# Towards On-the-fly Incremental Updates for Virtualized Routers on FPGA

Thilan Ganegedara, Hoang Le, Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA90089
Email: {ganegeda, hoangle, prasanna}@usc.edu

***Abstract*—Recently, *router virtualization* has gained much interest in networking community. However, hardware support for router virtualization is still in its primitive stages. One of the major problems in a virtualized router is how to support frequent routing table updates efficiently, without interrupting network traffic. In this paper, we propose a Field Programmable Gate Array (FPGA) based architecture for router virtualization that supports on-the-fly updates, while ensuring scalability and throughput requirements. We introduce a distance-based mapping technique named *Fill-In* to merge multiple virtual routing tables into a single search tree. Node sharing is avoided by using a uniform data structure that results in a scalable solution for router virtualization. The reconfigurability and abundant parallelism of FPGAs make them a desirable hardware platform for high-performance and cost-effective routers. We leverage the features of modern FPGA devices to implement a parallel-linear-pipelined packet processing engine. Our post place-and-route results show that the proposed architecture can support uninterrupted network traffic at $150$ Gbps for minimum size ($40$ Byte) packets. The scalability of the architecture is demonstrated for upto $17$ real routing tables. Using the proposed update techniques, our architecture handles an update with a single write bubble.**

## I. Introduction

Modern networks have stringent, security and quality of service (QoS) requirements. In order to meet these requirements, a router has to perform numerous lookups before actually routing a packet to its destination, such as packet header parsing, firewall search, network intrusion detection, etc. Such lookups are time critical and should be performed at *wire speed*. Meeting these tight requirements is extremely difficult, if not impossible, on software platforms [10]. The kernel function of a router is to perform *IP lookup*. An incoming packet's destination IP is looked up in the routing table to find the port, through which the packet should be forwarded. Field Programmable Gate Arrays (FPGAs) have become a promising platform for high-speed next generation routers, mainly because of their reconfigurability and abundant parallelism [2], [3], [19]. They are used extensively as a hardware platform for multi-Gigabit packet processing engines [2], [10], [13], [18].

Recently, *network virtualization* has drawn a lot of attention from the networking community [4], [5], [11]. Network virtualization allows multiple routers to be consolidated into a single hardware platform. This is known as *router virtualization*. Router virtualization significantly improves the efficiency of the underutilized router hardware. Further, it reduces the equipment and maintenance costs, complexity and power usage of traditional networks. Despite its appealing benefits, little work has been done so as to provide hardware support for router virtualization.

The key metrics for a router are 1) *throughput*, 2) *scalability* and 3) *update time*. The same metrics hold in virtualized domains as well. In the recent solutions for router virtualization, only scalability has been addressed [7], [8], [12], [17], [18]. Scalability is the ability of a router to accommodate the growth of a network, on the available hardware resources (e.g. memory, logic, etc.). The growth can be measured in terms of number of prefixes or number of virtual networks supported. In environments where resources are not limited, scalability requirements are not critical, as they can be met by adding more hardware resources. However, routing table updates or simply, updates, cannot be facilitated by merely adding more hardware.

An update can be of three forms: 1) Modify 2) Insert and 3) Delete. These updates should be applied immediately (i.e. on-the-fly), in order for the packets to be routed correctly. In a virtualized router, updates are more frequent since table update requests from multiple routers should be served . Handling incremental updates in a non-virtualized router itself is a non-trivial problem [1], [6]. Thus, for a virtualized router, this problem is aggravated and may potentially degrade the performance of the router. The routing table residing in a router's memory can be updated using two methods: 1) recompute and reload, or 2) modify the memory while the router is in operation. The former is the simplest but requires blocking of network traffic. The latter can be blocking or non-blocking, but is tedious to be performed in hardware, and is known as *incremental (table) updates* or simply, *updates*. Depending on the amount of preprocessing required, incremental updates can be applied on-the-fly, or can possibly be delayed.

