Scalable Balanced Pipelined IPv6 Lookup Algorithm

Zoran Cica
Department of Telecommunications, University of Belgrade School of Electrical Engineering,
Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia
zoran.cica@etf.bg.ac.rs

***Abstract*—One of the most critical router's functions is the IP lookup. For each incoming IP packet, IP lookup determines the output port to which the packet should be forwarded. IPv6 addresses are envisioned to replace IPv4 addresses because the IPv4 address space is exhausted. Therefore, modern IP routers need to support IPv6 lookup. Most of the existing IP lookup algorithms are adjusted for the IPv4 lookup, but not for the IPv6 lookup. Scalability represents the main problem in the existing IP lookup algorithms because the IPv6 address space is much larger than the IPv4 address space due to longer IPv6 addresses. In this paper, we propose a novel IPv6 lookup algorithm that supports very large IPv6 lookup tables and achieves high IP lookup throughput.**

***Index Terms*—IP lookup; IP networks; Packet switching; Routers.**

I. INTRODUCTION

The Internet constantly grows in every aspect: number of hosts and network devices, number of links, link capacities, variety of services and applications, QoS demands. Internet routers are the main components of the Internet infrastructure because the routers are responsible for the forwarding of IP packets to their proper destinations. Thus, routers enable the Internet's global connectivity. Therefore, performances of the Internet routers must be continuously upgraded so that the routers could support and enable Internet's growth. The Internet routers must implement efficient high-speed packet processing to keep up with the Internet's growth. Packet processing includes complex tasks, such as IP lookup, packet classification, packet scheduling and switching, packet buffering, etc. [1].

IP lookup is one of the most critical packet processing functions [1]–[3]. For each incoming packet, IP lookup examines the lookup table using the packet's destination IP address and retrieves the next-hop information. Lookup table contains forwarding information for all the networks known to the router. Forwarding information, collected via routing protocols (BGP, OSPF, RIP, IS-IS), comprises the pairs IP prefix and next-hop information (NHI). IP prefix represents the destination network address (or aggregated network address that represents multiple destination networks). We use term "prefix" to denote IP prefix in the remaining part of the paper. NHI represents the ID of the output port to which a packet needs to be forwarded.

Manuscript received 3 February, 2021; accepted 11 June, 2021.

The worst case, considering the IP lookup, is when only the shortest possible packets are arriving at a router. In this worst case scenario, IP lookup must be completed in a time that is equal to the shortest packet duration. Otherwise, packets would pile up and eventually some packets would be dropped. Nowadays, link capacities are extremely high, so the shortest packet duration is very small. For example, IP lookup function should support one IP lookup per 5.12 ns in case of ethernet frames on 100 Gbps link [1]. This is a challenging goal even with today's technology. Since link capacities continue to grow, the lookup time budget decreases.

Also, as a consequence of the Classless Inter-Domain Routing (CIDR), IP lookup can find multiple solutions (matching prefixes) for the given destination IP address [3]. The Longest Prefix Matching (LPM) rule is applied when multiple solutions are found [3]. LPM rule selects the prefix that has the longest match to a given destination IP address. Therefore, it is not enough to find a match during the lookup, but a found match must be the longest match as well. This makes the IP lookup a complex function because IP lookup is a two-dimensional problem because two prefix properties (prefix value and prefix length) need to be checked during the search process.

Most of the existing IP lookup algorithms have been developed to support IPv4 addresses, and many of them scale poorly to IPv6 addresses. However, transition to longer IPv6 addresses is inevitable. This transition increases lookup table size because of the larger IPv6 address space. The larger lookup table size makes more difficult the efficient IP lookup implementation. Therefore, IP lookup algorithms should have frugal memory requirements even when lookup table contains large number of entries (one million entries and beyond). Frugal memory requirements enable high-level of parallelization and use of fast on-chip memories, so faster IP lookup can be performed.

Network topology changes (router/link failure, traffic congestion in some parts of network) can be very frequent and they require lookup table updates. A lookup algorithm has to efficiently update lookup table entries. Otherwise, IP lookup process could be affected and as a consequence packets could be incorrectly forwarded or even dropped.

