Hybrid Data Structure for IP Lookup in Virtual Routers Using FPGAs

Oğuzhan Erdem
Electrical and
Electronics Engineering
Middle East Technical University
Ankara, TURKEY 06800
Email: ogerdem@metu.edu.tr

Hoang Le, Viktor K. Prasanna
Ming Hsieh Department of
Electrical Engineering
University of Southern California
Los Angeles, USA 90007
Email: {hoangle, prasanna}@usc.edu

Cüneyt F. Bazlamaçcı
Electrical and
Electronics Engineering
Middle East Technical University
Ankara, TURKEY 06800
Email: cuneytb@metu.edu.tr

***Abstract*—Network router virtualization has recently gained much interest in the research community, as it allows multiple virtual router instances to run on a common physical router platform. The key metrics in designing network virtual routers are (1) number of supported virtual router instances, (2) total number of prefixes, and (3) ability to quickly update the virtual table. Existing merging algorithms use leaf pushing and a shared next hop data structure to eliminate the large memory bandwidth requirement. However, the size of the shared next hop table grows linearly with the number of virtual routers. Due to the limited amount of on-chip memory and the number of I/O pins of Field Programmable Gate Arrays (FPGAs), existing designs cannot support large number of tables and/or large number of prefixes.**

This paper exploits the abundant parallelism and on-chip memory bandwidth available in the state-of-the-art FPGAs, and proposes a compact trie representation and a hybrid data structure to reduce the memory requirement of virtual routers. The approach does not require leaf-pushing; therefore, reduces the size of each entry of the data structure. Our algorithm demonstrates substantial reduction in the memory footprint compared with the state-of-the-art. Also, it eliminates the shared next hop information data structure and simplifies the table updates in virtual routers. Using a state-of-the-art FPGA, the proposed architecture can support up to 3.1M IPv4 prefixes. Employing the dual-ported memory available in the current FPGAs, we map the proposed data structure on a novel SRAM-based linear pipeline architecture to achieve high throughput. The post place-and-route result shows that our architecture can sustain a throughput of 394 million lookups per second, or 126 Gbps (for the minimum packet size of 40 Bytes).

I. Introduction

A. Internet Protocol (IP) Lookup

IP packet forwarding, or simply, IP-lookup, is a classic problem. In computer networking, a routing table is a database that is stored in a router or a networked computer. A routing table stores the routes and metrics associated with those routes, such as next hop routing indices, to particular network destinations. The IP-lookup problem is referred to as "*longest prefix matching*" (LPM), which is used by routers in IP networking to select an entry from the given routing table. To determine the outgoing port for a given address, the longest matching prefix among all the prefixes needs to be searched. Routing tables often contain a default route in case matches with all other entries fail.

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

Most hardware-based solutions for network routers fall into two main categories: TCAM-based and dynamic/static random access memory (DRAM/SRAM)-based solutions. In TCAM-based solutions, each prefix is stored in a word. An incoming IP address is compared in parallel with all the active entries in TCAM in one clock cycle. TCAM-based solutions are simple, and therefore, are de-facto solutions for today's routers. However, TCAMs are expensive, power-hungry, and offer little adaptability to new addressing and routing protocols [1], [2].

On the other hand, SRAM has higher density, lower power consumption, and higher speed [3]. The common data structure in SRAM-based solutions for performing LPM is some form of a tree. In these solutions, multiple memory accesses are required in order to find the longest matched prefix. Therefore, pipelining techniques are used to improve the throughput [3], [4], [5], [6].

B. Network Virtualization

Network virtualization helps reduce the cost of hardware and makes the manageability of computing resources simpler [7], [8]. The main goal of virtualization is to efficiently use the networking resources. Virtual router is the basic component of a virtual network. Multiple virtual routers can run on a single router, and multiple organizations can share this single physical router. A single router plays the role of multiple independent virtual routers. Hence, a virtual router should fulfill the following requirements: *Fair resource usage*, *Fault isolation*, and *Security*.

Several ideas have been proposed to realize router virtualization on a single hardware networking platform [9], [10]. These approaches use leaf pushing [11] and an additional shared search structure to eliminate the high memory bandwidth requirement. However, leaf pushing increases the complexity in virtual table updates and also decreases the memory efficiency by introducing data redundancy in the shared next hop structure. This redundancy increases with the number of virtual routers. Thus, these solutions do not scale well as the number of virtual routers increases.

This paper makes the following contributions:

1) A compact trie representation and a hybrid data structure for IP lookup that reduce the memory requirement. This structure does not need backtracking while preforming a search.
2) A merging algorithm that eliminates leaf pushing and simplifies the table updates in virtual routers.
3) A linear pipelined SRAM-based architecture on FPGAs that can support up to 3.1M IPv4 prefixes, while achieving a sustained throughput of 394 million lookups per second.

The rest of the paper is organized as follows. Section II overviews the existing solutions for IP lookup in virtual routers. Section III presents our hybrid data structure. Section IV describes the IP lookup architecture. The experimental results are given in Section V. Section VI summarizes the paper and gives future directions.