In this paper, we propose a technique named *Fill-In*, to consolidate multiple virtual routers to a single platform in an update friendly fashion, and a FPGA based architecture that supports on-the-fly incremental updates. By exploiting the dual-ported memory modules available in state-of-the-art

This work is supported by the United States National Science Foundation under grant No. 1018801. Equipment grant from Xilinx Inc. is gratefully acknowledged.

FPGA devices, we support *uninterrupted* network traffic at $150$ Gbps throughput for minimum size ($40$ Byte) packets, using a parallel-linear-pipelined architecture. With our proposed table update techniques, we show that our architecture can handle a route update with a single write bubble. In addition, we show that the proposed scheme achieves comparable scalability with the existing techniques for router virtualization. Our main contributions are:

- Support for incremental updates: *Fill-In*, an update friendly routing table merging technique that leads to an FPGA-based lookup architecture to efficiently handle the intermittent and frequent updates, without interrupting network traffic.
- Scalability: Comparable scalability with those of the existing solutions with respect to the number of prefixes/routing tables.
- Fine-grained resource management: Ability to manage the memory usage at each stage of the pipeline, to utilize the available memory in an efficient manner.
- High-speed hardware architecture: The parallel-linear-pipelined FPGA architecture that results in a throughput of $150$ Gbps.

The rest of the paper is organized as follows: Section II discusses related work. Section III presents our solution and the distance-based trie data structure. Our virtualized IP lookup architecture is introduced in Section **??**. Section V demonstrates the results obtained using real routing tables. Section VI concludes the paper.

## II. Related Work

*1) Router Virtualization:* Router virtualization exist in two forms: *separate* and *merged*. In the separate scheme, a router instance is created for each virtual router whereas all the virtual routers are served by only one routing instance in the merged scheme. Both have their advantages and disadvantages. The separate version ensures perfect isolation of routers hence avoids fault propagation from one router instance to the others. However, the extensive resource usage makes this approach less attractive. Juniper JCS1200 control system [11] and Cisco Hardware-Isolated Virtual Router (HVR) [5] adopt this method. In [18], Unnikrishnan et al. implements up to four separate virtual router instances on a NetFPGA-1G (Virtex-II Pro) card [14]. As their results illustrate, the scalability of the separate approach is not high in terms of hardware resources and throughput.

On the other hand, the merged scheme is more scalable than separate. However, it does not enforce isolation since all the routers share the same platform and the associated resources. Another important aspect of any virtualization scheme is fair resource usage. With the merged scheme, it is difficult to ensure this since one router can take up a larger portion of the available resources, leaving the rest of the virtual routers starving. Cisco Software-Isolated Virtual Router (SVR) [5], tree based virtualization [12], Multiroot [8], simple overlaying in [7] and trie braiding [17] are examples for this approach.

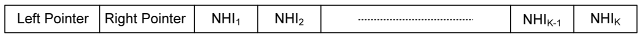


Fig. 1. Shared trie data structure for virtualized router

*2) Routing Table Updates:* Support for updates highly depends on the data structure used [16]. As mentioned in Section I, an update can be of three forms: 1) Modify 2) Insert and 3) Delete. Handling these updates in hardware is a great challenge. A TCAM update can possibly require an amount of time linear in the number of entries. The popular method used to update a tree/trie is using *write bubbles* [10], [12]. A write bubble traverses the pipeline the same way a packet traverses, either in forward or backward direction depending on the architecture. The update is applied at the corresponding stages using the information associated with the write bubble.

In a virtualized router, especially in the merged scheme, one of the main issues is support for incremental updates. Multiroot, simple overlaying and trie braiding requires a complete regeneration of the search tree, to apply an update. Such updates take $O(K \times N)$, $O(N \log N)$ and $O(N^2)$ times respectively, where $N$ is the average number of nodes in a trie and $K$ is the number of virtual routers to be merged. These delays are too large for practical high-speed networking environments. Even though tree based virtualization supports incremental updates, fine grained memory management is not possible due to 2-3 tree data structure [12].