In this paper, we propose a novel IPv6 lookup algorithm Balanced Pipelined Lookup (BPL). BPL uses pipeline technique to achieve high lookup throughput and to support

high-speed links. BPL is based on the binary tree that is split into non-overlapping subtrees. The nodes of the subtrees are evenly distributed across the pipeline stages. Each stage contains similar number of nodes. In this way, each pipeline stage uses the same type of memory making hardware implementation very efficient. Only non-empty nodes of the subtrees are stored to reduce the memory requirements. Non-empty node is a node that contains an existing IP prefix. The main contribution of the paper is the proposed scalable BPL scheme suitable for fast and efficient IPv6 lookups. BPL has lower memory requirements compared to other similar tree based lookup schemes. Also, BPL supports efficient lookup table updates that do not decrease the lookup performance.

The paper is organized as follows. Section II presents the related work. We give special attention to the tree based lookup algorithms because BPL belongs to this class of IP lookup algorithms. Section III contains detailed description of BPL. Section IV presents performance analysis of BPL. Also, we compare the BPL with the lookup algorithms that use similar techniques as the BPL. Section V concludes the paper.

II. RELATED WORKS

IP lookup was recognized as router's critical function very early [3]. Many lookup algorithms have been proposed so far [2]–[29]. Most of the lookup algorithms are designed to support IPv4 addresses, but they usually do not scale well to longer IPv6 addresses. There is an ongoing effort to develop IP lookup algorithms that achieve efficient IPv6 addresses support [2], [7], [9]–[10], [12], [14], [16], [19]–[20], [22], [25]–[26], [29]. All lookup algorithms can be classified into three main categories [2]: tree based algorithms [3]–[22], Ternary Content-Addressable Memory (TCAM) based algorithms [23]–[25], and hash based algorithms [26]–[29]. We present the tree based algorithms related work in this section because BPL belongs to this category.

The binary tree represents a natural way to describe lookup table content [3]. Each prefix is represented as a node in the binary tree. Path to the node represents the prefix value. The binary tree is traversed using one corresponding bit from the given destination IP address in each step. Obviously, the main disadvantage of the binary trees is large number of steps in the worst case. Thus, IP lookup is very slow because many memory accesses are required in the worst case. Since IP lookup should be very fast, small number of memory accesses per one lookup is allowed (only one for high-speed links).

The first technique that was introduced to optimize the binary tree was the path compression that collapses one-way branch nodes to reduce number of memory accesses on such paths [3]. Multiple nodes with prefixes can be visited in the binary tree during one lookup. Thus, the best current match must be remembered during the tree traversal. To avoid the remembering of the best current match, leaf pushing technique is introduced [4]–[6], [18]. Leaf pushing technique pushes the prefixes from the internal tree nodes to leaf nodes, where a leaf node is the node without any child nodes. In this way, the lookup result can be obtained only in the leaf nodes. Downside of the leaf pushing technique is increased number of nodes in the tree.

To reduce the number of memory accesses during lookup, m-ry trees are introduced [3], [6], [8], [17], [21]. M-ry trees use a stride of m bits in each step, so the number of memory accesses in the worst case is m times lower than in the binary tree's worst case. Downside is that each node in m-ry tree has 2^m child nodes, i.e., 2^m pointers must be stored in each node of the m-ry tree. Stride size can be different in each step to optimize the structure of the m-ry tree. Some prefixes are not visible in m-ry tree because they belong to binary tree levels that are not visible in the m-ry tree. Therefore, these prefixes need to be pushed to the closest level that is visible in m-ry tree. In [17], Huffman coding is used to compress the lookup table entries into resulting m-ry tree structure. Memory requirements are decreased compared to classic m-ry tree solution, but the main problem of multiple memory accesses remains.

The bitmap technique was introduced to minimize the memory requirements of the lookup algorithm [2], [6]–[7], [15], [18], [20]. One way to use the bitmap technique is to reduce the number of pointers in the m-ry trees [6], [20]. Instead of 2^m pointers in a m-ry tree node, only one pointer and associated bitmap vector are used. Pointer points to a start location of a memory block that contains child nodes. Bitmap vector length is equal to 2^m and each bit corresponds to one child node. Bit in the bitmap vector represents the existence of the corresponding child node. In this way, memory size of one node in the m-ry tree is significantly reduced. Typically, memory block, where child nodes of one m-ry tree node are stored, is sized to store all child nodes. Memory block can be sized to contain only the existing child nodes to decrease lookup table memory requirements, but it is very complicated to efficiently manage the positions of these memory blocks in the memory.

