Memory-Efficient and Scalable Virtual Routers Using FPGA

Hoang Le, Thilan Ganegedara and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
{hoangle,ganegeda,prasanna}@usc.edu

ABSTRACT

Router virtualization has recently gained much interest in the research community. 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. Limited on-chip memory in FPGA leads to the need for memory-efficient merging algorithms. On the other hand, due to high frequency of combined updates from all the virtual routers, the merging algorithms must be highly efficient. Hence, the router must support quick updates. In this paper, we propose a simple merging algorithm whose performance is not sensitive to the number of routing tables considered. The performance solely depends on the total number of prefixes. We also propose a novel *scalable, high-throughput* linear pipeline architecture for IP-lookup that supports *large virtual routing tables* and *quick non-blocking update*. Using a state-of-the-art Field Programmable Gate Array (FPGA) along with external SRAM, the proposed architecture can support up to 16M IPv4 and 880K IPv6 prefixes. Our implementation shows a sustained throughput of 400 million lookups per second, even when external SRAM is used.

Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Internetworking routers

General Terms

Algorithms, Design

Keywords

FPGA, IP Lookup, Virutal Routers, Pipeline

1. INTRODUCTION

1.1 IP Lookup

IP packet forwarding, or simply, IP-lookup, is a classic problem. In computer networking, a routing table is an electronic table or database that is stored in a router or a networked computer. The 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 a routing table. To determine the outgoing port for a given address, the longest matching prefix among all the prefixes needs to be determined. Routing tables often contain a default route, in case matches with all other entries fail.

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 [8]. These disadvantages are more pronounced when we move from IPv4 to IPv6, as the address length increases from 32 to 128 bits.

SRAM-based solutions, on the other hand, require multiple cycles to process a packet. The common data structure in algorithmic solutions for performing LPM is some form of a tree. Pipelining is used to improve throughput. These approaches, however, suffer from either inefficient memory utilization (in trie-based solutions), lack of support for quick updates (in tree-based solutions), or both. Additionally, *overlap* in prefix ranges prevent IP lookup from employing tree search algorithms without modification. Limited on-chip memory also becomes an important factor in supporting large routing tables. Due to these constraints, state-of-the-art SRAM-based designs do not scale to simultaneously support the increased table size and quick update requirement.

1.2 Network Virtualization

Network virtualization is a technique to consolidate multiple networking devices onto a single hardware platform. The main goal of virtualization is to make efficient use of the networking resources. It can be thought of as an abstraction of

Table 1: Sample virtual routing tables (maximum prefix length = 8)

Virtual table 1 (VID = 0)			Virtual table 2 (VID = 1)		
	Prefix	Next Hop		Prefix	Next Hop
P_{11}	0*	1	P_{21}	10*	2
P_{12}	01*	3	P_{22}	101*	3
P_{13}	101*	2	P_{23}	111*	4
P_{14}	111*	4	P_{24}	100*	1
P_{15}	0010*	5	P_{25}	1*	5
P_{16}	00*	3	P_{26}	0*	2

the network functionality away from the underlying physical network.

Device consolidation of routers at the physical layer results in consolidation up to the network layer where IP lookup is performed. Since router hardware is virtualized in such an environment, they are called virtualized routers, or simply virtual routers. Serving multiple virtual networks on the same routing hardware is a challenge for the research community.

Several ideas have been proposed to realize router virtualization on a single hardware networking platform [2, 6, 12, 18, 22]. These ideas mainly focus on how to implement a memory-efficient virtualized router, while guaranteeing the basic requirements of network virtualization. These requirements are: fair resource usage, fault isolation, and security.

Two existing approaches in the literature for network virtualization are *Separated* and *Merged* approach. The *separated* approach instantiates multiple virutal router instances on the same hardware routing platform by partitioning router resources. In contrast, the *merged* approach combines or merges all the routing tables into a single table to serve packets from different virtual networks.

These approaches have their own advantages and drawbacks. For example, the separated approach requires much more router hardware resources than the merged one, but it provides perfect traffic isolation, security, and avoids single point of failure. On the other hand, the merged approach requires much less hardware resources; yet, the traffic and fault isolation are not as strong as those of the separated approach.

The key issues to be addressed in designing an architecture for IP lookup in virtual routers are: (1) quick update, (2) high throughput, and (3) scalability with respect to the size of the virtual routing table. To address these challenges, we propose and implement a *scalable high-throughput, SRAM-based* linear pipeline architecture for IP-lookup in virtual router that supports *quick* update. This paper makes the following contributions:

- A simple merging algorithm results in the total amount of required memory to be *less sensitive* to the number of routing tables, but to the total number of virtual prefixes (Section 4).
- A tree-based architecture for IP lookup in virtual router that achieves high throughput and supports quick update (Section 5).
- Use of external SRAMs to support large virtual routing table of up to 16M virtual prefixes (Section 6).
- A scalable design with linear storage complexity and resource requirements (Section 5.4).
- A sustained throughput of 400 million lookups per second, even when external SRAM is used (Section 7).

The rest of the paper is organized as follows. Section 2 covers the background and related work. Section 3 gives a formal definition of the problem. Section 4 introduces the proposed merging and IP lookup algorithm. Section 5 and 6 describe the architecture and its implementation. Section 7 presents implementation results. Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 IP Lookup in Virtual Router

Two sample routing tables with the maximum prefix length of 8 are illustrated in Table 1. These sample tables will be used throughout the paper. Note that the next hop values need not be in any particular order. In these routing tables, binary prefix P_{13} (101∗) matches all destination addresses that are destined to virtual router 1 and begin with 101. Similarly, prefix P_{24} matches all destination addresses that are destined to virtual router 2 and begin with 100. The 8-bit destination address **IP**= 10100000 from virtual network 2 is matched by the prefixes P_{21}, P_{22}, and P_{25} in virtual table 2. Since $|P_{21}| = 2$, $|P_{22}| = 3$, $|P_{25}| = 1$, P_{22} 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 3 is returned.

2.2 Related Work

Network virtualization is a broad research area. It has proved itself to be a powerful scheme to support the coexistence of heterogenous networks on the same physical networking substrate. From research standpoint, this has been a great opportunity for the networking researchers to test their algorithms and/or protocols in production networks. This was not possible in the traditional IP networks due to the protocol rigidity of networking devices. However, this problem can be overcome by using mechanisms like Multi-Protocol Label Switching (MPLS) [3]. The Layer-3 routing information is encapsulated using the MPLS header. The packet switching is done based on the label rather than using the destination IP. Hence, packet routing is independent of the routing information in the virtual network [11].