## III. Fill-In Algorithm and Data Structure

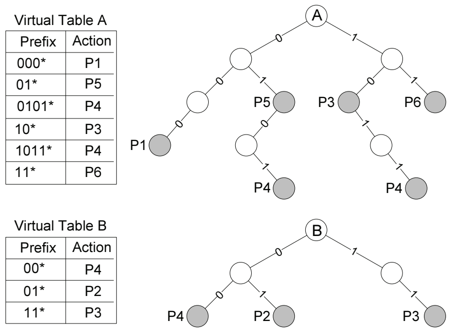
### A. Problem Definition

Given $K$ virtual routing tables $R_i, i = 0, 1, ..., K-1$, find (1) an algorithm to efficiently merge the routing tables with support for on-the-fly incremental updates (2) a scalable virtualized router architecture that can sustain the required throughput levels (specified by the Internet service providers (ISPs)). Further, the updates should not cause network traffic to be blocked or interrupted.

### B. Fill-In: A Distance-Based Mapping Technique

In the merged virtualization approach, all the virtual routing tables, or simply *virtual tables*, are mapped on to one search tree. Depending on the mapping technique, the performance of a given scheme can vary significantly [7], [8], [12], [17]. In *trie* data structure, the issue arises mainly because of node sharing. For example, in [7], [8] and [17], a single node has to serve multiple virtual networks. Each node has to have next hop information (NHI) field for all $K$ virtual tables. An example of a shared node is illustrated in Figure 1. Techniques such as *leaf pushing* can be applied to reduce the memory consumption, by pushing all the next hop information to the leaf nodes [16]. However, as we show in Section V, this makes incremental updates more difficult and also, memory efficiency can be poor.

Considering these drawbacks of node sharing, we introduce a non-shared trie data structure for virtualized routers. However, we employ the merged scheme due to its resource efficiency. In order to map multiple virtual routers on to a single lookup architecture, we use a distance-based mapping


Virtual Table A

| Prefix | Action |
|---|---|
| 000* | P1 |
| 01* | P5 |
| 0101* | P4 |
| 10* | P3 |
| 1011* | P4 |
| 11* | P6 |

Virtual Table B

| Prefix | Action |
|---|---|
| 00* | P4 |
| 01* | P2 |
| 11* | P3 |

Fig. 2. Virtual tries for tables A and B

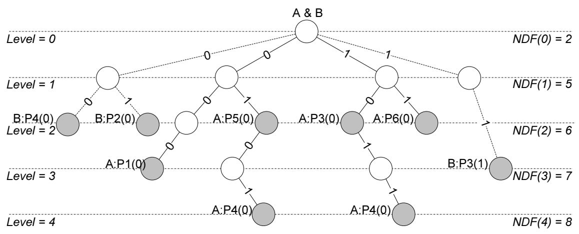


Fig. 3. Fill-In trie of virtual tries A and B

scheme called *Fill-In*. Fill-In takes the uni-bit tries built for all the virtual tables as input, and builds a single search tree that can be used to perform IP lookup for all the input virtual tables. We describe our distance-based mapping algorithm using the two virtual tries shown in Figure 2. Our algorithm shares similarities with [9], however we use Fill-In to merge routing tables and facilitate updates rather than for memory balancing.

The granularity of resources on FPGA depends on the underlying architecture. On modern FPGAs, the Block RAM (BRAM) is organized in either 36 Kbit (18 Kbit in older devices) size blocks. When the FPGA's distributed RAM is completely used or when it cannot accommodate the requirement, data should be stored in BRAM. In such situations, a complete BRAM block is allocated irrespective of the amount of data to be stored. If we were to use binary tries shown in Figure 2, the BRAM utilization becomes low and leads to inefficient memory usages. Fill-In takes this fact also into account. The designer has complete control over the number of nodes at each level, which we call the *node distribution*. Depending on the memory organization, the desired node distribution can change. However, for a binary trie it is fixed. With Fill-In, an arbitrary node distribution can be used. To show the effect of node distribution function (NDF) on the final architecture, for our implementation, we consider the following linear function:

$$NDF(l) = l \times K \times C$$

where $NDF(l)$ is the allowed number of nodes at level $l$, $K$ is the number of virtual routers, and $C$ is a design-time constant. Note that this function can be changed by the user, depending on the resource usage requirements. Also, $l = 0$ is reserved for the root node.