The other way to use the bitmap technique is to replace the parts of the binary tree (subtrees) with bitmaps [7], [20]. Instead of storing complete subtree with all the subtree nodes and their pointers, the subtree can be presented via bitmap vector where each bit corresponds to one subtree node. A bit in the bitmap vector signals the state of the corresponding subtree node (empty or non-empty). In this way, memory requirements for the subtree storage are reduced, and the subtree search is faster because only the bitmap vector is inspected.

To achieve high speed lookups, the maximal number of memory accesses per one lookup should be only one. The pipeline technique with multiple memory instances needs to be used to virtually achieve this goal [4], [7], [13]–[14], [18]. There are still multiple accesses per one lookup. However, pipeline and multiple memory instances enable lookups for multiple destination IP addresses in parallel. Thus, it virtually seems like there is only one memory access per one lookup. For efficient hardware implementation, tree nodes need to be evenly distributed across the pipeline stages so that each stage contains the similar number of nodes [4], [14]. The even distribution of nodes enables the use of the same type of memories in the pipeline stages making the hardware implementation more

efficient. However, in these solutions, the first few stages have lower number of stored nodes. In [13], the imbalance problem of the early pipeline stages is avoided by using the randomization principle and circular pipeline. The circular pipeline enables to start searching at any pipeline stage and the randomization principle enables even distribution of nodes across all stages. However, the problem is a need for scheduling algorithm that schedules the starts of the requested IP lookups. Scheduling is needed to avoid the potential collisions during the parallel lookups that started in different pipeline stages.

One of the greatest problems of the tree based lookup algorithms is transition to IPv6 addresses [2]. The IPv6 prefixes are longer than the IPv4 prefixes. The number of empty nodes is very large in IPv6 case because the nodes in the earlier tree levels are empty. The large number of empty nodes leads to non-efficient memory usage and increases the memory requirements [2].

In [14], the multilayer B-trees containing only non-empty nodes are created. In this way, storage of empty nodes is avoided and memory requirements are decreased. Each node stores multiple prefixes, however, some free space should be left in the nodes for efficient updates. In [5], the priority tree structure is proposed. Priority tree eliminates the empty nodes in the tree structure. Priority tree is constructed from the binary tree in the following way. Each empty node is filled with the prefix moved from the leaf that represents the descendent of the corresponding empty node. If multiple descendent leaves exist, then the leaf that corresponds to the longest prefix is selected. After that, the selected leaf node is deleted from the tree and all its parent nodes are also deleted until the non-empty node is reached or the node that has a child on the other side is reached. By repeating this process, all empty nodes are eliminated from the tree. The main advantage is that the number of nodes in the priority tree is always equal to the number of prefixes, so the priority tree solution is very scalable. It is very easy to calculate the memory requirements for the priority tree because the priority tree does not depend on the prefix distribution. The downside is that each node in the priority tree must contain the prefix value because the priority tree path itself is not sufficient to determine the value of the prefix associated with the node. In addition, the depth of the priority tree is usually not significantly smaller than the depth of the original binary tree, so the multiple memory accesses problem still remains. The priority tree technique can be applied to m-ry trees as well [8]. In [16], a hierarchical-balanced search tree is constructed. This tree contains only non-empty nodes that store prefix range information. Memory requirements are very similar to the priority tree ones, and the same problem of multiple memory requirements is also present. In [19], priority trees are combined with B+ trees and index tables to utilize the prefix length distribution characteristics. Multiple priority subtrees are created and consequently tree depth is decreased.

Many novel solutions are adjusted and tuned for specific platforms [20]–[22]. In [20], IP lookup is adjusted for Phase-Change Memory (PCM) based memory system, and in [21], the proposed IP lookup solution utilizes the Graphics Processing Unit (GPU) parallel computing. IP lookup solution proposed in [22] exploits the Single Instruction, Multiple Data (SIMD) instructions.

III. BPL

The tree based IP lookup solutions usually have two major problems that impact the overall performance. The first problem is the large memory requirements because the empty nodes are stored. The empty nodes do not carry any useful information and they are only needed to enable the tree traversing. On the other hand, frugal memory requirements are desirable to fit the IP lookup solution on-chip because only on-chip memory would be used then. This would enable more efficient implementation that uses pipeline and parallelization techniques. In the case of IPv6 lookup, most of the prefixes belong to the range of 32–48 bits [30]. This means that there is a huge number of empty nodes that need to be stored, which limits the scalability of the tree based IP lookup algorithms that store the empty nodes.