Although much work has been done for IP lookup ([21, 9, 20]), only a few have been targeted for virtual routers. In the industry, Juniper has introduced router virtualization in their JCS1200 router [2]. This router can support up to 192 virtual networks, where each virtual network is severed by a logical router (i.e. separated approach). Cisco has also proposed solutions for router virtualization in provider networks, using hardware and software isolated virtual routers [6]. They give a comprehensive comparison of the pros and cons of each approach, and the applicability of each for different networking environments.

Chowdhry et. al [11] has surveyed the opportunities and challenges associated with network virtualization from the networking and implementation perspectives. In [10], the authors evaluated the issues and challenges in the virtualized commercial operator environments.

Several solutions for router virtualization have been recently proposed by the research community. In [12, 18], the authors employed the merged approach, whereas in [22], the separated approach was used. In [22], the authors implemented the virtualized router on a hardware-software hybrid platform. They support up to four hardware virtual routers on a NetFPGA [4] platform. The rest of the virtual routers, which were implemented using the Click modular router[15], resided on a general purpose computer, and are virtualized using OpenVZ. The authors described an interesting idea of network adaptive routers. In their idea, a router is dynamically moved on to the hardware platform from the software platform, when the network traffic level for a particular virtual network increases beyond a threshold. The throughput of the implementation is relatively low ($\sim$ 100 Mbps), compared with the rates at which network links operate in production networks (multi Gbps). Moreover, due to extensive hardware utilization, the NetFPGA hardware cannot support more than 4 virtual router instances.

Fu et. al in [12] 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 [18] 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. They introduced a *braiding bit* at each node for each routing table to identify the direction of traversal along the trie. The number of nodes in the final trie is used as a metric to evaluate the performance of their algorithm. Even though the number of nodes are minimized, the memory requirement at each non-leaf node becomes $O(m+2P)$, where m is the number of virtual routers and P is the size of a pointer in bits. It is clear that the memory consumption grows linearly with m. Therefore, reduction in the total number of nodes does not necessarily lead to the reduction of the overall memory consumption. However, the authors claimed that they were able to store 16 routing tables with a total of 290K prefixes using only 36 Mbits. This scheme performs well only when the routing tables have different structure.

3. PROBLEM DEFINITION

The problem of merging and searching multiple virtual routing tables is defined as follows. Given m virtual routing tables $R_i, i = 0, .., m-1$, each having N_i number of prefixes, find (1) an algorithm that can efficiently merge these virtual tables to minimize the total memory requirement and (2) a search data structure that can support high throughput and quick update.

4. IP LOOKUP ALGORITHM FOR VIRTUAL ROUTER

4.1 Definitions

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***.

Definition Two distinct prefixes are said to be ***overlapped*** if and only if one is a proper prefix of the other.

Definition A set of prefixes is considered ***disjoint*** if and only if any 2 prefixes in the set do not overlap with each other.

Definition Prefixes that are at the leaves of the trie are called ***leaf prefixes***. Otherwise, they are called ***internal prefixes***.

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.

Definition Each virtual router instance has a routing table, which is called ***virtual routing table***, or simply ***virtual table***. Additionally, each virtual instance is associated with a ***virtual network ID***, or simply ***virtual ID***.

Definition ***Prefix update*** can either be (1) modification to the existing prefixes, (2) insertion of new prefixes, or (3) deletion of existing prefixes. It can also be referred as ***incremental update***.

4.2 Set-Bounded Leaf-Pushing Algorithm

Tree search algorithm is a good choice for IP forwarding engine as the lookup latency does not depend on the length of the prefix, but on the number of prefixes in the routing table. In case of tree search, the latency is proportional to log of the number of prefixes. Note that the trie-based approaches can also reduce the latency by using multi-bit trie. However, this reduction comes at the cost of memory explosion, and thus, results in a very *poor* memory efficiency. Furthermore, our experiment shows that the number of nodes in the trie drastically expands as the prefix length increases from 32 bits to 128 bits. Path compression techniques ([16, 17]) work well to contain the number of trie nodes. Nonetheless, they increase the computational complexity at the nodes and reduce the look-up performance.

In order to use tree search algorithms, the given set of prefixes needs to be processed to eliminate the overlap between prefixes. This elimination process results in a set (or sets) of *disjoint* prefixes. There are many approaches proposed to eliminate overlap, such as leaf-pushing, prefix expansion [20], prefix partitioning [14], and prefix merging [9]. Among these approaches, leaf-pushing is the most popular one due to its simplicity. However, as mentioned before, these approaches either increase the number of prefixes, the number of sets of disjoint prefixes, or both. What we are interested in is an algorithm that can bound the number of sets (to 2 in our work), while keeping the increase in number of prefixes minimal.

In the leaf pushing approach, the initial step is to build a trie from the given routing table. Leaf pushing first grows the trie to a full tree (i.e. all the non-leaf nodes have two child nodes), and then pushes all the prefixes to the leaf nodes. All the leaf nodes are then collected to build a *search tree*. Since the set of leaf nodes are disjoint, any tree search algorithm can be employed to perform the lookup. While leaf pushing eliminates overlap between prefixes, it has the negative effect of expanding the size of the prefix table. For all publicly available routing tables, the prefix tables expand about 1.6 times after leaf pushing [19].

From our analysis with real-life routing tables, we observe that about 90% of the prefixes in these public tables are *leaf prefixes* (Section 7.1). Moreover, if these leaves are removed and the non-prefix leaves are trimmed off the trie, the resulting trie still has the same properties as the original trie. This observation gives us a way to control the number of prefixes by partitioning the original routing table into *exactly* 2 sets of prefixes. Set 1 consists of all the leaf prefixes of the original trie, which are *disjoint*. Set 2 includes all the leaf-pushed prefixes of the trimmed trie, which are also *disjoint*. The process of partitioning the prefix table into 2 sets of disjoint prefixes is called "*set-bounded leaf-pushing*" (SBLP). Due to the disjoint prefixes, at most *one* match can be found in each set for each incoming IP address. In case each set has their own match, the longer match is chosen as the final matched prefix, which is from Set 1 as it is longer.