Fill-In merges the virtual tries one by one, constrained to the node distribution function described above. The algorithm is described in Algorithm 1. The Fill-In trie for the two tries in Figure 2 is shown in Figure 3. In this figure, we have used *arbitrary NDF values* to illustrate the operation of our scheme.

When two nodes from consecutive levels are mapped to the same level of the Filled-In trie, the search becomes complicated. In a pipelined implementation, this causes the search to perform two lookups if a child node and its parent exist in the same level. Since this degrades the performance, we avoid the above by checking the consistency of levels, of a node and its parent. For example, as shown in Figure 3, even though the NDF of level 1 is 5, level 2 nodes of trie B cannot be mapped to level 1 of Filled-In trie due to level inconsistency.

As a consequence of the above, and the constraints imposed by NDF, a node might not end up in its original level in the trie. This level shift is recorded as the *distance* value, in a node's respective parent. As Figure 3 shows (the value within parenthesis shows a node's level shift), the last node of trie B cannot be mapped to level 2, since the NDF value is 6. Therefore, it is moved to the immediate lower level (level 3 in this case). This level shift is stored in the parent node, for both left and right children. These shifts might cause the number of levels to increase above the bit width of an IP address, depending on the NDF used. However, as we show in Section V, this can be avoided by selecting the NDF appropriately.

**Algorithm 1** Fill-In(NDF)

```
Require: Virtual tries $T_k$, $k = 0, 1, ..., K-1$, Empty trie $T_m$
 1: for all Virtual trie $T_k$ do
 2:     Begin at $l = 0$
 3:     while level $l < L$ do
 4:         for all Node $n$ of $T_k$ at level $l$ do
 5:             if LevelInconsistent($n$) then
 6:                 $l \leftarrow l + 1$; Continue;
 7:             end if
 8:             if $|Nodes@Level(l, T_m)| < NDF(l)$ then
 9:                 Add $n$ to $T_m$
10:                 Update $n$'s parent's pointers
11:                 Update $n$'s distance
12:             else
13:                 $l \leftarrow l + 1$; Continue;
14:             end if
15:         end for
16:     end while
17: end for
18: return $T_m$
```

## C. Node Structure for Fill-In

Having a uniform node structure eases the update complexity of any router, especially on hardware. When a node is created on hardware, its structure cannot be changed at runtime, if changing the node requires more memory than what it is originally allocated. Leaf pushing, for example, creates two types of nodes: leaf and non-leaf. Non-leaf nodes contain two pointer fields (e.g. $2 \times 16$-bit) whereas the leaf nodes only contain the next hop information (e.g. 6-bit). Suppose the router received an update to change a leaf node to a non-leaf

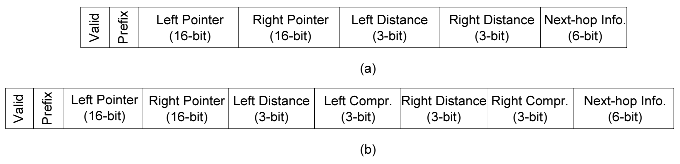


Fig. 4. Node structures for IP lookup with Fill-In: (a) IPv4 and (b) IPv6

node. Unless all the nodes are allocated the memory required for a non-leaf node, it is impossible to apply this change without a complete reconfiguration. This includes an undesirable delay overhead. On the other hand, allocating maximum size for all the nodes results in poor memory efficiencies. This issue is more noticeable in virtualized schemes.