The second problem is that the tree structure in its original form is not suitable for the pipeline technique. The reason is that the tree's structure is uneven, and typically more nodes reside at tree's bottom levels than in the tree's upper levels. In the case of a naive approach, each tree level would be one pipeline stage. This would create uneven memory requirements for the pipeline stages and pipeline implementation would be inefficient. Thus, a balancing of nodes should be performed to achieve the same memory requirements for all the pipeline stages.

Our proposed Balanced Pipelined Lookup (BPL) is based on the tree structure similar to priority tree structure [5] that stores only the non-empty nodes. This property eliminates the problem of empty nodes that most of the tree based solutions have. The tree that represents lookup table is divided in 2^K subtrees, where K represents the top K bits of the prefix. The non-empty nodes from all the subtrees are stored across the pipeline stages in a balanced way to achieve the same memory requirements for each pipeline stage. In this way, efficient pipeline implementation is enabled. In the remainder of this chapter, we denote non-empty nodes as nodes. BPL subtree structure is shown in Fig. 1. The example shows that the nodes of the subtree are distributed across the pipeline stages. Taking into account that all the subtrees are following the same principle, the number of nodes across the pipeline stages can be balanced. Thus, similar number of nodes in each pipeline stage can be achieved. This enables very efficient pipeline implementation of BPL. The children nodes of a subtree node are placed in the same pipeline stage as we explain in the following paragraph.

Note that each node in a subtree contains the prefix value and the right/left child pointer. The right and left child of a subtree node are stored in successive locations of the same pipeline stage in the case when both children exist. In this way, the memory requirements are decreased because only one pointer is used in each subtree node. During the lookup, the subtree is traversed using the bits of the destination IP address. In each visited node, IP address is compared to the prefix value stored in the node. When the end of the subtree

is reached, the last found positive match represents the longest matching prefix.

Figure 2 shows the architecture of our proposed BPL. IP lookup is very simple. The selector selects the subtree that should be searched. All subtrees are stored in the pipeline stages. The selected subtree is traversed and the longest matching prefix is found. The corresponding next-hop information (NHI) is retrieved from the next-hop memory (NHM) at the final step of the IP lookup process.

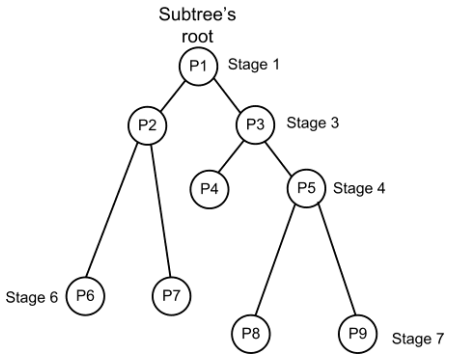

Fig. 1. BPL subtree structure.

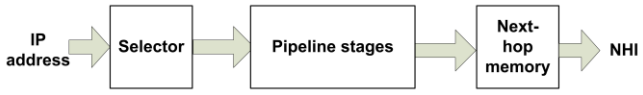

Fig. 2. BPL architecture.

The selector selects the subtree based on the top K bits of the destination IP address because the top K bits represent the common part for all the prefix values stored in the same subtree. We choose the value of $K = 16$ for two reasons. The first reason is that it provides sufficient number of subtrees for efficient balancing of nodes across the pipeline stages. The second reason is that typically there are no IPv6 prefixes shorter than 16 bits [30]. The selector comprises the memory that contains the pointers to the roots of all the subtrees. The pointer comprises two parts, the stage ID and the location address. The stage ID identifies the pipeline stage and the location address represents the location in the corresponding pipeline stage memory where the root node resides. The selector memory is addressed with the top K bits of the destination IP address and the pointer to subtree's root is retrieved. The selector also removes the top K bits from the IP address as they are no longer needed because all the prefixes in the selected subtree match these top K bits.

Each node in the subtree contains the prefix value and the left/right child pointer. The prefix value does not contain the top K bits as they are already inspected in the selector. If the node has both children, the left and right child nodes are stored in the same pipeline stage in the successive memory locations. Therefore, the pointer comprises a pointer value and a 2-bit indicator that indicates the existence of the left/right child node. The pointer value comprises two parts, the stage ID and the location address. During the lookup, subtree is traversed. In each visited node, the remaining part of IP address (the top K bits are removed by the selector) is compared to the stored prefix value. If there is a match, the stage ID and the location address of the matched node are remembered as the best solution. Each location in the NHM corresponds to one location in the pipeline (pipeline stage memories). In this way, the pointers to NHM locations are avoided and thus the memory requirements are reduced. Based on the stage ID and the location address of the longest matching prefix, the corresponding NHM location is determined and accessed to retrieve NHI as the final result of IP lookup. NHM stores only NHI.