II. Background

A. Trie-based LPM

Binary trie is the most common and simple data structure for IP lookup. In a binary trie, the path from the root to a node represents a prefix in a routing table. Given a binary trie, IP lookup starts from the root node and continues traversing the binary trie using the bits of the destination IP address. Search operation terminates when a leaf node or a null pointer is reached. The last matched prefix is selected as the longest matched prefix. If all the prefix nodes are pushed to leaves, then the binary trie is called a leaf-pushed binary trie [11]. In such a trie, a non-leaf node contains only pointers to its children, and the leaf node contains only the next hop information associated with the corresponding prefix. Fig. 1 illustrates a sample prefix table and its binary trie. In this figure, each black node corresponds to a prefix.

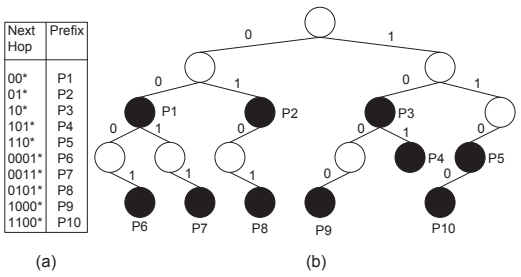

Fig. 1. (a) A sample prefix table (b) The corresponding binary trie

B. Related Work

Although the IP lookup problem has been extensively studied ([12], [11], [13]), only a few have been targeted for virtual routers. There are generally two approaches for IP lookup in virtual routers: *separate* and *shared* data structure. A separate data structure for each virtual forwarding information base (FIB) preserves the isolation among virtual routers at the cost of high memory and resource requirement. It also reduces the scalability (in the number of virtual routers) of the design [14]. Thus, the existing IP lookup algorithms are mostly based on shared data structures [9], [10].

Fu et. al in [10] used a shared data structure to realize router virtualization in the merged approach. They employed a simple overlaying mechanism to merge the virtual routing tables into a single routing table. By using the shared data structure, they have achieved significant memory saving, and showed the scalability of their solution for up to 50 routing tables. For this algorithm to achieve memory saving, the virtual routing tables must have *similar structure*. Otherwise, simple overlaying will result in the increase of memory usage significantly.

Trie braiding [9] is another algorithm introduced for the merged approach. The authors presented a heuristic to merge multiple virtual routing tables in an optimal way, in order to increase the overlap among different routing tables. A *braiding bit* is used at each node for each routing table to identify the direction of traversal at that node. Although the overlapping between tries increases and the number of nodes in the merged trie reduces, the size of each node also increases. Therefore, reduction in the total number of nodes does not necessarily lead to the reduction of the overall memory consumption. This scheme performs well only when the routing tables have different structure.

C. Discussion

In the existing merging schemes, applying leaf-pushing technique is inevitable. When the leaf-pushing technique is employed, the internal nodes contain only the pointer to the child nodes, and the leaf nodes contain the pointers to the next hop. However, leaf-pushing has two disadvantages. First, it makes route updates harder and time consuming. Second, the use of leaf-pushing in a merged binary trie creates storage redundancy. In order to demonstrate this case, we use an example of two virtual tables shown in Fig. 2. When leaf-pushing is used, the next hop information P_5 in FIB2 is redundantly copied to multiple nodes in the merged trie. We consider 2 cases: (1) the tries are leaf-pushed individually and (2) the tries are merged and the merged trie is leaf-pushed. Let S_1 and S_2 denote the total number of next hops in these 2 cases, respectively. *Storage redundancy* is defined as $\Delta = S_2 - S_1$, which shows the number of prefixes extended redundantly to some longer prefixes. In the example shown in Fig. 2, the storage redundancy is $\Delta = 2$ (or 20%), where $S_2 = 10$ and $S_1 = 8$.

Our motivation is to use a shared data structure approach without leaf-pushing. In this paper, we propose a new hybrid data structure and a method to merge virtual routing tables.

III. Hybrid Data Structure for IP Lookup in Virtual Routers

A. Definitions

Definition ***Prefix node*** in a trie is any node to which a path from the root of the trie corresponds to an entry in the routing table. If there is no valid prefix stored in a trie node, then it is called ***non-prefix node***.