A uniform node structure overcomes this issue since all the nodes have exact same structure. Thus applying an update can be done quickly and incrementally. In our merging scheme, we require each node to have all the necessary fields, even if some of them might not be used at a particular level. In regular trie data structures, uniformity can be achieved by letting all the nodes have two pointer fields and a next hop information field. However, in our scheme, we require each parent node to store the distance values of its child nodes. The above method is sufficient for IPv4 ($32$-bit prefixes). For IPv6 (effectively $64$-bit prefixes), we use path compression [16] to alleviate the address length issue. The node structures used for IPv4 and IPv6 are shown in Figure 4. Due to space limitations, we do not discuss our IPv6 results in this paper.

### D. Memory Requirement Analysis

We give a brief theoretical analysis of the memory requirement of our scheme, compared with the existing approaches, in Table I. The notations used are as follows: $K$ - number of virtual tables/routers, $N_k$ - number of nodes in virtual trie $k$, $N_{max}$ - maximum of all $N_k$'s, $P$ - bit width of pointer field, $D$ - bit width of distance field, $H$ - bit width of next hop information field.

TABLE I
MEMORY REQUIREMENT ANALYSIS

| Scheme | Memory requirement |
|---|---|
| Fill-In | $\Theta(\sum_{k=0}^{K-1} N_k * (2P + 2D + H))$ |
| Separate [18] | $\Theta(\sum_{k=0}^{K-1} N_k * (2P + H))$ |
| Simple overlaying [7] | $\Omega(N_{max}/2 * (2P + H * K))$ |
| Trie braiding [17] | $\Omega(N_{max}/2 * (2P + H * K + K))$ |

## IV. FPGA IMPLEMENTATION

We use a parallel-linear pipelined architecture to perform IP lookup for our Fill-In trie and to perform on-the-fly incremental updates while ensuring throughput requirements for all the virtual routers. The IP lookup portion and update portion of our architecture are described in Sections IV-A and IV-B, respectively. Due to space limitations, only the architecture for IPv4 is presented.

### A. Architecture: IP Lookup

IP lookup in Fill-In is similar to any trie based pipelined IP lookup architecture, except for the additional distance field. The distance value simply lets the packet skip few stages of the pipeline. This can be achieved by executing a *No-op* (no operation) whenever the distance is non-zero. The distance value gets decremented as it traverses the pipeline and when the value becomes zero, the appropriate memory address is accessed, either to decide the next node to visit or to acquire next-hop information. The IP lookup architecture is illustrated in Figure 5. The *valid* and *prefix* bits, shown in Figure 4, are examined during the search to decide the next node to visit and to update next-hop field, respectively.

To take advantage of the dual-ported BRAM, we implement dual-linear pipelines to achieve twice the throughput of a single pipeline. This allows the two pipelines to share the same stage-memories to perform parallelized packet processing. With dual-ported BRAM, two memory accesses can be served independently, in a single clock cycle. These memory accesses can be Reads (R) or Writes (W). For IP lookup, R operations are used whereas for updates (Section IV-B) W operations are used. Depending on the inputs, the two pipelines can operate in the following modes: RR, RW, WR or WW.

### B. Architecture: Incremental Updates

In order to provide support for incremental updates, we augment each stage of the pipeline shown in Figure 5 with an update module. In Section I, three types of updates were mentioned. Out of the three, modifications are the easiest. An existing prefix can be updated by sending a *write bubble* with the corresponding stage, memory address and the updated prefix information. It should be noted that in [7] and [17], even a prefix update requires a complete reloading of the updated routing table, since leaf-pushing can potentially require multiple node updates in a single stage, for a prefix modification.