Figure 3 shows the design of one pipeline stage. *Pointer_in* represents the address of the next node that should be visited. *Result_in* represents the current lookup solution. *IP_addr_in* represents the IP address without top K bits that are removed by the selector. *Bit_position_in* represents the position of the bit in the *IP_addr_in* that should be used for determining the next node (left or right child). All these inputs are delayed because the stage memory introduces read latency. *Location* part of the *Pointer_in* represents the stage memory read address. The delayed *Stage_ID* part is compared to hardcoded ID of the current pipeline stage. If there is no match, all input values are passed to corresponding outputs without any processing because the current stage does not contain the node of interest. If there is a match, the node of interest is accessed in the current pipeline stage and processing of the delayed input values and node's content is performed. The bit position value is incremented. IF_INC increments the pointer value if both child nodes exist and the right child should be visited next, otherwise the pointer is not incremented. The corresponding multiplexer is set to pass the new pointer value to *Pointer_out* output. If there is a match between the IP address and prefix value, we set the currently best solution to address of the visited node and set the corresponding multiplexer to pass this new solution to *Result_out* output.

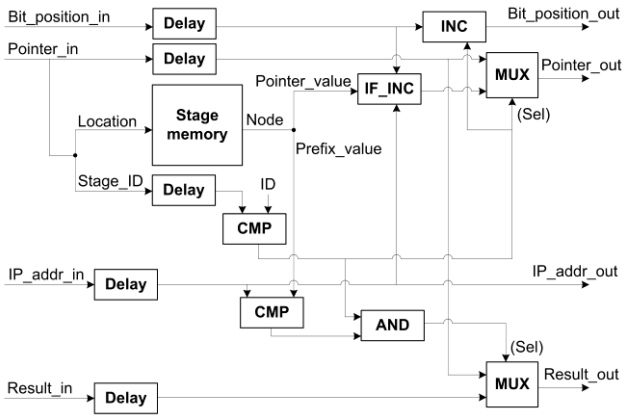

Fig. 3. Pipeline stage design.

We use incremental updates to update the lookup table. The algorithm for adding a new prefix is very simple. First, the top K bits of the prefix are inspected to determine the existence of the corresponding subtree. If the subtree is empty, the root node is created and the new prefix is placed in the root node. Note that prefixes stored in the subtrees are the original prefixes with the top K bits removed since all the prefixes in the same subtree have the same value of the top K bits. If the subtree is not empty, we traverse the subtree starting from the root node. The traversed path is determined by the bits of the new prefix (top K bits are omitted). In the traversal step i, we inspect the bit on position i to determine the direction of the next step. In each

visited node, we compare the length of the new prefix with the length of the prefix stored in the currently visited node. If the new prefix is longer or has equal length, then we just move to the child node determined by the value of the corresponding bit in the new prefix. If the new prefix is shorter, then we store the new prefix in the currently visited node, and the prefix that was previously stored in that node becomes the new prefix and we continue to traverse the tree. When the end of the path is reached (there is no corresponding child to traverse to), a new node is created and the new prefix is stored in the new node.

To achieve similar number of nodes in each pipeline stage, the update process evenly distributes the subtree nodes across the pipeline stages. When a new node is added, the leaf node in the corresponding subtree is created. If the parent of the created leaf node already has the other child, the new node will be added in the pipeline stage where that other child resides, because the child nodes must be in successive locations. Note that we reserve location for the missing child node when both children do not exist to simplify the update process. The justification for this is that the number of the non-leaf nodes that do not have both child nodes is negligible compared to the total number of the non-leaf nodes. If the parent of the created leaf node does not have the other child, the created node will be added in the least populated pipeline stage. If the least populated stage is located after the pipeline stage where the parent node resides, then the new node is simply stored in the least populated pipeline stage. However, if the least populated stage is located prior the pipeline stage where the parent node resides, then nodes on the path to the added leaf node are moved to prior stages as depicted in Fig. 4. An example in Fig. 4 shows that the least populated pipeline stage is stage 2. The prefixes P2, P3, and P4 are moved to the prior stages. The P2 is moved to the stage 2, while the P3 is moved to the place previously occupied by P2 in the stage 4. In the same way, the P4 is moved to the location previously occupied by P3 in the stage 6, while the P5 is stored in the location previously occupied by P4 in the stage 12. Note that Fig. 4 shows only the nodes on corresponding path of the subtree. However, the movement of the nodes comprises the movement of both nodes in successive locations. For example, when P3 as a right child moves to P2's old location (right child location), the P2's left child also moves to P2's old location (left child location). This does not disturb the remaining part of the subtree because the P2's left child points to stages latter than stage 4 if P2's left child has descendant nodes. This means that the update process does not disrupt the subtree structure and connectivity. As a result of the described update process, we are able to keep the even distribution of the nodes across the pipeline stages.