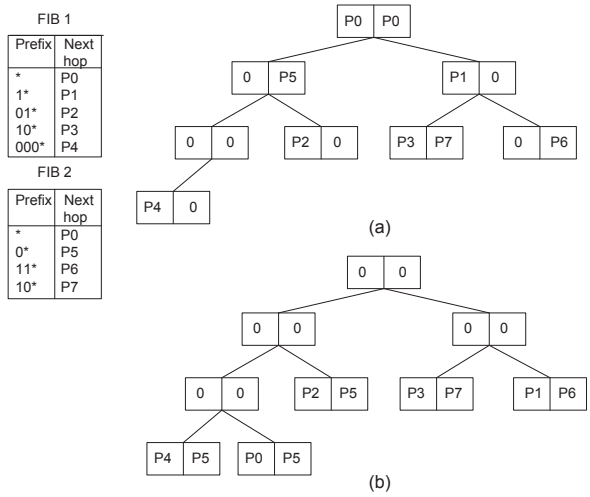

Fig. 2. (a) Shared binary trie (b) Leaf pushed shared binary trie

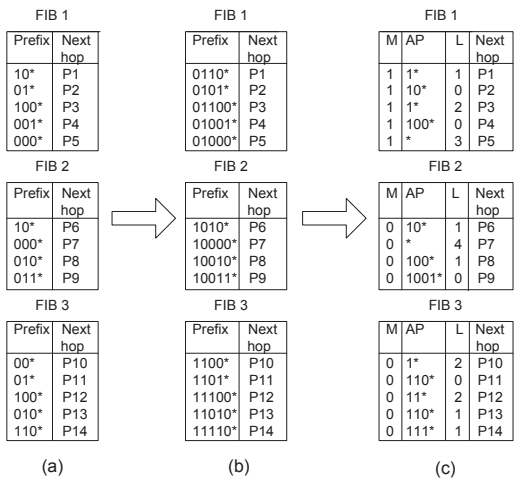

Fig. 3. (a) Virtual tables (b) Virtual id prepending (c) Converted virtual tables

Definition Each virtual router has its own table which is called ***virtual routing table*** or ***virtual table***. Each virtual router is associated with a ***virtual ID***.

Definition ***Active part*** (AP) of a prefix is the bit string between the most and least significant set bits of a prefix including the most significant set bit but not the least significant set bit. For instance, the active parts of the prefixes $011010*$ and $0010100*$ are 110 and 10, respectively.

Definition If two prefixes have the same active part, then they are called ***conflicted prefixes***. For instance, the prefixes $0101*$ and $001010*$ are conflicted because they both have same the active part of 10.

Definition If a trie node is shared by more than one prefix, then it is called a ***super prefix node***.

B. Prefix table conversion

Each virtual routing table is converted prior to constructing the trie. The conversion process includes two steps: (1) *virtual ID prepending* and (2) *prefix conversion*. Fig. 3 illustrates these steps for three sample FIBs.

Virtual ID prepending: In this step, prefixes in each virtual routing table are prepended with the corresponding virtual ID information. The number of virtual ID bits is $\lceil \log_2 k \rceil$, where k is the number of virtual tables. The initial tables are converted to new tables after prepending virtual ID as presented in Fig. 3b. Note that these sample tables have the virtual ID of 01, 10 and 11.

Prefix conversion: The trie is built using only the *active part* (AP) of the prepended prefixes. Two *conflicted prefixes* are distinguished from each other either by their most significant set bit position or the number of 0's following their least significant set bit, or both. For instance, the two conflicted prefixes $01001*$ and $100100*$ have different most significant set bit positions (1 and 0) and the number of trailing zeros (0 and 2). The two conflicted prefixes may not only be in the same virtual table, but also in the different virtual tables.

In a single virtual table, all the prefixes have the same virtual ID. Hence, these prefixes have the same most significant set bit position. Two prefixes in the same virtual table may conflict if and only if they have the same active part but different number of trailing zeros. For instance, prefixes $1010*$ and $101000*$ in the same table have the same active part, even when we prepend them with any virtual ID. However, they have different numbers of trailing zeros (1 and 3). Two prefixes in different virtual tables may also conflict. For example, prefix $1001*$ in virtual table 01 and prefix $001*$ in virtual table 11 will have the same active part (1100) after prepending the virtual IDs. However, they have different most significant bit positions (0 and 1). These two cases imply the necessity of storing most significant set bit position (M) and the number of trailing zeros (L) in addition to the active part (AP) for each prefix to distinguish the conflicted prefixes. The size of L is $\lceil \log_2(W + \lceil \log_2 k \rceil) \rceil$, where W denotes the maximum length of an IP address (32 in IPv4 and 128 in IPv6), and k is the number of virtual tables. The size of most significant bit position field is also $\lceil \log_2(W + \lceil \log_2 k \rceil) \rceil$, but it can be reduced to $\lceil \log_2(\lceil \log_2 k \rceil) \rceil$ if the virtual ID including all 0's is not used. For instance, for 8 virtual tables in IPv4, the size of M and L must be 2 and 6 bits, respectively. Therefore, each prepended prefix is represented by a tuple $\langle M, L, AP \rangle$. The sample converted tables are shown in Fig. 3c.

Algorithm 1 TableConversion

Input: RT_i where $0 \leq i < K$, K is the number of virtual tables; number of prefixes in each table N_i
Output: $RT_i converted$

```
for i = 0 to K − 1 do
    for j = 0 to N_i − 1 do
        Prepend Virt_ID_i to prefix P_ij in RT_i
        Find M_ij, L_ij and extract Active Part (AP_ij)
        Store prefix P_ij in RT_i_converted as (M_ij,AP_ij,L_ij)
    end for
end for
```

The prefix table conversion algorithm is presented in Algo-

rithm 1. Virtual ID prepending is performed in Line 3, while Lines 4-5 perform the prefix conversion.

