High-Throughput IP-Lookup Supporting Dynamic Routing Tables using FPGA

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

***Abstract*—Advances in optical networking technology are pushing internet link rates up to 100 Gbps. Such line rates demand a throughput of over 150 million packets per second at core routers. Along with the increase in link speed, the size of the dynamic routing table of these core routers is also increasing at the rate of 25-50K additional prefixes per year. These dynamic tables require high prefix deletion and insertion rates. Therefore, quick prefix update without disrupting router operation has also emerged as a critical requirement. Furthermore, IPv6 standard extends the current IPv4 prefix length from 32 to 128 bits. Thus, it is a major challenge to scale the existing solutions to simultaneously support increased throughput, table size, prefix length and quick update. While the existing solutions can achieve high throughput, they cannot support *large routing tables* and *quick update* at the same time. We propose a novel *scalable, high-throughput* linear pipeline architecture for IP-lookup that supports *large routing tables* and *single-cycle non-blocking update*. Using a state-of-the-art Field Programmable Gate Arrays (FPGA) along with external SRAM, the proposed architecture can support over 2M prefixes. Our implementation shows a throughput of 348 millions lookups per second, even when external SRAM is used.**

I. Introduction

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]. 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. In SRAM-based solutions, the common data structure in algorithmic solutions for performing LPM is some form of a tree. In these solutions, pipelining techniques are used to improve the throughput. These SRAM-based 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, *overlapping* in prefix ranges prevent IP lookup from employing tree searching algorithms without modification. The ability to utilize external SRAM also becomes an important factor in using devices with limited on-chip memory. 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.

Supported by the U.S. National Science Foundation under grant No. CCF-0702784. Equipment grant from Xilinx is gratefully acknowledged.

We propose and implement a *scalable high-throughput, SRAM-based* linear pipeline architecture for IP-lookup that supports 1-*cycle* update. This architecture utilizes 2-3 tree structure to achieve high throughput and supports quick update. Moreover, the lower levels of the tree can be moved onto external SRAMs to overcome the limitation in the amount of the on-chip memory.

This paper makes the following contributions:

1) A data structure that eliminate *overlapping* in IP prefix ranges (Section III).
2) An architecture based on 2-3 tree that (i) can use *external* SRAM to support a routing table consisting of over 2M prefixes (Section IV), (ii) support *single-cycle* update even in the worst case (Section V), and (iii) achieve a *sustained throughput* of 348 MLPS (Section VI).

The remainder of the paper is organized as follows. Section II covers the background and related work. Section III introduces the proposed IP-lookup algorithm. Sections IV describes the architecture and its implementation. Section VI presents implementation results. Section VII concludes the paper.

II. Background and Related Works

A sample routing table with the maximum prefix length of 8 is illustrated in Table I. In this routing table, binary prefix P_5 (01001$*$) matches all destination addresses that begin with 01001. Similarly, prefix P_6 matches all destination addresses that begin with 01011.

Despite the large amount of proposed IP-lookup solutions, most of them do not target FPGA platform. Since the proposed work addresses FPGA implementation, we only summarize the related work in this area. In general, architectures for IP-lookup on FPGA can be classified into the following categories: trie-based approach, tree-based approach, and hash-

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

	Prefix	Next Hop		Prefix	Next Hop
P_1	0*	1	P_7	011*	7
P_2	000*	2	P_8	1*	8
P_3	010*	3	P_9	10*	9
P_4	01000*	4	P_{10}	110*	10
P_5	01001*	5	P_{11}	11010*	11
P_6	01011*	6	P_{12}	111*	12

based approach. Each of the three approaches has their own advantages and disadvantages.

In *trie-based architectures* ([2], [3], [4], [5], [6], [7], [8]), IP-lookup is performed by traversing the trie according to the bits in the IP address. Multibit-trie is also used in [4]. These architectures are simple, have high throughput, and support quick update. However, the main disadvantages of the trie-based solutions are: (1) latency proportional to prefix length, (2) moderate memory efficiency, and (3) difficult to interface with external memory.

In *tree-based architectures* ([9], [10]), each prefix is treated as a search key and the entire routing table is stored into multiple binary search trees [9] or in one binary search tree [10]. The advantages of these architectures are: (1) simple architecture, (2) good memory efficiency, (3) high-throughput (over 300 MLPS), (4) latency independent of the length of prefixes, and (5) easy to use external memory. Their major drawback is the lack of support for quick update.