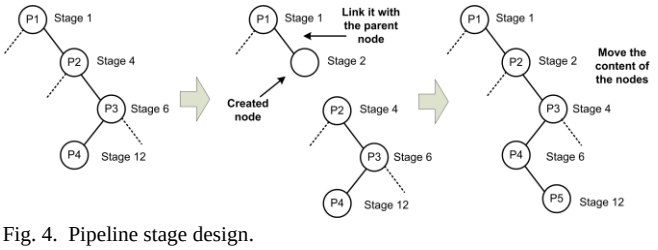

Fig. 4. Pipeline stage design.

The update part of the pipeline stage is not shown in Fig. 3 for the sake of simplicity. The update is very simple. The update data are pushed in the pipeline in the form of the address/data pair. The address represents the stage and the location where the data should be written. Since dual-port memories are used, the IP lookup is not interrupted with the update. The replica of the lookup table is stored in the control plane. In this way, all calculations are performed in the control plane, and the data plane receives only the aforementioned address/data pairs for the update process. This decoupling of the control and data plane enables the Software Defined Networking (SDN) support as well.

IV. PERFORMANCE EVALUATION

We compare our proposed lookup algorithm to several recently proposed tree based lookup algorithms. We select Linear Pipelined IPv6 Lookup Architecture (LPILA) [14], Hierarchical Balanced Search Tree (Hi-BST) [16], Splitting Approach to IP Lookup with Population Counting (SAIL-PC) [18], and Multilevel Length-based-classified Index Table (MLIT) [19] algorithms because each of them uses some of the techniques used in BPL (balancing technique, pipeline technique, and only non-empty nodes storage). To achieve high lookup throughput, the on-chip memories need to be used for the lookup table implementation. We compare the on-chip memory requirements of the lookup algorithms because the memory requirements of the lookup algorithm need to be small enough to fit the on-chip memory. We assume the NHI in all inspected lookup algorithms is stored in the external memory as the NHI is accessed in the last step of the lookup. Since IPv6 lookup tables are still not very large (~100 K prefixes [30]), we simulate the contents of the large IPv6 lookup tables using the concept of the FRuG tool [31].

Table I shows the on-chip memory requirements for the compared lookup algorithms for the table sizes 500 K–1500 K prefixes. The on-chip memory is a critical resource for the efficient IP lookup implementation. When the on-chip memory requirements are too large, the slower external memory needs to be used which negatively impacts the IP lookup performance. Table I shows that the proposed BPL has the lowest on-chip memory requirements. BPL has lower memory requirements than Hi-BST because the top K bits are omitted from the stored prefix values. In LPILA scheme, the multiple prefixes are stored in one B-tree node. LPILA has slightly larger memory requirements than Hi-BST and BPL, because a free space is required in B-tree nodes for fast updates. SAIL-PC is the second best solution in terms of on-chip memory requirements due to efficient utilization of bitmap technique and population counting technique that decreases the memory requirements for pointers. MLIT exhibits good performance for 500 K table size, however, the memory requirements significantly increase for the larger table sizes. Compared to the next best lookup algorithm, the BPL has around 27 % lower on-chip memory requirements that represent critical resource for the IP lookup implementation.

We have also tested BPL on the FPGA chip XC7VX980T from the Virtex7 family. For the lookup table of 753 K IPv6

prefixes, the design requires 6.44 MB (100 %) of the on-chip memory, 31 K (5 %) LUTs, and 13.5 K (1 %) regs. The achieved lookup throughput is the 120 millions of lookups per second. The critical resource is the on-chip memory. Larger IPv6 tables can be supported using the chips with larger on-chip memories. For example, the FPGA chips from Xilinx's Virtex UltraScale family have up to 15.8 MB on-chip memory [32]. Thus, our solution can support even lookup tables with 1500 K prefixes according to the results shown in Table I.

TABLE I. ON-CHIP MEMORY REQUIREMENTS.