C. Constructing the Hybrid Data Structure

In the trie construction step, a ***single*** binary trie is constructed for all the prepended prefixes of the virtual tables, using only their active part. The resulting trie is shallower and denser than a traditional binary trie. We call the resulting trie a *compressed binary trie* or simply *compressed trie*. The search algorithm for this compact trie is presented in the next subsection. As previously stated, extra information is stored at each node to distinguish the conflicted prefixes. Therefore, in addition to the child pointers and the next hop information fields, the most significant set bit position M and length information L are also required.

The corresponding next hop information of the conflicted prefixes is also stored at each trie node. However, the variable number of conflicted prefixes at each node results in memory inefficiency in hardware implementation. An auxiliary data structure can be constructed for the conflicted prefixes. Yet, a secondary search is required at each node. Alternatively, backtracking can be employed to get the correct next hop result. In backtracking, search proceeds in the backward direction after it fails to find a match in the forward direction. Backtracking in hardware pipelining requires either stalling the pipeline or duplicating the memory for backward searching, and this is not desirable.

Our analysis of the synthetic and real routing tables collected from [15] shows that the number of conflicted prefixes is less than 10% of the total number of prefixes. Hence, it is more efficient to introduce a single *secondary search structure* to store these prefixes. Consequently, only one prefix of each group of the conflicted prefixes is stored in the compact trie (the *primary search structure*). The remaining prefixes are moved to a secondary data structure or a TCAM. Note that searches in these structures are performed in parallel.

Claim: Backtracking is eliminated using the proposed hybrid data structure.

Proof: First, in the primary search structure, each node stores at most one prefix. The search in this structure can proceed in the forward direction. Secondly, all the remaining conflicted prefixes are stored in a single secondary search structure instead of one such structure for each super prefix node. The search in this structure can also proceed in the forward direction. Finally, searches in both structures are performed independently in parallel. Therefore, backtracking is not required. ■

In this paper, we use a binary search tree (BST) for the secondary search structure. Yet, other alternatives can also be used. In a BST, each node has a value (prefix) and an associated next hop index. The left subtree of that node contains only values less than or equal to the node's value and the right subtree contains values greater than the node's value. The BST construction algorithm is described in detail in [16]. Fig. 4 shows the hybrid data structure composed of a trie (primary search structure) and a tree (secondary search structure) for the sample prefixes in Fig. 3.

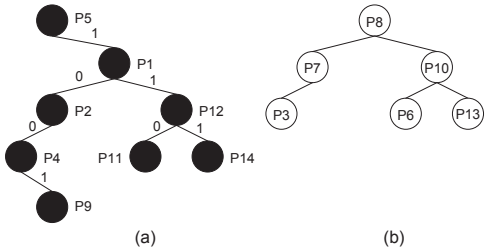

Fig. 4. (a) Compact binary trie (b) A binary tree

D. IP Lookup algorithm

For each incoming packet, the destination IP address is extracted. The address is prepended with the virtual ID corresponding to that packet. The prepended IP address is searched in the trie and tree data structures in parallel. The outputs from both data structures are compared and the longer match is returned as the final result.

Trie search: The active part is obtained from the prepended address. In the search operation, 3 data fields are used: (1) active part AP_{key}, (2) the most significant set bit position M_{key} of the prepended IP address, and (3) the latest matched next hop information NHI_{key}. Search starts from the root node and at each node visited, AP_{key} is left-shifted by one bit. Match is determined by the M and L fields stored at the node. The direction of traversal is determined by the most significant bit of AP_{key} (left if 0 or right otherwise). If there is a match in a node then NHI_{key} is updated. The search ends when a leaf node is reached. The search algorithm for binary trie is presented in Algorithm 2. The notations used in the algorithm are shown in Table I.

TABLE I
LIST OF NOTATIONS USED IN THE ALGORITHM

Notation	Meaning
AP_{key}	Active part of key (virtual ID.IP address)
M_{key}	Most significant set bit position of key
NHI_{key}	Next hop information for key
L_{key_int}	The number of consucutive zeros after the most significant set bit in AP_{key}.
B_0	Most significant bit value in AP_{key}
M_{prefix}	Most significant bit position of prefix
L_{prefix}	Length of succeeding zeros in prefix
NHI_{prefix}	Next hop information of prefix

Tree search: IP lookup in BST is performed by comparing the searched IP address with the data value stored at each node. The tree is traversed left or right, depending on the comparison result at each node. Search ends when the IP address matches with the value stored in the node, or when a leaf node is reached.