Inserts and deletes are more complex than a prefix modification. However, by using a write bubble table [12] and dual-ported memory on FPGA, these updates can be easily handled in our pipelined architecture. For a trie data structure update, only a single node update is required at a given stage. This requires one write operation at the respective stages, which can be performed on either of the two pipelines. The remaining pipeline can be used for IP lookup. Since updates do not require regular traffic to be blocked, they are *non-blocking*. Figure 6 illustrates our architecture for the three types of updates mentioned earlier. The modifications are treated differently as opposed to inserts/deletes, and are distinguished by the *update type* field in the write bubble. Currently, we set the size of the write bubble table of our architecture to the number of virtual tables supported, so that each virtual router gets an opportunity to queue one update per lookup operation. Update scheduling is beyond the scope of this paper, hence not discussed.

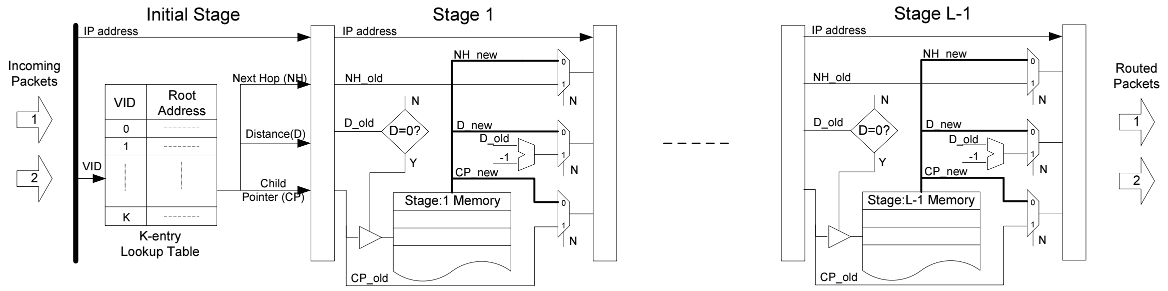


Fig. 5. IP lookup architecture for Fill-In on FPGA. Two packets/updates can be fed, every clock cycle, to the parallel-linear pipeline

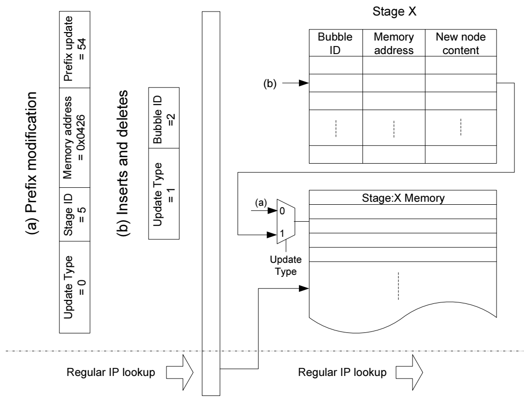


Fig. 6. Support for updates (a) Modifies and (b) Inserts/Deletes with dual-ported BRAM

## V. Experimental Results

### A. FPGA Platform and Routing Table Sources

For our experiments, we considered a Xilinx Virtex 5 device (XC5VLX220) as our target platform. It has $6912$ Kbits of BRAM, $2280$ Kbits of distributed RAM and supports up to $550$ MHz clock frequency. This device provides adequate resources (memory and logic) for our experiments.

In order to evaluate the performance of our approach, we obtained several *real* routing tables from [15]. We considered routing tables of 17 different edge networks connected to the Internet. The routing table sizes varied from 37 to 3725 prefixes, which had a total of 14094 prefixes. We avoided using the publicly available core routing tables as their prefix distributions are similar and have nearly the same number of prefixes. Further, we avoided partitioning core routing tables to generate smaller routing tables, as this causes the generated routing tables to be unrealistic. We do not assume any particular structure for the routing tables as done in [7] and [17]. Also, we make no assumptions on the size of the routing tables considered. This makes our solution generic for any set of virtual routing tables.