There are 3 steps involved in the algorithm:

1. Move the leaves of the trie into Set 1
2. Trim the leaf-removed trie
3. Leaf-push the resulting trie and move the leaf-pushed leaves into Set 2

The following notations are used in the analysis:

- N: the number of prefixes in the given routing table
- N': the number of prefixes after processing
- k: the leaf-pushing expansion coefficient, defined as the ratio of the total number of prefixes after and before leaf-pushing
- l: the leaf percentage, defined as the ratio of the number of leaf-prefixes and the total number of prefixes

The number of prefixes in the two sets can be expressed as:

$$N_{S_1} = lN \quad (1)$$

$$N_{S_2} = kN(1-l) \quad (2)$$

$$N' = N_{S_1} + N_{S_2} = kN + lN(1-k) = N(k+l-kl) \quad (3)$$

Our analysis on IPv4 routing tables showed that $l = 0.9$ and $k = 1.6$ for IPv4, hence, $N' = 1.06N$. Therefore, by using the set-bounded leaf-pushing algorithm, the total number of prefixes in 2 sets is just slightly larger than the total number of original prefixes (by 6%). For IPv6 routing tables, the result is even better, with $N' \approx N$. The experimental results of the real and synthetic routing tables agree with our analysis and are presented in Section 7.

Complexity: The first step of the algorithm is to build a trie from a given routing table consisting of N prefixes. This step has a complexity of $O(N)$. The leaf-prefixes are then moved to Set 1. This step has a complexity of $O(N)$. The trie is then trimmed and leaf-pushed, with a complexity of $O(N)$. In the final step, the leaf-prefixes are moved to Set 2, also with a complexity of $O(N)$. Since all the steps are sequential, the overall complexity of the set-bounded leaf-pushing algorithm is $O(N)$.

4.3 2-3 Tree

We propose a memory efficient data structure based on a 2-3 tree [1], which is a type of B-tree with order of 3. A 2-3 tree is a balanced search tree that has the following properties:

1. There are three different types of nodes: a leaf node, a 2-node and a 3-node (Figure 1).
2. A leaf node contains one or two data fields.

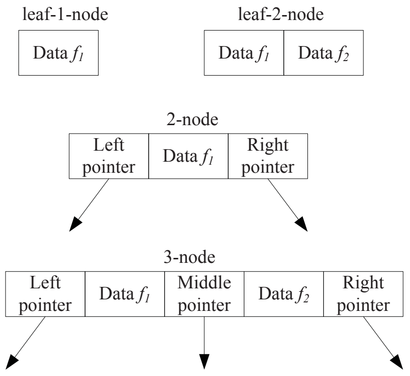


Figure 1: Different types of nodes of a 2-3 tree

3. A 2-node is the same as a binary search tree node, which has one data field and references to two children.
4. A 3-node contains two data fields, ordered so that the first is less than the second, and references to three children. One child contains values less than the first data field, another child contains values between the two data fields, and the last child contains values greater than the second data field.
5. All of the leaves are at the lowest level.

A 2-3 tree with n nodes never has a height greater than $\log_2(n+1)$. Traversing in 2-3 tree can be performed in a similar way as in a binary search tree: compare and move down the correct link until a leaf node is reached. The search algorithm in 2-3 tree runs in $O(\log_2 n)$ time, where n is the total number of elements in the tree.

The 2-3 tree is chosen as our search data structure over other data structures due to its advantages: (1) a 2-3 tree is always balanced, (2) search in a 2-3 tree is performed the same way as in a binary search tree, and (3) the number of nodes in each level of the tree grows exponentially as we move further from the root. Moreover, insertion and deletion in a 2-3 tree take $O(\log n)$ steps. It will be clearer in the later sections that this complexity can be brought down to $O(1)$ by utilizing the parallelism in hardware.

4.4 Merging Algorithm

We argue that any merging algorithm designed for virtual router need to have the following characteristics:

- The algorithms should be simple and have fast execution time. The main reason is the frequency of updates. In a single routing table, update does not tend to occur frequently. However, when multiple routing tables are considered, the updates are aggregated; therefore, the update rate increases dramatically.
- The algorithms should not be routing-table sensitive. For different routing tables, the total required memory should not have a large variance as it is difficult to allocate memory at design time. Another reason is that new prefixes can break the optimal point of the algorithm and adversely affect the memory footprint.
- The algorithms should not depend on the number of virtual routing tables, but on the total number of prefixes.

Table 2: Merged virtual routing table

	Virtual prefix	Length	Next Hop
P_{11}	**00***	2	1
P_{12}	**0**01*	3	3
P_{13}	**0**101*	4	2
P_{14}	**0**111*	4	4
P_{15}	**0**0010*	5	5
P_{16}	**0**00*	3	3
P_{21}	**1**10*	3	2
P_{22}	**1**101*	4	3
P_{23}	**1**111*	4	4
P_{24}	**1**100*	4	1
P_{25}	**1**1*	2	5
P_{26}	**1**0*	2	2

We propose a simple algorithm for merging m virtual routing tables $R_i, i = 0, 1, .., m-1$. All the virtual routing tables are combined into a single virtual routing table R_{all}. Each prefix in table R_{all} is a structure consisting of 3 values: (1) virtual prefix value, (2) prefix length, and (3) next hop information. The *virtual prefix* is the concatenation of the virtual ID and the prefix value of each prefix in that order. Each prefix structure is identified by this *virtual prefix*. The maximum length of the virtual prefix is $(L + L_{VID})$, where L_{VID} is the length of the virtual ID and L is the maximum prefix length (32 for IPv4 and 128 for IPv6). The terms *prefix* and *prefix structure* are used interchangeably in this paper.

A *single* binary trie is built for *all* the virtual prefixes in R_{all}. We then apply the *set-bounded leaf-pushing* algorithm (Section 4.2) to generate 2 sets of *disjoint* prefixes, namely S_1 and S_2. Each prefix is padded with 0 up to the maximum length. The number of padded 0s is $(L - L_P)$, where L_P is the length of the prefix. The padded prefix is then attached with a prefix length. Note that each prefix in 2 sets can appear in more than one virtual table. However, these duplicated prefixes have different virtual ID associated with the virtual table that they belong to.

