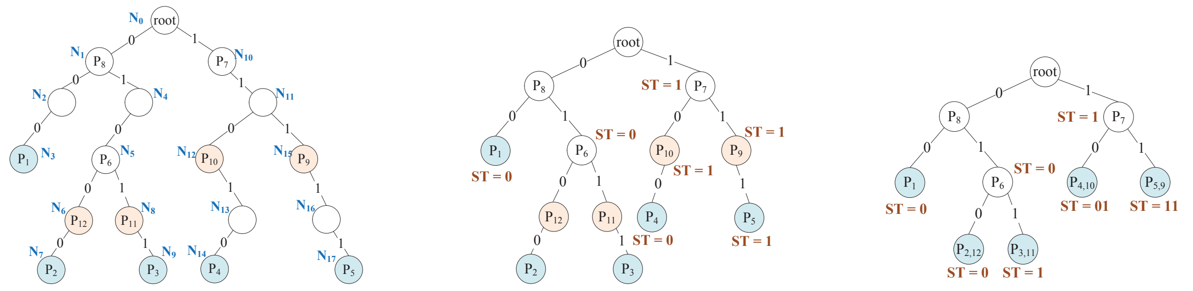
The total memory requirement can be calculated as $M_{SP} = N_{SP} \times (2A + D + 2\lceil \log D \rceil + H)$. In the formula, D varies from 0 (no compression) to $L_P - 1$ (maximum compression). Hence, the *optimal* skip-distance D_{opt} is determined for all the possible values of D such that M_{SP} is minimum. That is $M_{SP} = \min_{D=0}^{L_P-1}\{M_{SP}\}$.

An additional constraint can be applied to further simplify the hardware implementation. Note that in the above algorithm, the prefix bit can be at any position in the skip-string. Therefore, a data field (#4) is required to keep track of this bit. If we restrict the prefix bit to only be at the end of the skip-string, then each node needs not to store the position of the prefix bit. This restriction also simplifies the hardware implementation. The total memory footprint is reduced to Equ. 1.

$$M_{SP} = N_{SP} \times (2A + D + \lceil \log D \rceil + H) \qquad (1)$$

A real-life core IPv4 routing table collected from Project - RIS [1] on $06/03/2010$ is used to verify the result. The *SP-DBPC* is applied for each value of the skip-distance in $[0, 31]$ (0 is no compression and 1 is maximum compression). The number of nodes and the size of memory footprint are recorded and plotted in Fig. 2 for each case. Fig. 2(a) shows that the number of nodes of the compressed trie reduces as the skip-distance increases. However, the memory consumption reaches its minimum when $skip\text{-}distance = 3$ and then increases. Hence, the optimal skip-distance for this routing table is 3. The result confirms that reducing the number of nodes does not necessarily lead to the reduction of the memory footprint.

Although the *SP-DBPC* algorithm works fairly well with IPv4 routing tables, its performance on IPv6 table needs to be verified. Since the largest publicly available IPv6 routing tables consist of less than 3K prefixes, it is difficult to obtain reliable statistics for these tables. Thus, a synthetic IPv6 routing table generated by the same method as in [23] is used. The generator takes a real-life IPv4 table as an input and produces a comparable IPv6 table that would better predict a likely outcome of the production table than using simple random prefixes. Similarly to the IPv4 case, The *SP-DBPC* is applied for each value of the skip-distance in $[0, 63]$ (0 is no compression and 63 is maximum compression). The number of nodes and the size of memory footprint are recorded and plotted in Fig. 3 for each case. The analogous result is also observed for the IPv6 table. The number of nodes of the compressed trie reduces as the skip-distance increases. The memory consumption also



(a) A sample trie (b) Maximum number of prefixes $= 1$ (c) Maximum number of prefixes $= 2$

Fig. 4. A sample trie and its compressed tries for different maximum number of prefixes per node with $skip\text{-}distance = 2$ (ST = *skip-string*)

reaches its minimum when $skip\text{-}distance = 27$ and then increases. Therefore, the optimal skip-distance for this routing table is 27. Similar experiments with different real-life and synthetic routing tables show consistent results, which agree with the above analysis. These results are presented in detail in Section V.