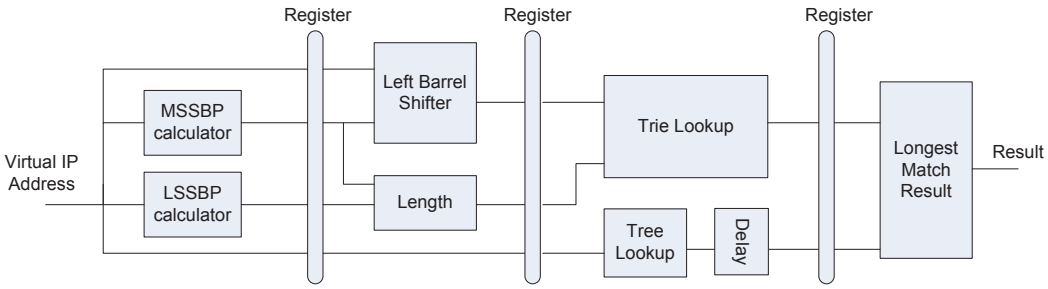

Fig. 5. Block diagram of the IP lookup architecture

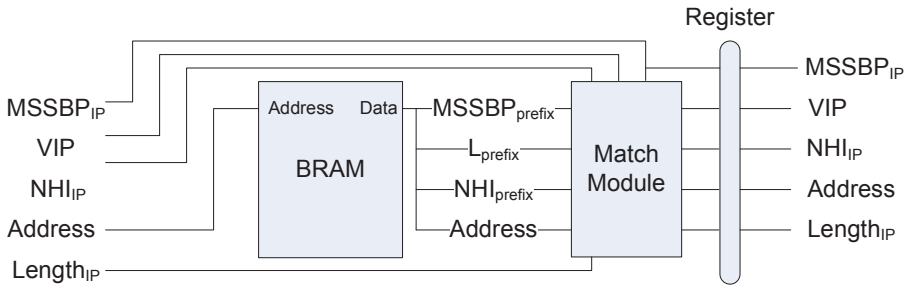

Fig. 6. A basic stage of the pipeline for the trie lookup

Algorithm 2 IP_lookup

Input: Virtual ID, Destination IP address
Output: NHI_{key}

Prepend $Virt\_ID$ to IP address
Find M_{key} and extract Active Part (AP_{key})
Start search from root node, traverse trie nodes using B_0
while current node is not leaf node **do**
  shift AP_{key} one bit left
  if $M_{key}! = M_{prefix}$ **then**
    no match
  else if $B_0 == 0$ **then**
    no match
  else if $L_{prefix} == 0$ **then**
    match and update NHI_{key} = NHI_{prefix}
  else if $L_{key\_int} \geq L_{prefix}$ **then**
    match and update NHI_{key} = NHI_{prefix}
  else
    no match
  end if
  Update current node
end while
return NHI_{key}

IV. Architecture and Implementation on FPGA

A. Architecture

Pipelining is used to improve the throughput. There are two pipelines: the trie and the tree pipelines. The number of pipelines stages are determined by the height of the trie/tree used. Fig. 5 describes the overall architecture of the proposed IP lookup engine for virtual routers. A delay block is added at the end of the tree path to match the latency of the two search paths.

The IP address and virtual ID are extracted from the incoming packet and concatenated to form the virtual IP address (VIP). The VIP is routed to both pipelines and the searches are performed in parallel. The results are fed through a priority resolver to select the next hop index of the longest matched prefix.

Trie search: In Fig. 5, the MSSBP and LSSBP blocks calculate the most and least significant set bit positions of VIP, respectively. The active part (AP) of VIP is extracted by the left barrel shifter. The length of AP is used to early terminate the search once its last bit is checked.

There are at most $W + \lceil \log_2 k \rceil$ stages in the trie pipeline. Fig. 6 presents a single pipeline stage. Each stage includes a match module and a BRAM where the trie nodes are stored. The BRAM in FPGAs supports dual-ported feature. To take advantage of it, the architecture is configured as dual-linear pipelines to double the lookup rate. At each stage, the memory has dual Read/Write ports so that two packets can be input every clock cycle.

The inputs for each pipeline stage include: (1) virtual IP address (VIP), (2) most significant set bit position ($MSSBP_{IP}$) of VIP, (3) next hop information (NHI_{IP}), and (4) $length_{IP}$. The memory address, which is used to retrieve the node stored in BRAM, is forwarded from the previous stage. At each stage, the VIP bits are used to determine the direction of traversal, the memory address is updated, VIP is shifted one bit to the left, and the $length_{IP}$ is decremented by one. If there is a match in a stage, then NHI_{IP} is also updated. When $length_{IP} = 0$, the search can be terminated. By doing so, power saving becomes possible

since no lookup operation is required in the subsequent stages.

Each entry in BRAM includes 5 data fields: (1) left child node address, (2) right child node address, (3) next hop information (NHI_{prefix}), (4) most significant set bit position ($MSSBP_{prefix}$) of the prefix, and (5) the number of trailing zeros of the prefix (L_{prefix}). The operation of the match module is described in Algorithm 2.

Tree search: Binary search tree (BST) structure is utilized in our design. The number of pipeline stages is determined by the height of the BST. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table). The content of each entry in the memory includes the prefix and its next hop routing index. In each pipeline stage, there are 3 data fields forwarded from the previous stage: (1) the IP address, (2) the node address, and (3) the next hop index. The forwarded memory address is used to retrieve the node prefix, which is compared with the IP address to determine the matching status. In case of a match, the next hop index of the new match replaces the old result. The comparison result (1 if the IP address is greater than node prefix, 0 otherwise) is appended to the current memory address and forwarded to the next stage. IP lookup using BST is covered in detail in [16].