We use an example to illustrate the idea. Consider 2 sample virtual routing tables as depicted in Table 1. The merged routing table is shown in Table 2. Using this merged table, a single binary trie is built and its leaf-prefixes are moved to Set S_1. The leaf-removed trie is trimmed and then leaf-pushed. All the leaf-prefixes are collected in Set S_2. These 3 steps are shown in Figure 2, 3 and 4, respectively. All the prefixes in each set are padded with 0 to form the virtual padded prefixes, which are shown in Table 3.

Complexity: All the prefixes are prepended with its virutal ID in the first step with a complexity of $O(N)$, where N is the total number of prefixes. We then apply the set-bounded leaf-pushing algorithm to the combined virtual routing table. The complexity of this step is $O(N)$ (Section 4.2). Therefore, the complexity of the merging algorithm is $O(N)$.

The merging algorithm can also be parallelized to reduce the execution time. Each virtual routing table can be processed independently. For each virtual table i, 2 sets of disjoint prefixes are generated, S_1^i, S_2^i. The final step is to simply combine all the corresponding sets into a single set:

$$S_1 = \sum_{i=0}^{m-1} S_1^i, S_2 = \sum_{i=0}^{m-1} S_2^i$$

4.5 IP Lookup Algorithm for Virtual Router

Using our merging algorithm described above, m virtual

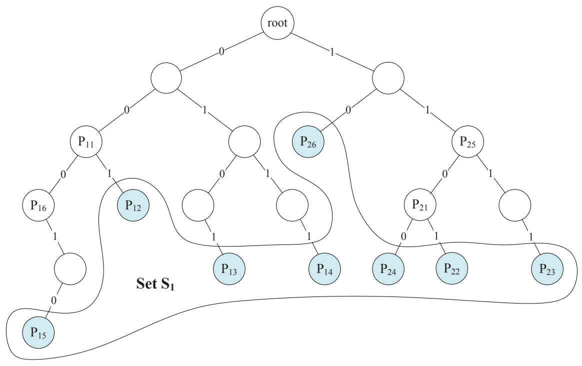


Figure 2: Trie is built and leaves are moved to Set S_1

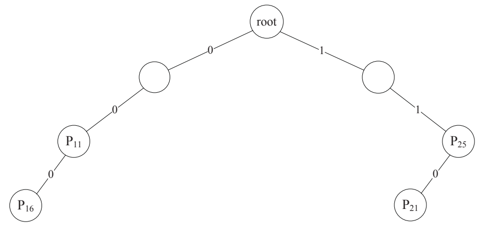


Figure 3: Trie is trimmed

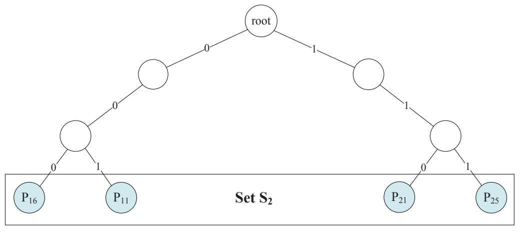


Figure 4: Trie is leaf-pushed and leaves are moved to Set S_2

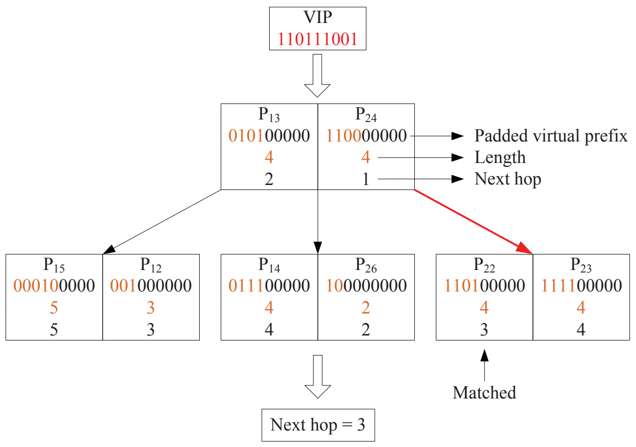


Figure 5: Corresponding 2-3 tree of Set S_1

Table 3: Padded virtual prefixes in 2 sets

Set S_1			
	Padded virtual prefix	Length	Next Hop
P_{12}	001000000	3	3
P_{13}	010100000	4	2
P_{14}	011100000	4	4
P_{15}	000100000	5	5
P_{22}	110100000	4	3
P_{23}	111100000	4	4
P_{24}	110000000	4	1
P_{26}	100000000	2	2

Set S_2			
P_{11}	001000000	3	1
P_{16}	000000000	3	3
P_{21}	110000000	3	2
P_{25}	111000000	3	5

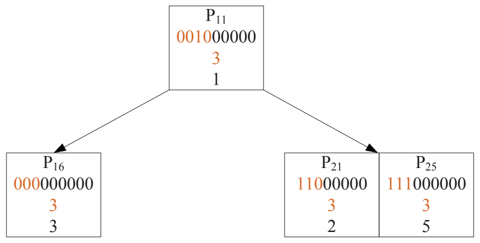


Figure 6: Corresponding 2-3 tree of Set S_2

routing tables, $R_0, .., R_{m-1}$, are merged and processed to generate 2 sets of prefixes, S_1 and S_2. For each set we build a 2-3 tree using the padded virtual prefix as the key ($S_1 \mapsto T_1, S_2 \mapsto T_2$). Note that since all the prefixes in each set are disjoint, any tree search algorithm can be used to search for the matching prefix. Figure 5 and 6 show the corresponding 2-3 trees of the 2 sample sets in Table 3. After building the 2 trees, IP-lookup operation can be performed as follows. The IP address and virtual ID are extracted from the incoming packet. The virtual ID is appended with the IP address to form the *virtual IP address* (VIP). The VIP is searched in both T_1 and T_2 to find a possible match.

In each step of the comparison, the virtual IP address is compared against the virtual prefix of the current node. There are 2 types of comparison. The first type is to determine if the virtual prefix matches the virtual IP address. The second comparison is to determine the direction of traversal. If the node is a 2-node, which has one 1 data field f_1, then the direction is *left* (if **VIP**$\leqslant f_1$), or *right* otherwise. If the node is a 3-node, which has 2 data fields f_1 and f_2 ($f_1 < f_2$), then the direction can be *left* (if **VIP**$\leqslant f_1$), *middle* (if $f_1 <$**VIP**$\leqslant f_2$), or *right* otherwise.