Search in a SP-DBPC trie: After building the compressed trie, IP-lookup operation can be performed as follows. The destination IP address is extracted from the incoming packet. At each node of the compressed trie, there are 3 steps to be executed:

1) The skip-string and its skip-distance d are extracted. If $d = 0$, skip to Step 3.
2) The skip-string is compared with the next d bits of the IP address. If there is no match and the current node is not a prefix-node, then the search is terminated. Otherwise, the next hop information is updated and the search is terminated.
3) If the current node is a prefix-node, then the next hop information is updated and the IP address is left-shifted by $(d+1)$ positions. If a leaf-node is reached, then the search is terminated; otherwise, go back to Step 1.

Complexity: The first step of the algorithm is to build a trie from a given routing table consisting of N_P prefixes. This step has a complexity of $O(N_P)$. The trie is then traversed to merge nodes on the same non-branching path up to a maximum skip-distance. Since every node of the trie needs to be processed, this step has a complexity of $O(N_P \times L_P)$. Because the two steps are sequential, the overall complexity of the single-prefix distance-bounded path compression algorithm is $O(N_P \times L_P)$.

C. Algorithm 2: Multiple-Prefix Distance-Bounded Path Compression (MP-DBPC)

Consider the same sample routing table in Table I with 4 additional prefixes P_{9-12}, as shown in Fig. 4(a). The new trie is denser than the original trie in Fig. 1(a). In this modified sample trie, if each node can store only one prefix, then the compressed trie is not optimal. Fig. 4(b) and 4(c) illustrate the compressed tries with $skip\text{-}distance = 2$ for the maximum number of prefixes per node of 1 and 2, respectively. It is clear from the example that, in some tries, the maximum number of prefixes per super-node should be determined for optimal memory efficiency.

Algorithm 2 MP-DBPC

```
Input: Current trie node $n$, maximum skip-distance $D$, maximum number of prefixes $n_p$
Output: Compressed trie
 1: $m$ = # of trie nodes to be merged, $d$ = skip-distance
 2: Find the non-branching path $P$ of the current node that has at most $n_p$ prefixes, calculate $d$
 3: if $d \leq D$ then
 4:     $m = d$
 5: else
 6:     $m = D$
 7: end if
 8: Merge $m$ nodes of $P$ to the current node
 9: Update the skip-string of the current node
10: Update children of the super-node
11: MP-DBPC ($n \rightarrow left\_child, D, P$)
12: MP-DBPC ($n \rightarrow right\_child, D, P$)
```

The *SP-DBPC* algorithm can be extended to include more than one prefix per super-node. The multiple-prefix distance-bounded path compression (MP-DBPC) algorithm is particulary useful for a denser routing table with many non-branching paths. Let n_p denote the maximum number of prefixes included in a super-node. Similarly to the *SP-DBPC*, the *MP-DBPC* takes a unit-bit trie and the maximum skip-distance as inputs. The only new parameter is the maximum number of prefixes per super-node n_p. At each node of the trie (starting from the root), *MP-DBPC* finds the non-branching path P of the current node that has *at most* n_p prefixes and calculates the skip-distance d. Let m denote the number of nodes following the current node on path P that can be merged with the current node. m can be calculated as $\min(d, D)$. These m nodes are merged with the current node. The skip-string of the current node is updated. The child-nodes of the last merged node become the child-nodes of the current node. The *MP-DBPC* algorithm is then executed recursively until the leaf-node is reached. Once the algorithm completes, the total number of nodes is calculated and returned. The details are described in Algorithm 2.

In hardware implementation, each node of the *MP-DBPC* trie has the following data fields: (1) 2 child pointers, (2) skip-string, (3) length of actual skip-string, (4) position of prefix bit(s), and (5) next hop information. The total memory requirement can be calculated as:

$$M_{MP} = N_{MP} \times (2A + D + (n_p + 1)\lceil \log D \rceil + n_p H)$$

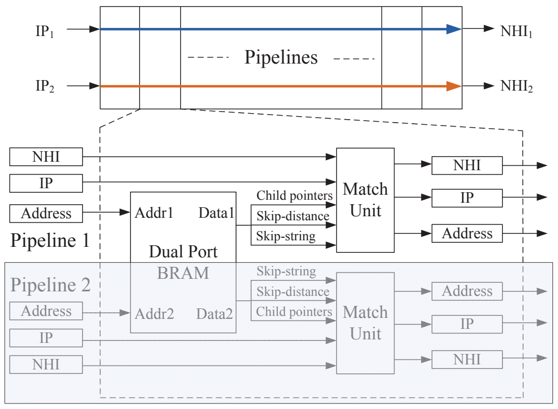


Fig. 5. Block diagram of the proposed IP-lookup architecture (NIH-Next hop information; IP-IP address)