In *hash-based architectures* ([4], [11], [12]), hashing functions or bloom filters are used. These architectures have fast lookup rate (over 250 MLPS) with good hashing function, low latency, and moderate memory efficiency. However, they suffer from non-deterministic performance due to hash collisions and slow update rate.

III. IP-LOOKUP ALGORITHM

A. Algorithm

Definition 1: Given two distinct prefixes, P_A and P_B ($|P_A| < |P_B|$), if P_A is a prefix of P_B then P_A is a *child prefix* of P_B, and P_B is a *parent prefix* of P_A. P_A and P_B are said to be *overlapped*.

We propose a memory efficient data structure based on a 2-3 tree, which is a balanced search tree. A 2-3 tree has three different types of nodes: a leaf node, a 2-node and a 3-node. A 2-node is the same as a binary search tree node, which has one data field and references to two children. A 3-node contains two data fields, ordered so that the first is less than the second, and references to three children.

Let T be the given routing table. In order to use the 2-3 tree as the data structure to perform IP-lookup, we need to process T to eliminate *prefix overlap*. T can simply be processed by merging the child prefixes with their parents. For example, P_3 is merged with P_4, P_5, and P_6. We build the trie from table T. The leaf-prefixes representing parent-prefixes are grouped together in the parent-prefix table T_p. The next hop routing information of the child-prefixes are stored in child-prefix table T_c. The corresponding trie of the sample

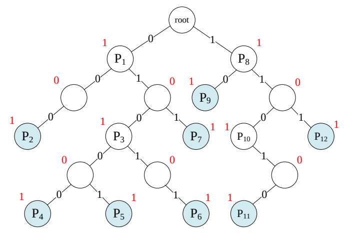
TABLE II
SAMPLE ROUTING TABLE IN TABLE I AFTER PROCESSING

(a) Parent-prefix table T_p

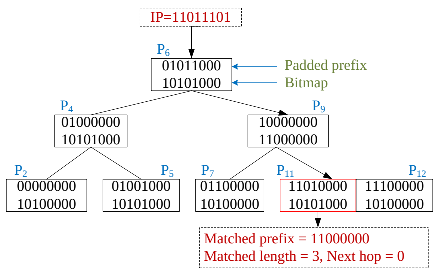
	Padded prefix	Bitmap	Next Hop
P_2	00000000	10100000	2
P_4	01000000	10101000	4
P_5	01001000	10101000	5
P_6	01011000	10101000	6
P_7	01100000	10100000	7
P_9	10000000	11000000	9
P_{11}	11010000	10101000	11
P_{12}	11100000	10100000	12

(b) Child-prefix table T_c

	Prefix	Length	Next Hop
P_1	00000000	1	1
P_3	01000000	3	3
P_8	10000000	1	8
P_{10}	11000000	3	10



(a) Sample trie



(b) Parent-prefix search tree

Fig. 1. The trie and 2-3 tree of the sample routing table in Table I

routing table in Table I is shown in Figure 1(a). The edge value represents a bit in the prefix. The value next to each node indicates whether that node is a prefix, and is called *prefix bit*.

Prefixes in parent table T_p are padded with “0s” up to the maximum prefix length (32 for IPv4 and 128 for IPv6). The *bitmap* column is introduced to keep track of the sub-prefixes included in the current prefixes. The *bitmap* value of a prefix is calculated by collecting all prefix bits from the root to the prefix. The bitmap value is also padded with “0s” up to the maximum prefix length. The *padded prefixes* are used to build the parent-prefix 2-3 tree. Each entry is assigned a next hop information associated with the parent-prefix. The *padded prefixes* are used as index keys to build the tree. The corresponding prefix table and 2-3 tree of the sample routing table in Table I are depicted in Table II(a) and Figure 1(b), respectively.

Table T_c is used to lookup the next hop routing information of the child prefix. Entries in table T_c are identified by the padded child-prefixes and their length. The padded child-prefix, appended with its length, is used as an index key to build the child 2-3 tree. Since all combinations of $\{prefix, length\}$ are uniquely identified in table T_c, any search algorithm can be used. A similar 2-3 tree as the parent tree is used to support the quick update.