The search results from both T_1 and T_2 are combined to give the matching result. Note that T_1 has higher priority than T_2 due to the longer matched prefix. Hence, if there are matches in both trees, the one from tree T_1 is returned as the final result. For instance, assume that a packet with a destination address of IP = 10111001 with a virtual ID of 1 arrives. The virtual IP address (VIP) of this packet is 110111001. At the root of T_1, VIP is compared with node values 010100000 and 110000000, yielding no match and a "greater" result. Thus, the packet traverses to the right branch. The comparison with the prefix in this node results in a match with $matched\ prefix =$ 110100000, $matched\ length = 4, next\ hop = 3$, which is the final outcome.

Table 4: List of notations used in the paper

Notation	Meaning
m	Number of virtual routing tables
R_i	Virtual routing table i
N_i	The number of prefixes in R_i
N	The total number of prefixes in all the virtual routing tables
L	Maximum prefix length (32 for IPv4, 128 for IPv6)
L_P	Prefix length = $\lceil \log L \rceil$
L_{VID}	Length of the virtual ID = $\lceil \log m \rceil$
L_{NHI}	Length of the next hop information
M	Size of the required memory
L_{Ptr}	Address length
$\lvert S_i \rvert$	Number of prefixes in set S_i

4.6 Memory Requirement

We now analyze the merging and searching algorithm. Recall that there are 2 types of nodes in a 2-3 tree: 2-node and 3-node. A 2-node has one data field and 2 pointers, or 2 pointers per data field. The 3-node has 2 data fields and 3 pointers, or 1.5 pointers per data field. Note that each data field holds one prefix. It is obvious that the average number of pointers per prefix is between 1.5 and 2. Therefore, for the sake of simplicity, we will assume that the number of pointers per prefix is 2 in our calculation. The list of notations used in the analysis are shown in Table 4.

The memory requirement M_1 and M_2 of the first and second search trees are:

$$M_1 = |S_1|(L + L_P + L_{VID} + L_{NHI} + 2L_{Ptr_1}) \tag{4}$$
$$M_2 = |S_2|(L + L_P + L_{VID} + L_{NHI} + 2L_{Ptr_2}) \tag{5}$$

Where

$$L_P = \log L$$
$$L_{Ptr_1} = \log |S_1|$$
$$L_{Ptr_2} = \log |S_2|$$

The total memory requirement M is:

$$\begin{aligned} M &= M_1 + M_2 \\ &= (|S_1| + |S_2|)(L + L_P + L_{VID} + L_{NHI}) \\ &+ 2|S_1|L_{Ptr_1} + 2|S_2|L_{Ptr_2} \end{aligned} \tag{6}$$

In reality, nodes in the last level of the tree need not contain any pointer as they have no children. Additionally, the number of leaf-nodes is at least half the total number of nodes of the tree. Therefore, the total memory requirement can be rewritten as:

$$\begin{aligned} M &= (|S_1| + |S_2|)(L + L_P + \log m + L_{NHI}) \\ &+ |S_1|L_{Ptr_1} + |S_2|L_{Ptr_2} \end{aligned} \tag{7}$$

From the formula, we can observe that the total memory requirement M is not sensitive to the number of routing tables m. In fact, M is proportional to $L_{VID} = \lceil \log m \rceil$. As we learned in Section 3, for IPv4, $|S_1| = 0.9N, |S_2| = 0.16N, |S_1| + |S_2| = 1.06N$, where N is the total number of prefixes in all the routing tables. Thus, the storage complexity is $O(N \times \log m)$. If we fix the number of virtual routers, then M grows linearly with N.

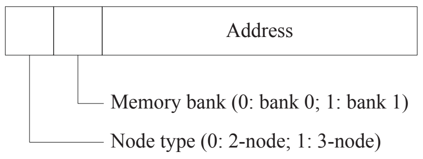

Address
Memory bank (0: bank 0; 1: bank 1)
Node type (0: 2-node; 1: 3-node)

Figure 7: Content of the pipeline forwarded address

5. ARCHITECTURE

5.1 Overall Architecture

Pipelining is used to produce one lookup operation per clock cycle, and thus increase the throughput. The number of pipeline stages is determined by the height of the 2-3 tree. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table).

Figure 9(a) describes the overall architecture of the proposed IP lookup engine for virtual routers. There are 2 pipelines (one for each set of prefixes). 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 all branches. The searches are performed in parallel in all the pipelines. The results are fed through a priority resolver to select the next hop index of the longest matched prefix. The variation in the number of stages in these pipelines results in latency mismatch. The delay block is appended to the shorter pipeline to match with the latency of the longer pipeline.

The block diagram of the basic pipeline and a single stage are shown in Figure 9(b). 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 virtual IP addresses can be input every clock cycle. In each pipeline stage, there are 3 data fields forwarded from the previous stage: (1) *Virtual IP address*, (2) *next hop*, and (3) *memory address*. The memory address, whose content is shown in Figure 7, is used to retrieve the node stored on the local memory. The node value is compared with the input virtual IP address to determine the match status and the direction of traversal. 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. In this case, search in subsequent stages is unnecessary as all the prefixes with a set are disjoint. Hence, those subsequent stages can be turned off to save power consumption. Furthermore, if a match has been found in Set 1 (or pipeline 1), search in Set 2 (or pipeline 2) can also be terminated as Set 1 has a higher priority compared with Set 2.

5.2 Memory Management

The major difficulty in efficiently realizing a 2-3 tree on hardware is the difference in the size of the 2-node and 3-node. The space allocated for a 2-node cannot be reused later by a 3-node. The available memory management is also more complicated. To overcome this problem, a 3-node is divided into two 2-nodes (Figure 8). The left pointer of the first node points to the left child. The right pointer of the first node and the left pointer of the second node both point to the middle child. Finally, the right pointer of the second node points to the right child. Although a pointer is redundant, the benefit is justifiable. It creates *uniformity* in the size of the nodes, and simplifies the memory management. Additionally, this node-splitting scheme allows us to precisely estimate the amount of required memory, as each prefix is stored in one *effective* 2-node.

Two memory banks are used to support the node-splitting scheme. A 2-node can be placed in any bank, while a 3-node spans over 2 banks. Note that when a 3-node is split into two 2-nodes, each of them must be placed in the same memory location in each bank. This placement allows us to use a single address to access both nodes.