Similarly to the *MP-DBPC*, if we restrict the last prefix bit to only be at the end of the skip-string, then each node needs not to store the position of this prefix bit. This restriction also simplifies the hardware implementation. The total memory footprint is reduced to Equ. 2. Note that when $n_p = 1$, the *MP-DBPC* algorithm becomes the *SP-DBPC*. As a result, Equ. 1 and 2 are identical, and $M_{MP} = M_{SP}$.

$$M_{MP} = N_{MP} \times (2A + D + n_p\lceil \log D\rceil + n_p H) \qquad (2)$$

To further optimize the memory consumption, a bit-vector B of length D can also be used to keep track of the position of the prefix bit(s) in the skip-string. In this case, data field #4 is replaced by bit-vector B. The total memory requirement is:

$$M'_{MP} = N_{MP} \times (2A + 2D + \lceil \log D\rceil + n_p H) \qquad (3)$$

Comparing Equ. 2 and 3, M'_{MP} is more efficient than M_{MP} if and only if Inequality 4 is satisfied. It shows that when n_p is larger than $\frac{D}{\lceil \log D\rceil}$, the bit-vector design should be used to achieve better memory efficiency.

$$(n_p - 1)\lceil \log D\rceil \geq D \Leftrightarrow n_p \geq \frac{D}{\lceil \log D\rceil} + 1 \qquad (4)$$

The detail evaluation of the *MP-DBPC* algorithm is presented in Section V.

Search in a MP-DBPC trie: IP lookup in a *MP-DBPC* trie is identical to that in the *SP-DBPC* trie. The only difference is in Step 3, where all the prefixes stored at the node are checked for a match. In case of multiple matches, the next hop information of the the longest match is returned.

Complexity: Similarly to the *SP-DBPC*, the *MP-DBPC* algorithm has a complexity of $O(N_P \times L_P)$.

IV. Architecture and Implementation on FPGA

A. Architecture

Pipelining is used to achieve the high throughput. Each level of the trie is mapped onto a pipeline stage, which has its own memory (or table). The block diagram of the basic pipeline and a single stage are shown in Fig. 5. The pipelined architecture can sustain one lookup operation per clock cycle. The on-chip memory of FPGA is dual-ported. To take advantage of this, the architecture is configured as dual-linear pipelines. This configuration doubles the lookup rate. At each stage, the memory has 2 sets of Read/Write ports so that two IP addresses can be input every clock cycle. In each pipeline stage, there are 3 data fields forwarded from the previous stage: (1) *IP address*, (2) *next hop information*, and (3) *memory address*. The memory address is used to retrieve the node stored on the local memory. The node value is compared with the input IP address in the *match unit* to determine the match status and the direction of traversal, as described in Section III. The *memory address* is updated in each stage depending on the direction of traversal. However, the *next hop information* is only updated if a match is found at that stage. The search continues until it comes out of the pipeline and the carried match information is returned.

B. Update

A routing table update can be any of the three operations: (1) modification of an existing prefix (i.e. change of the next hop information), (2) deletion of an existing prefix, and (3) insertion of a new prefix. The first update requires changing the next hop indices of the existing prefixes in the routing table, while the others require inserting a prefix into, or deleting a prefix from a given routing table.

We update the memory in the pipeline by inserting write bubbles, as described in [5], [13]. The new content of the memory is computed off-line. When an update is initiated, a write bubble is inserted into the pipeline. Each write bubble is assigned an ID. There is one write bubble table (WBT) in each stage. It stores the update information associated with the write bubble ID. When it arrives at the stage prior to the stage to be updated, the write bubble uses its ID to lookup the WBT. Then it retrieves (1) the memory address to be updated in the next stage, (2) the new content for that memory location, and (3) a write enable bit. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. Note that for one route update, we may need to insert multiple write bubbles consecutively.

C. Implementation on FPGA

A shown in Section V-B, the optimal value of the skip-distance and the maximum number of prefixes $\{n_p, D\}$ is $\{3, 2\}$ (IPv4) and $\{27, 1\}$ (IPv6). Hence, in IPv4 implementation, the bit-vector approach is employed and Equ. 3 is used to calculate the memory requirement. For IPv6, Equ. 1 is used. In the equations, let $H = 5$, $A = 16$. The size of memory for IPv4 (M_{v4}) and IPv6 routing tables (M_{v6}) can be shortened to: $M_{v4} = 50N_{v4}$ and $M_{v6} = 69N_{v6}$, where N_{v4} and N_{v6} denote the total number of nodes in the IPv4 and IPv6 compressed tries, respectively. The compressed trie of a core IPv4 routing table with 330K prefixes has about 530K nodes. This trie requires 26.5 Mb of memory. Similarly, a core IPv6 routing table with 330K prefixes needs 47 Mb of memory. Assuming the same node distribution, a state-of-the-art FPGA device with 36 Mb of on-chip memory (e.g. Xilinx