B. Virtual Routing Table Update

Virtual routing table updates include: (1) prefix insertion, (2) prefix deletion, and (3) route changes. In binary trie, the complexity of an update operation is $O(W)$, where W is the maximum length of a prefix (32 for IPv4 and 128 for IPv6). For every prefix update, the active part (AP), most significant set bit position (M), and number of trailing zeros (L) of that prefix are calculated. Prefix update procedure for the binary tree is covered in [16].

Prefix insertion: If the new prefix conflicts with the existing prefixes (having the same active part), then the new prefix is added to the tree data structure. Otherwise, it is added to the trie data structure.

Prefix deletion: The target prefix is located and its valid bit is reset.

Route change: The target prefix is located and its next hop information is updated.

C. Implementation on FPGA

Table II shows the list of notations used in the analysis. The memory consumption of the trie and the tree structures can be calculated using Equation 1 and 2.

$$M_{trie} = N_{trie} \times (2P + M + L + NHI) \tag{1}$$

$$M_{tree} = N_{tree} \times (2P + \lceil \log_2 k \rceil + W + \lceil \log_2 W \rceil + NHI) \tag{2}$$

If we assign $k = 16$ to support 16 virtual routing tables, then $M = \lceil \log_2(\lceil \log_2 k \rceil) \rceil = 2$ and $L = \lceil \log_2(\lceil \log_2 k \rceil + W) \rceil = 6$. We set $NHI = 6$ to support 64 next hop information, $P = 16$ to support up to 64K nodes in each level. By substituting these values in Equation 1 and 2, we get $M_{trie} = 46N_{trie}$ and $M_{tree} = 79N_{tree}$. We observed that the memory consumption increases linearly with the number of prefixes. A state-of-the-art FPGA device with 36 Mb of BRAM (e.g. Xilinx Virtex6) can support up to 380K prefixes (for IPv4) without using external SRAM (assuming the same prefix distribution).

TABLE II
LIST OF NOTATIONS USED IN MEMORY SIZE CALCULATIONS

Notations	Meaning
M_{trie}	Total memory size of trie
M_{tree}	Total memory size of tree
N_{trie}	Total number of trie nodes
N_{tree}	Total number of tree nodes
P	Number of pointer bits
k	Number of virtual tables
NHI	Number of bits to store the nexthop information
M	Number of bits to store $MSSBP_{prefix}$ ($\lceil \log_2(\lceil \log_2 k \rceil) \rceil$)
W	IP address length (32 for IPv4 and 128 for IPv6)
L	Number of bits to store the number of trailing zeros ($\lceil \log_2(\lceil \log_2 k \rceil + W) \rceil$)

On the other hand, external SRAMs can be used in our design to store larger routing tables. The last stages of pipelines can be moved onto external SRAMs. In this scheme one external SRAM is allocated to one stage avoiding sharing of external SRAMs by multi stages. Our analysis of synthetic and real routing tables collected from [15] shows that the trie occupies $\approx 92\%$ of the total memory consumption. Therefore, the BST is stored completely on BRAM, while some stages of the trie pipelines can be moved onto external SRAMs. Each stage uses dual-ported memory, which requires two address and two data ports. Hence, each external stage requires ≈ 126 pins going into the FPGA chip. The largest Virtex package, which has 1517 I/O pins, can interface up to 10 banks of dual-ported SRAMs. Equations 3-7 are used to estimate the maximum number of prefixes that can be supported. M_{BRAM} is the maximum amount of on-chip memory of the target FPGA device. M_{total} is the total memory consumption, and is defined in Equation 3.

$$M_{total} = M_{tree} + M_{trie} \tag{3}$$

$$M_{trie} = M^b_{trie} + M^s_{trie} \tag{4}$$

$$x = \frac{M_{tree}}{M_{total}} \qquad y = \frac{M^b_{trie}}{M_{trie}} \tag{5}$$

$$M_{BRAM} = M_{tree} + M^b_{trie} \tag{6}$$

$$M_{total} = \frac{M_{BRAM}}{x + y - xy} \tag{7}$$

D. Memory Requirement

Let M^b_{trie} and M^s_{trie} denote the amount of BRAM and external SRAM to store the trie, respectively. The required memory for trie can be defined as in Equation 4. Let x be the ratio of memory requirement for tree to total memory, y be the ratio of memory needed in BRAM for trie to the total memory required for storing trie (Equation 5). M_{BRAM} can also be written in terms of M_{tree} and M^b_{trie}, as shown