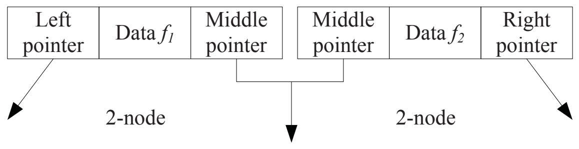
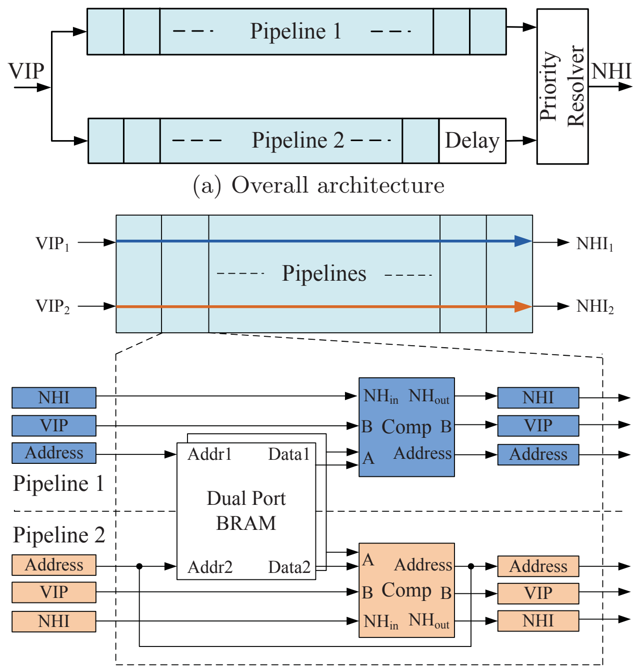


Figure 8: The modified 3-node



(a) Overall architecture

(b) A basic pipeline stage

Figure 9: Block diagram of the proposed IP-lookup architecture (NIH-Next hop information; VIP-Virtual IP address)

5.3 Virtual Routing Table Update

A virtual routing table update can be any of 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.

The first type of update can easily be done by first finding the correct node that contains the prefix. Once the node is found, the next hop information is updated. However, the second and third type of updates require extra work. The

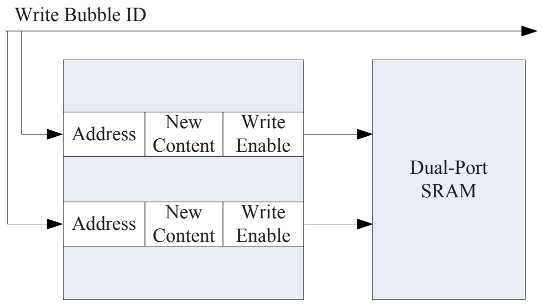


Figure 10: Route update using dual write-bubbles

insertion and deletion cause changes in *at most two nodes* at each level of the tree, in the worst case [24]. For the insertion, the first task is to find the (non-leaf) node that will be the parent p of the newly inserted node n. There are 2 cases: (1) p has only 2 children and (2) p has 3 children. In the first case, n is inserted as the appropriate child of p; p becomes a 3-node. A new 2-node needs to be added for p, and a new 2-node is allocated for n in the next level. In the second case, n is still inserted as the appropriate child of p. However, p has 4 children. An internal node m is created with p's two rightmost children. m is then added as the appropriate new child of p's parent (i.e., add m just to the right of p). If p's parent had only 2 children, insertion is done. Otherwise, new nodes are created recursively up the tree. If the root is given 4 children, then a new node m is created as above, and a new root node is created with p and m as its children. The deletion process is similar, with merging instead of splitting.

Once all the changes in each level of the tree are precomputed, the update operation is performed starting from the root of the tree. Since at most two nodes are modified in each level of the tree, this update can easily be done by inserting *only one* write bubble, as shown in Figure 10. There is one dual-ported write bubble table (WBT) in each stage. Each table consists of at least 2 entries. Each entry composes of (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. The new content of the memory is computed offline in $O(\log_2 N)$ time, where N is the number of nodes. However, it is not necessary to download a new forwarding table for every route update. Route updates can be frequent, but routing protocols need time, in the order of minutes, to converge. Thus, the offline update computation can also be done at the control plane.

When a prefix update is initiated, the memory content of the write bubble table in each stage is updated, and a write bubble is inserted into the pipeline. When it arrives at the stage prior to the stage to be updated, the write bubble uses the new content from the WBT to update the memory location. At most 2 nodes can be simultaneously updated at each stage, using the dual-ported memory. When the bubble moves to the next level (stage), the tree up to that level is fully updated, and the subsequent lookup can be performed properly. This updating mechanism supports non-blocking prefix updates at system speed.

5.4 Scalability

The use of the 2-3 tree leads to a linear storage complexity in our design, as each *effective* 2-node contains exactly one virtual prefix. The height of a 2-3 tree is at most $(1 + \lfloor \log_2 N \rfloor)$, where N is the number of nodes. Since each level of the tree is mapped to a pipeline stage, the height of the tree determines the number of stages. Our proposed architecture is simple, and is expected to utilize a small amount of logic resource. Hence, the major constraint that dictates the number of pipeline stages, and in turn the size of the supported virtual routing table, is the amount of on-chip memory. As mentioned before, the per-level required memory size grows exponentially as we go from one level to the next of the tree. Therefore, we can move the last few stages onto external SRAMs. However, due to the limit on the number of I/O pins of FPGA devices, we can fit only a certain number of stages on SRAM, as described in detail in Section 6.

The scalability of our architecture relies on the close relationship between the size of the virtual routing tables and the number of required pipeline stages. As the number of prefixes increases, extra pipeline stages are needed. To avoid reprogramming the FPGA, we should allocate the maximum possible number of pipeline stages.