	Algorithm	Table size		
		500 K	1000 K	1500 K
On-chip memory [MB]	BPL	4.39	8.75	13.04
	LPILA	7.97	13.81	19.91
	Hi-BST	6.37	12.99	19.85
	SAIL-PC	6.01	12.03	18.07
	MLIT	8.00	16.15	32.32

V. CONCLUSIONS

In this paper, we have proposed a novel lookup algorithm BPL. BPL combines several techniques to achieve high performances: pipeline, even distribution of nodes across the pipeline stages, and tree with the non-empty nodes structure. By comparing the BPL with the existing tree based IP lookup algorithms, we show that BPL solves the common problem of large on-chip memory requirements of the tree based lookup algorithms. BPL has 27 % lower on-chip memory requirements when compared to the next best tree based lookup solution. Using the pipeline technique, BPL achieves high lookup throughput of one lookup per clock cycle, thus supporting link capacities of 100 Gbps and beyond. BPL's even distribution of nodes across the pipeline stages is very attractive property for the efficient hardware implementation. Since only non-empty nodes are stored, BPL has frugal memory requirements. The frugal memory requirements enable the BPL to fit on today's FPGA chips. BPL supports very large IPv4 and IPv6 lookup tables. Update complexity of BPL is low, so BPL can support frequent network topology changes without negative effects on the IP lookup performance.

CONFLICTS OF INTEREST

The author declares that he has no conflicts of interest.

REFERENCES

[1] Z. Cica, "Analysis and implementation of packet processing functions in internet routers", in *Proc. of 2012 20th Telecommunications Forum (Telfor)*, Belgrade, Serbia, 2012, pp. 218–225. DOI: 10.1109/TELFOR.2012.6419186.

[2] A. Smiljanic and Z. Cica, "A comparative review of scalable lookup algorithms for IPv6", *Computer Networks*, vol. 56, no. 13, pp. 3040–3054, 2012. DOI: 10.1016/j.comnet.2012.04.027.

[3] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms", *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2001. DOI: 10.1109/65.912716.

[4] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup", in *Proc. of IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, Phoenix, USA, 2008, pp. 1786–1794. DOI: 10.1109/INFOCOM.2008.241.

[5] H. Lim, C. Yim, and E. E. Swartzlander, "Priority tries for IP address lookup", *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, 2010. DOI: 10.1109/TC.2010.38.

[6] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/Software IP lookups with incremental updates", *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, 2004. DOI: 10.1145/997150.997160.

[7] Z. Cica and A. Smiljanic, "Balanced parallelised frugal IPv6 lookup algorithm", *IET Electronics Letters*, vol. 47, no. 17, pp. 963–965, 2011. DOI: 10.1049/el.2011.0966.

[8] S.-Y. Hsieh and Y.-C. Yang, "A classified multi-suffix trie for IP lookup and update", *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 726–731, 2012. DOI: 10.1109/TC.2011.86.

[9] T. Ganegedara and V. Prasanna, "A high-performance IPV6 lookup engine on FPGA", in *Proc. of 2013 23rd International Conference on Field programmable Logic and Applications*, Porto, Portugal, 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645558.

[10] O. Erdem and A. Carus, "Clustered linked list forest for IPv6 lookup", in *Proc. of 2013 IEEE 21st Annual Symposium on High-Performance Interconnects (HOTI)*, San Jose, USA, 2013, pp. 33–40. DOI: 10.1109/HOTI.2013.11.

[11] K. Huang, G. Xie, Y. Li, and A. X. Liu, "Offset addressing approach to memory-efficient IP address lookup", in *Proc. of IEEE INFOCOM 2011*, Shanghai, China, 2011, pp. 306–310. DOI: 10.1109/INFCOM.2011.5935151.

[12] Y.-H. Yang, Y. Qu, S. Haria, and V. K. Prasanna, "Architecture and performance models for scalable IP lookup engines on FPGA", in *Proc. of 2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, Taipei, Taiwan, 2013, pp. 156–163. DOI: 10.1109/HPSR.2013.6602306.

[13] Y. Wu, G. Nong, and M. Hamdi, "Scalable pipelined IP lookup with prefix tries", *Computer Networks*, vol. 120, pp. 1–11, 2017. DOI: 10.1016/j.comnet.2017.03.017.

[14] M. M. Vijay and D. S. Punithavathani, "Implementation of memory-efficient linear pipelined IPv6 lookup and its significance in smart cities", *Computers & Electrical Engineering*, vol. 67, pp. 1–14, 2018. DOI: 10.1016/j.compeleceng.2018.02.044.