TABLE III
PREFIX DISTRIBUTION OF TRIE AND TREE IN REAL CORE IPV4 TABLES WITH HYBRID DATA STRUCTURE

	rrc00	rrc01	rrc02	rrc03	rrc04	rrc05	rrc06	rrc07	rrc08	rrc09
Total no of prefix	332117	324172	272743	321618	347232	322997	321578	322558	319953	323669
No of prefix (trie)	303077	296840	252394	294107	319152	295266	294557	295172	292733	296042
No of prefix (tree)	29040	27332	20349	27511	28080	27731	27021	27386	27220	27627

TABLE IV
NUMBER OF NODES (K)/ TOTAL MEMORY (MBIT) IN IPV4

No. of FIB	1	2	3	4	5	6	7	8	9	10
Separate	330/14	676/29	999/42	1339/57	1665/70	1985/84	2320/98	2662/112	3033/128	3422/144
Simple overlay 1	330/14	604/30	856/48	1113/71	1317/91	1532/114	1668/134	1732/149	1800/165	1945/190
Simple overlay 2	526/21	969/43	1365/67	1776/92	2096/114	2437/145	2655/165	2756/179	2864/194	3101/219
Hybrid	198/10	396/20	586/29	775/41	963/50	1138/60	1333/72	1522/84	1741/96	1977/109

in Equation 6. The total memory consumption M_{total} can be derived from these equations, and is given in Equation 7.

Our analysis of the experimental routing tables shows that $x \approx 0.08$ and $y \approx 0.02$. Thus, $M_{total} = 10.16 \times M_{BRAM}$. In a state-of-the-art FPGA device, $M_{BRAM} = 36$Mb. If we use the results in Section V, then our architecture can support up to 3.1M prefixes. Note that if the size of on-chip memory increases, then the number of prefixes supported by the architecture also increases.

V. PERFORMANCE EVALUATION

A. Experimental setup

Router virtualization is mainly used in provider edge networks. However, due to the unavailability of real provider edge routing tables, only synthetic routing tables generated using FRuG [17] were used. FRuG takes a seed routing table and generates a synthetic table with the same statistics as that of the seed. Ten synthetic IPv4 and IPv6 routing tables were generated, each of them consisted of 100K prefixes.

Additionally, ten experimental IPv4 core routing tables were collected from Project - RIS [15]. These tables were analyzed to see the occurrence of conflicted prefixes in real scenario.

B. Throughput

The architecture was configured to support 10 virtual routing tables, with a total of 1M prefixes. Since the BRAM is not sufficient to store these prefixes, the last 5 stages of the trie are moved onto external SRAMs. The proposed hardware design was simulated and implemented in Verilog, using Xilinx ISE 12.4, with Xilinx Virtex-6 XC6VSX475T with -2 speed grade as the target. The post place-and-route results are recorded in Table V. With the clock period of 5.084 ns, the design is capable of running at 197 MHz. Using dual-pipeline configuration, the architecture can support 394 million lookups per second (MLPS), or 126 Gbps (for the minimum packet size of 40 Bytes).

Table III shows the prefix distribution between the trie and tree data structures when only the prefix conversion step (Section III-B) of our algorithm is applied to the real IPv4 routing tables individually. The results show that the number of conflicted prefixes (prefixes stored in the tree) is less than 10% of the total number of prefixes in each table.

TABLE V
IMPLEMENTATION RESULTS

Clock period (ns)	Frequency (MHz)	Number of Slices	BRAM (36-Kb block)	SRAM (Mbit)
5.084	197	4058	720	72

Four algorithms were investigated: (1) *separate*, (2) *simple overlaying 1* (that does not do leaf pushing) (3) *simple overlaying 2* (that does leaf pushing), and (4) *hybrid*. Table IV shows the memory requirement of these algorithms for various numbers of virtual routing tables. Note that the results are presented in A/B format, where A indicates the total number of nodes, and B is the total memory size (in Mbits). Table VI shows the results for IPv6 virtual routing tables.

Our compact binary trie can be used with path compression [18], in which each internal node of a trie can be removed if it has only one child to shorten the path from the root. Table VII demonstrates the memory requirement of all the approaches after path compression is applied. Path compression can not save memory when applied to the leaf pushed trie. Therefore, results for the simple overlaying with leaf pushing are not listed in Table VII.

The results in Table IV show that the separate approach is better than the simple overlaying approach because the performance of the overlaying approach strictly depends on the similarity of individual tries. If the shapes of the tries are different, then the final merged trie has a large overhead. Additionally, whenever leaf pushing is applied, the empty fields that cause the overhead are filled by redundant next hop information. In our approach, we initially prepend Virtual ID to each prefix making all the virtual tables disjoint. Since a single binary trie is built using only the active part (AP) of the prepended prefixes, the performance of our algorithm is much less sensitive to the lack of similarity among the tables when compared with the simple overlaying. Our hybrid data structure achieves a $1.32\times$ memory reduction over the separate