TABLE II
NUMBER OF PREFIXES OF REAL-LIFE AND SYNTHETIC ROUTING TABLES

Core routing tables			Edge routing tables		
IPv4	IPv6	# prefixes	IPv4	IPv6	# prefixes
rrc00	rrc00_6	332118	rrc00_e	rrc00_e6	95048
rrc01	rrc01_6	324172	rrc01_e	rrc01_e6	98390
rrc03	rrc03_6	321617	rrc03_e	rrc03_e6	90867
rrc04	rrc04_6	347232	rrc04_e	rrc04_e6	95358
rrc05	rrc05_6	322997	rrc05_e	rrc05_e6	98305
rrc06	rrc06_6	321577	rrc06_e	rrc06_e6	97410
rrc07	rrc07_6	322557	rrc07_e	rrc07_e6	95007
rrc10	rrc10_6	319952	rrc10_e	rrc10_e6	97376
rrc11	rrc11_6	323668	rrc11_e	rrc11_e6	91031
rrc12	rrc12_6	320015	rrc12_e	rrc12_e6	97386
rrc13	rrc13_6	335154	rrc13_e	rrc13_e6	95163
rrc14	rrc14_6	325797	rrc14_e	rrc14_e6	95032
rrc15	rrc15_6	323986	rrc15_e	rrc15_e6	91019
rrc16	rrc16_6	328295	rrc16_e	rrc16_e6	94780

Virtex6) can support up to 450K prefixes (for IPv4), or up to 250K prefixes (for IPv6), *without* using external SRAM.

In our design, external SRAMs can be used to handle even larger routing tables, by moving some stages of the pipelines onto external SRAMs. Currently, SRAM is available in $2-32$ Mb chips, with data widths of 18, 32, or 36 bits, and a maximum access frequency of over 500MHz. Each stage uses dual port memory, which requires two address and two data ports. Hence, each external stage requires ≈ 150 and ≈ 190 I/O pins for IPv4 and IPv6, respectively. Note that Quad Data Rate (QDR) SRAM can also be used in place of SRAM to provide higher chip-density and access bandwidth. The largest Virtex package, which has 1517 I/O pins, can interface with up to 8 banks of dual port SRAMs for IPv4, and up to 6 banks for IPv6. Moreover, since the access frequency of SRAM is twice that of our target frequency (200 MHz), the use of external SRAM will not adversely affect the throughput of our design.

Note that for routing tables with different distribution than that of the analyzed tables, the proposed algorithms need to re-calculate the optimal value of D. If D changes, the architecture also changes, and the entire FPGA chip needs to be reconfigured. This is the advantage of FPGA over ASIC.

V. PERFORMANCE EVALUATIONS

A. Experimental Setup

Fourteen experimental IPv4 core routing tables were collected from Project - RIS [1] on 06/03/2010. These core routing tables were used to evaluate our algorithm for a real networking environment. Additionally, synthetic IPv4 routing tables generated using FRuG [11] were also employed as we did not have access to any real provider edge routing tables. FRuG takes a seed routing table and generates a synthetic table with the same statistic as that of the seed. From these synthetic core and edge routing tables, we generated the corresponding IPv6 edge routing tables using the same method as in [23]. The IPv4-to-IPv6 prefix mapping is *one-to-one*. Hence, the number of prefixes in an IPv4-IPv6 (i.e. {rrc00 and rrc00_6}, {rrc00_e and rrc00_e6}) table pair is identical. The numbers of prefixes of the experimental routing tables are shown in Table II.