After building the parent and child tree, IP-lookup is performed as follows. The input is the incoming IP address. First, the parent-prefix tree is searched. Three data fields are returned: *matched prefix*, *matched length* and *next hop*. At each node of the parent 2-3 tree, 2 comparisons are performed to determine (1) the direction of traversal and (2) the match status. The first comparison involves the node value and the incoming address IP. If the node is a 2-node, which has one 1 data field f_1, then the direction is *left* (if $IP \leqslant f_1$), 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 $IP \leqslant f_1$), *middle* (if $f_1 < IP \leqslant f_2$), or *right* otherwise. In the second comparison, all returned data fields are computed. Note that the *next hop* is only updated if the IP address *fully* matches the parent prefix at that node.

Search in the child tree is *only necessary* if the *next hop* returned from the first step is 0 and the *matched length* is non-zero. In this case, the *matched prefix* and the *matched length* from the parent tree are used as the *search key*. This problem can be seen as finding the exact match of the search key.

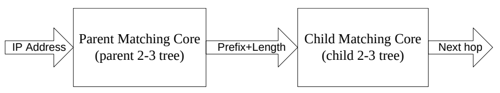
B. IP-Lookup Algorithm Analysis

Let N be the number of prefixes and L be the length of the IP address ($L = 32$ for IPv4 and 128 for IPv6). The preprocessing step has the computational complexity of $O(NL)$ since every bit of the routing table needs to be scanned to build the trie. The parent table T_p and child table T_c can be built in $O(NL)$ time. Let N_1, N_2 be the number of leaf nodes (parent prefixes) and non-leaf prefix nodes (child prefixes) in the trie, respectively. Clearly, $N_1 + N_2 = N$. If there is no overlap in the routing table, then $N_1 = N, N_2 = 0$. The parent tree and child tree can be constructed in $O(N)$ time. Search, insertion and deletion in 2-3 trees take $O(\log_2 N)$ time. It will be clear in Section IV that these update time can be brought down to exactly 1 *cycle* by using pipelining techniques and parallelism in hardware.

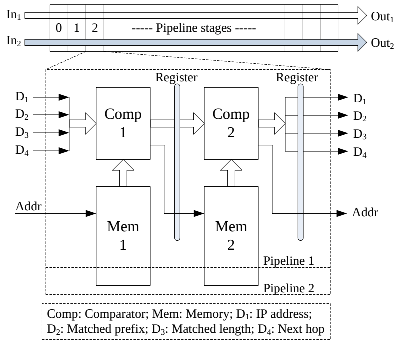
IV. ARCHITECTURE

The overall architecture is depicted in Figure 2(a). As mentioned above, there are 2 matching cores: (1) parent matching and (2) child matching. The 2 cores are connected in a tandem fashion. The first core matches the IP address of the incoming packet against the parent table, while the second core matches the *prefix-length* combination returned by the first core to find the next hop of the matched child-prefix.

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). 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.



(a) Overall architecture



In1
In2
0
1
2
----- Pipeline stages -----
Out1
Out2
Register
Register
D1
D2
D3
D4
Comp 1
Comp 2
Addr
Mem 1
Mem 2
Pipeline 1
Pipeline 2
Comp: Comparator; Mem: Memory; D1: IP address;
D2: Matched prefix; D3: Matched length; D4: Next hop

(b) A basic pipeline stage of the parent matching core

Fig. 2. Block diagram of the proposed IP-lookup architecture

The available memory management is also more complicated. To overcome this problem, a 3-node is divided into two 2-nodes. The left pointer of the first node points to the left child. The right pointer of the first node points to the second node. The left pointer of the second node points to the middle child. Finally, the right pointer of the second node points to the right child. Although a pointer is wasted, the benefit is justifiable. It creates uniformity in the size of the nodes, and simplifies the memory management.

The parent table T_p has N_1 entries. Since each 3-node is split into two 2-nodes and each node hold one prefix, the effective number of 2-nodes is also N_1. Each 2-node consists of a prefix field of L bits, a bitmap of L, and 2 pointers of $\log N_1$ bits each. Hence the size of each node is $(2L + 2\log N_1)$. The parent tree requires a storage size of $2N_1(L+\log N_1)$. Similarly, the child 2-3 tree has N_2 effective 2-nodes. Each node consists of a prefix field of L bits and 2 pointers of $\log N_2$ bits each. The child tree requires a storage size of $N_2(L + 2\log N_2)$. The total storage size S required for both trees is $S = 2N_1(L + \log N_1) + N_2(L + 2\log N_2)$. The maximum storage size is reached when $N_1 = N$, in this case $S_{max} = 2N(L + \log N)$. In reality, nodes in the last level of the tree need not contain any pointer as they have no children. The number of leaf-nodes is at least half the total number of nodes, i.e $N/2$. The maximum storage size required can be brought down to $S_{max} = N(2L + \log N)$. Hence, a FPGA device with 18 Mb of on-chip memory can support a routing table of over 200K prefixes (for IPv4), or up to 64K prefixes (for IPv6), *without* using external SRAM. Note that the on-chip memory of FPGA (BRAM) is dual-ported. To take advantage of this, the architecture is configured as dual-linear pipelines. This configuration doubles the lookup rate.