### B. Update Capability

Our main claim in this work is support for on-the-fly incremental updates. Using the FPGA based architecture introduced in Section IV, we support table updates for all three types of updates. Each update requires only one write bubble, that can be executed in $L$ clock cycles. However, it should be noted that depending on the NDF used, the length of the pipeline

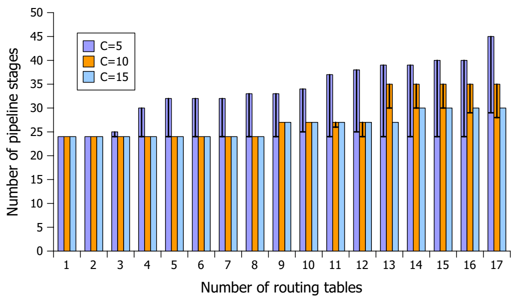


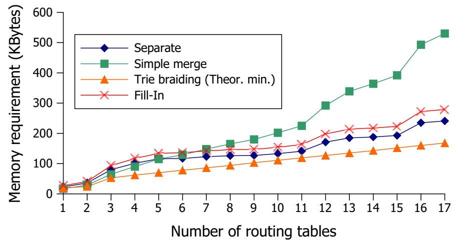
Fig. 7. Variation of pipeline length for different $C$ values

can increase. By choosing an appropriate NDF, one can avoid or limit the increase in the pipeline length. Figure 7 illustrates the effect of NDF on the pipeline length. For the experiments, we used the NDF mentioned in Section III. It can be seen that the choice of value $C$ directly affects the length of the pipeline. Inside each column, we show the maximum distance a node has experienced. In our experiments, a $C$ value of $15$ resulted in zero distance for all the nodes.

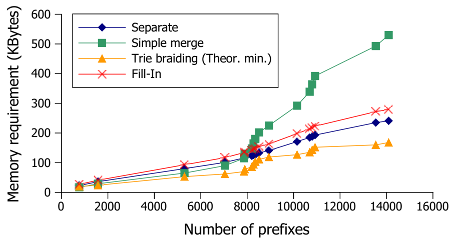
To run Fill-In on a trie with $N$ nodes, it takes $O(N)$ time. For the largest routing table used in our experiments, it took only $0.02$ ms to complete the execution of Fill-In, on a dual Quad-core AMD Opteron $2350$ processor running at $2.0$ GHz. Our experiments show that, computing the write bubble content for a single table update can be done at wire-speed. Therefore, Fill-In does not become a bottleneck for the update process. Previous work such as trie braiding [17] takes time in the order of seconds for preprocessing to achieve its optimal memory efficiency.

### C. Scalability

Scalability is the ability of a router to accommodate the growth of a network, on the available hardware resources (e.g. memory, logic, etc.) and can be measured with respect to the number of prefixes or routing tables. We use total memory consumption (pointer and next-hop information) as the metric. Figure 8(a) compares various techniques with respect to this metric when the number of routing tables ins varied. Note that, this metric is relevent if the routing tables are of the same size. In our experiments, we use routing tables of different sizes. Hence, Figure 8(b) does not reflect the actual memory increase of our scheme. Memory increase with the number of prefixes is shown in Figure 8(a). In both figures, it can be seen that Fill-In achieves almost linear memory increase. Note that, for trie braiding, we have used the theoretical minimum memory requirement mentioned in Table I. However, depending on the prefix distribution of the tables, this estimation can easily increase.



(a)



(b)

Fig. 8. Memory requirement for increasing (a) virtual tables and (b) prefixes

## D. Performance and Resource Usage

Here we analyse performance and resource usage of our FPGA architecture. The performance is measured in terms of clock frequency and throughput for minimum size ($40$Byte) packets. Further, the number of BRAM blocks and slices used are measured. To show the effect of the *distance*, we use the pipeline constructed for $C = 5$. Our experiments show that the proposed architecture is able to sustain a throughput of $150$ Gbps, in average. The results are presented in Figure 9.