TABLE III
MEMORY REQUIREMENT (IN MBIT) OF DIFFERENT DESIGNS

	Core routing tables		Edge routing tables	
Design	IPv4	IPv6	IPv4	IPv6
Our design	25.52	46.07	7.24	13.90
UBT design	30.05 (1.18×)	283.79 (6.16×)	13.48 (1.86×)	84.72 (6.09×)
OCT design	45.16 (1.77×)	69.12 (1.50×)	11.50 (1.59×)	21.20 (1.53×)

B. Memory Efficiency

We first fix the maximum number of prefixes per node n_p at 1, and vary the skip-distance D in $[0:31]$ and $[0:63]$ for IPv4 and IPv6, respectively. The memory requirement is calculated for each case and the results are plotted in Fig. 6. The top group of curves is for the core tables, and the bottom is for the edge tables. It shows that all the experimental routing tables have similar characteristic. As the skip-distance D increases, the memory size reduces then increases. It reaches its minimum when $D=3$ for IPv4 and $D=27$ for IPv6. Another observation is that the effect of D on the memory requirement. When D increases from its optimal value, the memory requirement increases quicker in IPv4 than in IPv6. This is because the IPv4 trie is denser than the IPv6 trie. Hence, when D increases, the number of nodes decreases faster in IPv6 than in IPv4.

We also performed experiments with different values of D and the maximum number of prefixes n_p to find their optimal value. In these experiments, $n_p \in [1, 32]$ for IPv4, and $[1, 64]$ for IPv6. Experiments with all the routing tables in Table II show that the optimal value of pair $\{D, n_p\}$ is $\{3, 2\}$ and $\{27, 1\}$ for IPv4 and IPv6, respectively.

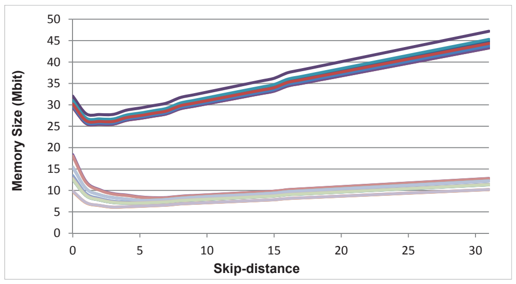
C. Performance Comparison

The memory efficiency of our design are compared with that of the straight-forward uni-bit trie (UBT) and the original compressed trie (OCT). Note that UBT is a special case of our algorithms when $D=0$ and $n_p=1$. Similarly, OCT is a special case when $D=31$ (in IPv6, $D=63$) and $n_p=1$. The memory requirement of 3 designs are depicted in Table III.

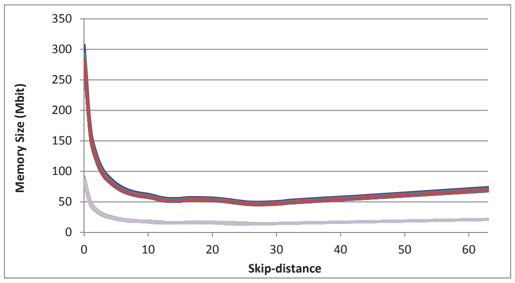
The memory efficiency of our design outperforms that of the UBT design 1.86× and 6.16× in IPv4 and IPv6, respectively. Similarly, our design outperforms the OCT design 1.77× and 1.53× in IPv4 and IPv6, respectively.

D. Throughput

The proposed architectures (IPv4 and IPv6) were simulated and implemented in Verilog, using Xilinx ISE 12.4, with Virtex-6 XC6VSX475T as the target. These architectures supports the largest backbone IPv4/v6 routing tables consisting of 330K prefixes. The place and route results are collected in Table IV. The IPv4 architecture fits in the on-chip BRAM of the FPGA. However, the IPv6 architecture requires some stages moved onto external SRAMs. As previously described in Section IV-C, at most 6 dual-ported SRAM banks can be interfaced with the FPGA chip. Hence, 6 largest stages are moved onto external SRAMs. With this configuration, the IPv6



(a) IPv4



(b) IPv6

Fig. 6. Memory requirement of all the IPv4/v6 experimental routing tables for different skip-distances with single prefix per super-node

TABLE IV
IMPLEMENTATION RESULTS

Architecture	Clock period (ns)	Slices	BRAM (36-Kb block)	SRAM (Mbit)
IPv4	4.286	3524	756	0
IPv6	5.372	8252	448	36

architecture employs only 36 Mbit of SRAM to support a core IPv6 routing table. Using dual-ported memory, the design can support 466 and 372 million lookups per second (MLPS) for IPv4 and IPv6, respectively. These results surpass the worst-case 150 MLPS required by the standardized 100GbE line cards [21].