6. IMPLEMENTATION

As analyzed in Section 4.6, the total required memory is:

$$\begin{aligned} M = {} & (|S_1| + |S_2|)(L + L_P + L_{VID} + L_{NHI}) \\ & + |S_1|L_{Ptr_1} + |S_2|L_{Ptr_2} \end{aligned}$$

For the sake of simplicity, let $L_{Ptr_1} = L_{Ptr_2} = L_{Ptr}$ and $(|S_1| + |S_2|) = N_S$, then

$$M = N_S(L + L_P + L_{VID} + L_{NHI} + L_{Ptr})$$

In IPv4, $L = 32, L_P = 5$, whereas in IPv6, $L = 128, L_P = 7$. We assign $L_{VID} = 5$ to support up to 32 virtual routing tables, $L_{NHI} = 6$ to support up to 64 next hop information, and $L_{Ptr} = 20$. Note that the unit of storage size is in bits unless otherwise stated. The total required memory is:

$$M_{IPv4} = N_S(32 + 5 + 5 + 6 + 20) = 68N_S \qquad (8)$$

$$M_{IPv6} = N_S(128 + 7 + 5 + 6 + 20) = 164N_S \qquad (9)$$

Therefore, a state-of-the-art FPGA device with 36 Mb of on-chip memory (e.g. Xilinx Virtex6) can support up to 530K prefixes (for IPv4), or up to 220K prefixes (for IPv6), *without* using external SRAM. Note that the total number of nodes is approximately equal to the number of supported virtual prefixes (Section 4.6).

In our design, external SRAMs can be used to handle even larger routing tables, by moving the last stages of the pipelines onto external SRAMs. Currently, SRAM is available in $2 - 32$ Mb chips [7], 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 ≈ 180 and ≈ 560 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 6 banks of dual port SRAMs for IPv4, and up to 2 banks for IPv6. For each additional pipeline

Table 5: Number of prefixes and leaf-prefixes of real-life and synthetic routing tables

Core IPv4 routing tables			Edge IPv4 routing tables			Edge IPv6 routing tables		
Table	# prefixes	# leaf prefixes	Table	# prefixes	# leaf prefixes	Table	# prefixes	# leaf prefixes
rrc00	332117	300764 (90.56%)	rrc00_e	95048	71923 (75.67%)	rrc00_e6	95048	95039 (99.99%)
rrc01	324172	294148 (90.74%)	rrc01_e	98390	83445 (84.81%)	rrc01_e6	98390	98388 (99.99%)
rrc03	321617	291885 (90.76%)	rrc03_e	90867	63887 (70.31%)	rrc03_e6	90867	90844 (99.99%)
rrc04	347231	317282 (91.37%)	rrc04_e	95358	72790 (76.33%)	rrc04_e6	95358	95346 (99.99%)
rrc05	322996	293099 (90.74%)	rrc05_e	98305	83188 (84.62%)	rrc05_e6	98305	98297 (99.99%)
rrc06	321577	292260 (90.88%)	rrc06_e	97410	78853 (80.95%)	rrc06_e6	97410	97404 (99.99%)
rrc07	322557	292918 (90.81%)	rrc07_e	95007	71969 (75.75%)	rrc07_e6	95007	95002 (99.99%)
rrc10	319952	290608 (90.83%)	rrc10_e	97376	78685 (80.81%)	rrc10_e6	97376	97369 (99.99%)
rrc11	323668	293810 (90.78%)	rrc11_e	91031	64089 (70.4%)	rrc11_e6	91031	91000 (99.99%)
rrc12	320015	290742 (90.85%)	rrc12_e	97386	78715 (80.83%)	rrc12_e6	97386	97378 (99.99%)
rrc13	335153	303923 (90.68%)	rrc13_e	95163	72338 (76.01%)	rrc13_e6	95163	95153 (99.99%)
rrc14	325797	295613 (90.74%)	rrc14_e	95032	71923 (75.68%)	rrc14_e6	95032	95023 (99.99%)
rrc15	323986	293921 (90.72%)	rrc15_e	91019	64085 (70.41%)	rrc15_e6	91019	91000 (99.99%)
rrc16	328295	297486 (90.62%)	rrc16_e	94780	71801 (75.76%)	rrc16_e6	94780	94767 (99.99%)

stage, the size of the supported routing table at least doubles. Thus, the architecture can support up to 16M prefixes, or 880K prefixes for IPv4 and IPv6, respectively. 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.

Employing DRAM in our design requires some modifications to the architecture. Due to its structural simplicity, DRAM has very high density and very high access bandwidth. The major drawback is its high access latency. Therefore, the design needs to have enough memory requests to DRAM in order to hide this expensive latency. One possible solution is to have multiple pipelines sharing the same DRAM module. However, these pipeline must stall when waiting for the requested data coming back from the DRAM module.

7. PERFORMANCE EVALUATION

7.1 Experimental Setup

Fourteen experimental IPv4 core routing tables were collected from Project - RIS [5] on 06/03/2010. These core routing tables were used to evaluate our algorithm for a real networking environment. However, as mentioned before, router virtualization mainly happens at provider edge networks. Therefore, synthetic IPv4 routing tables generated using FRuG [13] were also used 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 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_e and rrc00_e6) table pair is identical. The number of prefixes and leaf-prefixes of the experimental routing tables are shown in Table 5.

There are 3 groups of routing tables: Group 1 (core IPv4 routing tables), Group 2 (edge IPv4 routing tables), and Group 3 (edge IPv6 routing tables). The tables in each group are merged using our merging algorithm and the results are reported in Table 6. We observe that total number of prefixes in 2 sets is only 1.12×, 1.17×, 1.01× the total number of original prefixes in Group 1, 2, 3, respectively. These results agree with our analysis in the previous sections. With regard to memory footprint, Group 1 and 2 require 40 MB and 11 MB, while Group 3 needs 27 MB of memory. The reported memory requirement is for both on-chip and off-chip memory combined.

Table 6: Merging results of 2 groups of routing tables

	Number of prefixes			
Group	Original	Set 1	Set 2	Total
1	4569133	4148459	962492	5110951(1.12×)
2	1332172	1027691	543135	1570826(1.17×)
3	1332172	1332010	2622	1334632(1.01×)

7.2 Throughput

The proposed architecture was implemented in Verilog, using Synplify Pro 9.6.2 and Xilinx ISE 11.3, with Virtex-6 XC6VSX475T as the target. The implementation showed a maximum frequency of 200 MHz, while utilizing less than 10% of the on-chip logic resource. Using dual-ported memory, the design can support 400 million packets per second (MLPS). This result surpasses the worst-case 150 MLPS required by the standardized 100GbE line cards [19]. Note that our solution can also be implemented on other platforms, such as ASIC and multicore. However, it is out of the scope of this paper, and therefore is not presented.