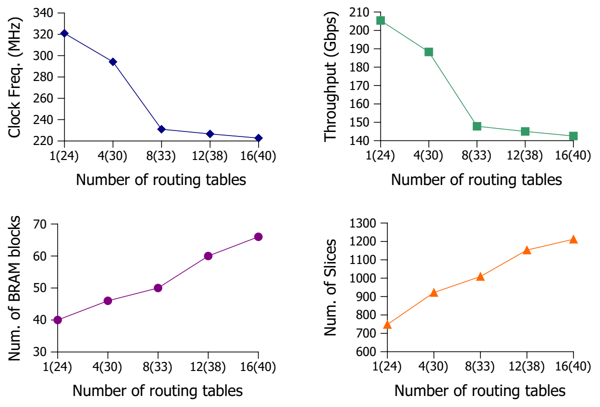


Fig. 9. Performance (clock frequency and throughput) and resource usage (number of BRAM blocks and slices used) of the FPGA-based architecture. X-axis shows the number of virtual routers and within parenthesis, the number of pipeline stages

## VI. Conclusion and Future Work

In this paper, we discussed a novel distance-based mapping technique, Fill-In, to merge multiple virtual tables into a single search tree. Additionally, we provided a high-speed FPGA architecture to perform IP lookup as well as on-the-fly updates for a virtualized router. This has not been addressed in any of the previous work. During our initial experiments, we noticed that node sharing results in very inefficient memory usage when the routing table sizes vary. Therefore, we considered the proposed mapping technique to avoid node sharing, which resulted in an update friendly, yet a scalable solution. Fill-In is extended to IPv6 using path compression techniques. Our future work will focus on, how to eliminate the need for a write bubble table by keeping track of the available free memory locations, which will further enhance the update process.

## References

[1] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. *Networking, IEEE/ACM Transactions on*, 13(3):690 – 703, 2005.

[2] M. Baxter and G. Brebner. Area-efficient 100g+; efec calculation with xilinx fpgas. In *Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference, 2010 Conference on (OFC/NFOEC)*, pages 1 –3, 2010.

[3] G. Brebner. Packets everywhere: The great opportunity for field programmable technology. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 1 –10, 2009.

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

[5] Cisco. Hardware and software virtualized routers. http://www.cisco.com/en/US/solutions/collateral/ns341/ns524/ns562/ns573/white_paper_c11-512753_ns573_Networking_Solutions_White_Paper.html.

[6] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34:97–122, April 2004.

[7] J. Fu and J. Rexford. Efficient ip-address lookup with a shared forwarding table for multiple virtual routers. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 21:1–21:12, New York, NY, USA, 2008. ACM.

[8] T. Ganegedara, W. Jiang, and V. Prasanna. Multiroot: Towards memory-efficient router virtualization. In *Communications (ICC), 2011 International Conference on*, 2011.

[9] W. Jiang and V. Prasanna. A memory-balanced linear pipeline architecture for trie-based ip lookup. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 83 –90, 2007.

[10] W. Jiang and V. K. Prasanna. Multi-terabit ip lookup using parallel bidirectional pipelines. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 241–250, New York, NY, USA, 2008. ACM.

[11] Juniper. Jcs1200 control system. http://www.juniper.net/us/en/local/pdf/whitepapers/2000261-en.pdf.

[12] H. Le, T. Ganegedara, and V. Prasanna. Memory-efficient and scalable virtual routers using fpga. In *Field Programmable Gate Arrays (FPGA), 2011 International Symposium on*, 2011.

[13] H. Le and V. Prasanna. Scalable high throughput and power efficient ip-lookup on fpga. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 167 –174, 2009.

[14] NetFPGA. Netfpga boards. http://netfpga.org/.

[15] Potaroo. Bgp analysis reports. http://bgp.potaroo.net/.

[16] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 15(2):8 –23, 2001.

[17] H. Song, M. Kodialam, F. Hao, and T. Lakshman. Building scalable virtual routers with trie braiding. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –9, 2010.

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

[19] Xilinx. Xilinx xcell journal. http://www.xilinx.com/publications/xcellonline/.