[15] T. Yang, G. Xie, A. X. Liu, Q. Fu, Y. Li, X. Li, and L. Mathy, "Constant IP lookup with FIB explosion", *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1821–1836, Aug. 2018. DOI: 10.1109/TNET.2018.2853575.

[16] T. Shen, X. Yu, G. Xie, and D. Zhang, "High-performance IPv6 lookup with real-time updates using hierarchical-balanced search tree", in *Proc. of 2018 IEEE Global Communications Conference (GLOBECOM)*, Abu Dhabi, United Arab Emirates, 2018, pp. 1–7. DOI: 10.1109/GLOCOM.2018.8647190.

[17] B. Indira, K. Valarmathi, and D. Devaraj, "A trie based IP lookup approach for high performance router/switch", in *Proc. of 2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, India, 2019, pp. 1–6. DOI: 10.1109/INCOS45849.2019.8951425.

[18] M. I. Islam and J. I. Khan, "SAIL based FIB lookup in a programmable pipeline based Linux router", in *Proc. of 2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, Xi'an, China, 2019, pp. 1–8. DOI: 10.1109/HPSR.2019.8808129.

[19] S.-Y. Hsieh, S.-J. Huang, and T.-H. Ho, "Multilevel length-based classified index table for IP lookups and updates", *Journal of Computer and System Sciences*, vol. 112, pp. 66–84, 2020. DOI: 10.1016/j.jcss.2020.04.001.

[20] C. Kim and H. Lee, "A high-bandwidth PCM-based memory system for highly available IP routing table lookup", *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 246–249, 2018. DOI: 10.1109/LCA.2018.2883461.

[21] K. Kaczmarski and A. Wolant, "GPU R-Trie: Dictionary with ultra fast lookup", *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, 2019. DOI: 10.1002/cpe.5027.

[22] Y. Ueno, R. Nakamura, Y. Kuga, and H. Esaki, "Fast longest prefix matching by exploiting SIMD instructions", *IEEE Access*, vol. 8, pp. 167027–167041, 2020. DOI: 10.1109/ACCESS.2020.3023156.

[23] J.-Y. Huang and P.-C. Wang, "TCAM-based IP address lookup using longest suffix split", *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 976–989, Apr. 2018. DOI: 10.1109/TNET.2018.2815999.

[24] W. Li *et al.*, "A power-saving pre-classifier for TCAM-based IP lookup", *Computer Networks*, vol. 164, Dec. 2019. DOI: 10.1016/j.comnet.2019.106898.

[25] R. Avazeh and N. Yazdani, "A new TCAM architecture for IP routing

with update complexity equal to O(1)", *Canadian Journal of Electrical and Computer Engineering*, vol. 43, no. 4, pp. 207–217, Fall 2020. DOI: 10.1109/CJECE.2019.2897277.

[26] B. Fradj, B. Wolff, N. Belanger, and Y. Savaria, "Implementation of a cache-based IPv6 lookup system with hashing", *in Proc. of 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, Florence, Italy, 2018, pp. 1–4. DOI: 10.1109/ISCAS.2018.8351362.

[27] L. Liu, J. Hu, Y. Yan, S. Gao, T. Yang, and X. Li, "Longest prefix matching with pruning", *in Proc. of 2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, Xi'an, China, 2019, pp. 1–6. DOI: 10.1109/HPSR.2019.8808125.

[28] H. Byun, Q. Li, and H. Lim, "Vectored-bloom filter for IP address lookup: Algorithm and hardware architectures", *Applied Sciences*, vol. 9, no. 21, 2019. DOI: 10.3390/app9214621.

[29] T. Stimpfling, N. Belanger, J. M. P. Langlois, and Y. Savaria, "SHIP: A scalable high-performance IPv6 lookup algorithm that exploits prefix characteristics", *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1529–1542, 2019. DOI: 10.1109/TNET.2019.2926230.

[30] IPv6 BGP Table Data. [Online]. Available: https://bgp.potaroo.net

[31] T. Ganegedara, W. Jiang, and V. Prasanna, "FRuG: A benchmark for packet forwarding in future networks", in *Proc. of International Performance Computing and Communications Conference (PCCC)*, Albuquerque, NM, 2010, pp. 231–238. DOI: 10.1109/PCCC.2010.5682304.

[32] Xilinx, Virtex UltraScale family. [Online]. Available: https://www.xilinx.com