7.3 Performance Comparison

Four key comparisons were performed with respect to (1) the time complexity of the merging algorithm, (2) memory efficiency, (3) quick-update capability, and (4) throughput. We compare our merging algorithm and lookup architecture with the state-of-the-art designs. These candidates are the trie-overlapping [12] (A1) and the trie braiding [18] (A2) approaches. Note that it is difficult to make a fair and meaningful comparison with these approaches due to the lack of common experimental set of routing tables.

Time complexity: Our merging algorithm has the time complexity of $O(N)$. Scheme A1 has the time complexity of $O(N \log N)$ due to the quick-sort algorithm applied on the forwarding information base. Scheme A2 has the time complexity of $O(N^2)$ due to the linear programming algorithm performed at each node of the trie. Therefore, our merging algorithm has a better time complexity compared with other schemes.

Memory efficiency: When all the routing tables contain the similar set of prefixes, scheme A1 performs extremely well and is effective. However, when the similarity is low, this scheme does not lead to any gains over storing the tries separately. In this case, scheme A2 performs better than A1. It was reported in [18] that for 16 routing tables consisting of 290K prefixes, scheme A2 and A1 require up to 36 Mb (4.5 MB) and 72 Mb (9 MB), respectively. On the other hand, our scheme requires only 15 MB for 14 routing tables consisting of over 1.3M prefixes. Note that our scheme is not sensitive to the number of routing tables, but to the total number of prefixes. Thus, if we need to support 290K prefixes, our scheme utilizes only 2.4 MB of memory.

Quick-update capability: All the 3 schemes support quick incremental update. However, scheme A1 may not be memory and lookup efficient after a large number of updates. Therefore, it is occasionally required to reconstruct the entire lookup data structure for optimal lookup efficiency. In scheme A2, new prefix insertion can easily break the optimal trie structure. Hence, it is also required to be recomputed over a longer period of time in order to minimize the trie size. The recomputing period should not be frequent due to the computational-intensive braiding processing. In contrast, our scheme does not require any reconstruction over time. New prefix can quickly be merged using our simple merging algorithm, and the new prefix can be inserted into the tree by injecting the update bubbles into the traffic stream.

Throughput: We cannot directly compare the throughput with the other 2 schemes as they were not implemented on hardware. However, our implementation on FPGA shows a sustained throughput of 400 million lookups per second, even when external SRAM is used. This translates to the worst-case throughput of 128 Gbps (for a minimum packet size of 40 bytes, or 320 bits).

8. CONCLUDING REMARKS

In this paper, we have described a simple algorithm to merge a number of virtual routing tables. The proposed algorithm is not sensitive to the number of routing tables, but to the total number of prefixes. Hence, a virtual router can be shared by as many virtual router instances as desired, as long as the total number of prefixes is less than a threshold set by the amount of available memory. Along with the merging algorithm, a high-throughput, memory-efficient linear-pipeline architecture has also been proposed and implemented. Also, the architecture can easily interface with external SRAM to handle larger virtual routing tables. Using a state-of-the-art Field Programmable Gate Arrays (FPGA) with external SRAM, the proposed architecture can support up to 16M and 880K prefixes for IPv4 and IPv6, respectively. Our post place-and-route results show that the architecture can sustain a throughput of 400 million lookups per second. With these advantages, our algorithm and architecture can be used in virtual routers that require the following criteria: (1) fast internet link rates up to and beyond 100 Gbps, (2) large size of virtual routing tables, (3) large number of virtual router instances, and (4) quick update to minimize interruption in operation. One drawback of the proposed merging algorithm is that it completely disregards the similarities between routing tables. For future work, we plan to exploit these similarities in order to improve the memory efficiency.

9. REFERENCES

[1] 2-3 Tree [Online]. [http://en.wikipedia.org].

[2] Control plane scaling and router virtualization [Online]. [http://www.juniper.net/us/en/local/pdf/whitepapers/2000261-en.pdf].

[3] MPLS [Online]. [http://en.wikipedia.org].

[4] NetFPGA [Online]. [http://netfpga.org/].

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

[6] Router virtualization in service providers [Online]. [http://www.cisco.com/en/US/solutions/collateral/ns341/ns524/ns562/ns573/white_paper_c11-512753.pdf].

[7] SAMSUNG SRAMs [Online]. [http://www.samsung.com].

[8] F. Baboescu, S. Rajgopal, L. Huang, and N. Richardson. Hardware implementation of a tree based IP lookup algorithm for oc-768 and beyond. In *Proc. DesignCon '05*, pages 290–294, 2005.

[9] 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.

[10] J. Carapinha and J. Jiménez. Network virtualization: a view from the bottom. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 73–80, New York, NY, USA, 2009. ACM.

[11] N. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Comput. Netw.*, 54(5):862–876, 2010.

[12] J. Fu and J. Rexford. Efficient ip-address lookup with a shared forwarding table for multiple virtual routers. In *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–12, 2008.

[13] T. Ganegedara, W. Jiang, and V. Prasanna. Frug: A benchmark for packet forwarding in future networks. In *IPCCC '10: Proceedings of IEEE IPCCC 2010*, 2010.

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

[15] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[16] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[17] K. Sklower. A tree-based packet routing table for berkeley unix. In *Winter Usenix Conf.*, pages 93–99, 1991.

[18] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman. Building scalable virtual routers with trie braiding. In *INFOCOM'10: Proceedings of the 29th conference on Information communications*, pages 1442–1450, Piscataway, NJ, USA, 2010. IEEE Press.

[19] H. Song, M. S. Kodialam, F. Hao, and T. V. Lakshman. Scalable ip lookups using shape graphs. In *Proc. ICNP '09*, 2009.

[20] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17:1–40, 1999.

[21] D. Taylor, J. Turner, J. Lockwood, T. Sproull, and D. Parlour. Scalable ip lookup for internet routers. *Selected Areas in Communications, IEEE Journal on*, 21(4):522 – 534, may. 2003.

[22] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier. Scalable network virtualization using fpgas. In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 219–228, New York, NY, USA, 2010. ACM.

[23] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random generator for ipv6 tables. In *HOTI '04: Proceedings of the High Performance Interconnects, 2004. on Proceedings. 12th Annual IEEE Symposium*, pages 35–40, 2004.

[24] Y.-H. E. Yang and V. K. Prasanna. High throughput and large capacity pipelined dynamic search tree on fpga. In *Proc. FPGA '10*, 2010.