In our design, external SRAMs can be used to handle even larger routing tables, by moving the last stages of the pipeline onto external SRAMs. Currently, SRAM is available in $2-32$ Mb chips [13], with data widths of 18, 32, or 36 bits, and a maximum access frequency of over 500MHz. 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. Hence, the architecture can support at least 2M IPv4 prefixes, or 512K IPv6 prefixes. 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.

V. Supporting Quick Update Operations

The insertion and deletion cause changes in *at most two nodes* at each level of the tree, in the worst case [14]. 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 this 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 pre-computed, the update operation is performed starting from the root of the tree. This update can easily be done by inserting *only one* write bubble [2]. There is one dual-ported write bubble table (WBT) in each stage. Each table consists of 2 entries, each composes of (1) the memory address to be updated in the next stage and (2) the new content for that memory location. 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.

VI. Evaluation Results

We collected 4 IPv4 routing tables from Project - RIS [15] on 06/03/2010. The number of parent prefixes, child prefixes and the total number of prefixes for each routing table are shown in Table III. The proposed IP-lookup architecture was implemented on Virtex-5 FX200T. The implementation showed a minimum clock period of 5.75 ns, or a maximum frequency of 174 MHz. With a dual pipeline architecture, this design can achieve a throughput of 348 MLPS, or 111.4 Gbps.

With a lookup rate of 340 MLPS, our design achieves a throughput comparable to that of the state-of-the-art (Section II). Our architecture, however, can support *single-cycle* updates, which include prefix modification, insertion and deletion, even in the worst case. Among other solutions, the trie-based approaches, which are known to support quick updates, require multiple cycles to preform an update. The tree-based approaches may require the entire memory to be reloaded, and the hash-based solutions may need partial/full chip reconfiguration.

TABLE III
Experimental routing tables (as of 03/24/2010)

Table	# prefixes	# parent prefixes	# child prefixes
rrc00	325984	295066	30918
rrc11	317716	288416	29345
rrc12	316409	287347	29062
rrc16	324919	294199	30720

VII. Concluding Remarks

This paper proposed a data structure and a scalable high throughput, SRAM-based pipeline architecture for IP-lookup, that does not use TCAM. The data structure eliminates the prefix overlapping. By employing a pipelined 2-3 search tree architecture, dynamic routing updates (modification, insertion and deletion) can be done in exactly 1 *cycle*. The architecture can easily interface with external SRAM to handle larger routing tables. Our design sustained a lookup rate of 348 MLPS even when external SRAM is used. This throughput translates to at least 111.4 Gbps (using the minimum packet size of 320 bits), which is 2.8× the speed of an OC-768 data line. The proposed architecture can also be scaled to support IPv6.

References

[1] 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*, 2005, pp. 290–294.

[2] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," in *Proc. INFOCOM '03*, 2003, pp. 64–74.

[3] H. Le, W. Jiang, and V. K. Prasanna, "A sram-based architecture for trie-based ip lookup using fpga," in *Proc. FCCM '08*, 2008.

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

[5] W. Jiang and V. K. Prasanna, "A memory-balanced linear pipeline architecture for trie-based ip lookup," in *Proc. HOTI '07*, 2007, pp. 83–90.

[6] 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*, 2005, pp. 123–133.

[7] I. Sourdis, G. Stefanakis, R. de Smet, and G. N. Gaydadjiev, "Range tries for scalable address lookup," in *Proc. ANCS '09*, 2009.

[8] O. Erdem and C. F. Bazlamaçi, "Array design for trie-based ip lookup," *Comm. Letters.*, vol. 14, no. 8, pp. 773–775, 2010.

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

[10] H. Le, W. Jiang, and V. K. Prasanna, "Scalable high-throughput sram-based architecture for ip-lookup using fpga." in *FPL*. IEEE, 2008, pp. 137–142.

[11] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed ip lookup algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, 2005.

[12] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," in *Proc. ANCS '05*, 2005, pp. 81–90.

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

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

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