TABLE VI
NUMBER OF NODES (K)/ TOTAL MEMORY REQUIREMENT (MBIT) IN IPV6

No. of FIB	1	2	3	4	5	6	7	8	9	10
Separate	2210/108	4590/224	6797/332	8913/435	11174/546	13537/661	15833/773	18044/881	20338/993	22630/1105
Simple overlay 1	2210/108	4572/259	6751/422	8791/618	11024/840	13243/1086	15502/1362	17533/1678	19560/1987	21665/2327
Simple overlay 2	4224/202	8753/462	12917/719	16801/1017	21072/1338	25315/1681	29637/2055	33504/2487	37362/2882	41377/3313
Hybrid	2004/98	4195/218	6204/321	8117/444	10150/555	12312/673	14409/816	16420/946	18487/1065	20578/1186

TABLE VII
NUMBER OF NODES (K)/ TOTAL MEMORY REQUIREMENT (MBIT) IN IPV6 WITH PATH COMPRESSION

No. of FIB	1	2	3	4	5	6	7	8	9	10
Separate	200/14	418/29	617/43	813/57	1015/71	1232/87	1440/101	1639/115	1847/130	2054/144
Simple overlay 1	200/14	418/33	617/53	810/74	1012/101	1222/129	1429/159	1621/190	1814/223	2011/259
Hybrid	196/15	399/32	595/49	790/66	988/83	1190/102	1388/119	1584/138	1781/155	1978/172

TABLE VIII
REDUNDANCY (%)

No. of FIB	1	2	3	4	5	6	7	8	9	10
$\sigma = \Delta/S_2$	0	44.03	61.01	69.99	74.77	78.45	80.19	80.85	81.30	82.42

approach, and $2\times$ reduction over the overlaying approach for IPv4.

The performance of our approach in IPv6 is the same with separate approach but achieves $2\times$ memory reduction over the simple overlaying without leaf pushing, and $3\times$ reduction over the leaf pushing case. Our proposed algorithm with path compression achieves approximately $7\times$ memory reduction compared with the case without path compression.

Table VIII presents the σ, which is the ratio of the data redundancy (Δ) to the total number of next hops in merged trie(Δ/S_2). The results show that the redundancy increases with the increasing number of virtual tables. For ten virtual tables, about 82% of the next hop fields are filled by redundant information.

VI. CONCLUSION

In this paper, we proposed a hybrid data structure to merge multiple virtual routing tables. The proposed structure achieves substantial memory saving without the need for backtracking. Our approach reduces the size of each entry in the data structure without using leaf-pushing. Furthermore, it eliminates the shared next hop information structure, and simplifies table updates. We also designed and implemented a high-throughput, linear-pipeline architecture to support the proposed data structure on FPGAs. The major drawback of this approach is the need of the secondary search structure. In the future work, we plan to (1) reduce the size of secondary storage, (2) allow different number of prefixes per node for each level of the trie, and (3) apply clustering techniques to further improve the memory efficiency of the overall design.

REFERENCES

[1] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 41, no. 3, pp. 712–727, 2006.

[2] A. J. Mcauley and P. Francis, "Fast routing table lookup using cams," in *Proc. INFOCOM '93*, pp. 1382–1391, 1993.

[3] W. Jiang, Q. Wang, and V. Prasanna, "Beyond tcams: An sram-based parallel multi-pipeline architecture for terabit ip lookup," in *Proc. INFOCOM '08*, pp. 1786 –1794, 2008.

[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*, no. 2, pp. 123–133, 2005.

[5] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "Camp: fast and efficient ip lookup architecture," in *Proc. ANCS '06*, pp. 51–60, 2006.

[6] O. Erdem and C. F. Bazlamaçi, "Array design for trie-based ip lookup," *Comm. Letters.*, vol. 14, pp. 773–775, August 2010.

[7] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Comput. Netw.*, vol. 54, pp. 862–876, April 2010.

[8] F. Baumgartner, T. Braun, E. Kurt, and A. Weyland, "Virtual routers: a tool for networking research and education," *SIGCOMM Comput. Commun. Rev.*, vol. 33, pp. 127–135, July 2003.

[9] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Building scalable virtual routers with trie braiding," in *Proc. INFOCOM'10*, INFOCOM'10, pp. 1442–1450, 2010.

[10] J. Fu and J. Rexford, "Efficient ip-address lookup with a shared forwarding table for multiple virtual routers," in *Proc. CoNEXT '08*, CoNEXT '08, pp. 21:1–21:12, 2008.

[11] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, pp. 1–40, February 1999.

[12] D. T. Jonathan, D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable ip lookup for internet routers," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 522–534, 2000.

[13] 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*, INFOCOM '09, pp. 2509 –2517, 2009.

[14] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier, "Scalable network virtualization using fpgas," in *Proc. FPGA '10*, FPGA '10, pp. 219–228, 2010.

[15] RIS RAW DATA. Online. http://data.ris.ripe.net.

[16] H. Le and V. K. Prasanna, "Scalable high throughput and power efficient ip-lookup on fpga," in *Proc. FCCM '09*, FCCM '09, 2009.

[17] T. Ganegedara, W. Jiang, and V. Prasanna, "Frug: A benchmark for packet forwarding in future networks," in *Proc. IPCCC '10*, IPCCC '10, 2010.

[18] D. R. Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *Journal ACM*, 15:514–534, October 1968.