VI. CONCLUDING REMARKS

In this paper, we have described a set of algorithms based on *distance-bounded path compression* to compress a given routing table. By exploiting the intrinsic distribution of the IPv4/v6 forwarding tables, these algorithms bound the maximum skip-distance and the maximum number of prefixes per node to achieve optimal memory efficiency. Additionally, using the compact data structures, these algorithms achieve not only high memory efficiency, but also low memory bandwidth. The proposed algorithms and their implementation on FPGAs also scale well to support very large forwarding tables, by utilizing external SRAMs. With these advantages, our algorithms can be used to improve the performance (throughput and memory efficiency) of any trie-based IPv4/v6 lookup schemes to satisfy the following criteria: (1) fast internet link rates up to and beyond 100 Gbps at core routers, (2) increase in routing table size at the rate of 25-50K prefixes per year, (3) increase in the length of the prefixes from 32 to 128 bits in IPv6, (4) compact memory footprint that can fit in the on-chip caches of multi-core and network processors, and (5) reduction in per-virtual-router storage memory of network virtual routers. We plan to extend these algorithms to have dynamic skip-distance and number of prefixes per node for different trie levels. We also like to apply these algorithms in virtual routers to improve their memory efficiency.

REFERENCES

[1] RIS RAW DATA [Online]. [http://data.ris.ripe.net].
[2] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Softw. Pract. Exper.*, 25:129–141, February 1995.
[3] M. Attig and G. Brebner. Systematic characterization of programmable packet processing pipelines. *Annual IEEE Symposium on FCCM*, 0:195–204, 2006.
[4] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Proc. ISCA '05*, pages 123–133, 2005.
[5] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. *IEEE/ACM Trans. Netw.*, 13:690–703, June 2005.
[6] M. Behdadfar, H. Saidi, H. Alaei, and B. Samari. Scalar prefix search - a new route lookup algorithm for next generation internet. In *Proc. INFOCOM '09*, 2009.
[7] H. J. Chao and B. Liu. *High performance switches and routers*. JohnWiley & Sons, Inc., Hoboken, NJ, USA, 2007.
[8] J. Dharmapurikar, S. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781–1792, 2006.
[9] H. Fadishei, M. S. Zamani, and M. Sabaei. A novel reconfigurable hardware architecture for IP address lookup. In *Proc. ANCS '05*, pages 81–90, 2005.
[10] M. French, E. Anderson, and D.-I. Kang. Autonomous system on a chip adaptation through partial runtime reconfiguration. In *FCCM '08*, pages 77–86. IEEE Computer Society, 2008.
[11] T. Ganegedara, W. Jiang, and V. Prasanna. Frug: A benchmark for packet forwarding in future networks. In *IPCCC '10*, 2010.
[12] M. Gokhale, J. Cohen, A. Yoo, W. M. Miller, A. Jacob, C. Ulmer, and R. Pearce. Hardware technologies for high-performance data-intensive computing. *Computer*, 41:60–68, April 2008.
[13] H. Le, W. Jiang, and V. K. Prasanna. A sram-based architecture for trie-based ip lookup using fpga. In *Proc. FCCM '08*, 2008.
[14] H. Le and V. K. Prasanna. Scalable high throughput and power efficient ip-lookup on fpga. In *Proc. FCCM '09*, 2009.
[15] H. Lu and S. Sahni. A b-tree dynamic router-table design. *IEEE Trans. Comput.*, 54(7):813–824, 2005.
[16] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
[17] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23, 2001.
[18] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani. Scalable, memory efficient, high-speed ip lookup algorithms. *IEEE/ACM Trans. Netw.*, 13(4):802–812, 2005.
[19] K. Sklower. A tree-based packet routing table for berkeley unix. In *Winter Usenix Conf.*, pages 93–99, 1991.
[20] H. Song, F. Hao, M. Kodialam, and T. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *Proc. INFOCOM '09*, 2009.
[21] H. Song, M. S. Kodialam, F. Hao, and T. V. Lakshman. Scalable ip lookups using shape graphs. In *Proc. ICNP '09*, 2009.
[22] I. Sourdis, G. Stefanakis, R. de Smet, and G. N. Gaydadjiev. Range tries for scalable address lookup. In *Proc. ANCS '09*, 2009.
[23] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random generator for ipv6 tables. In *HOTI '04*, pages 35–40, 2004.
[24] P. Warkhede, S. Suri, and G. Varghese. Multiway range trees: scalable ip lookup with fast updates. *Comput. Netw.*, 44(3):289–303, 2004.
[25] X. Zhang, B. Liu, W. Li, Y. Xi, D. Bermingham, and X. Wang. Ipv6-oriented 4xoc-768 packet classification with deriving-merging partition and field-variable encoding algorithm. In *INFOCOM'06*, pages 1–12, Piscataway, NJ, USA, 2006. IEEE Press.