FPGA Implementation of Lookup Algorithms

Zoran Chicha, Luka Milinkovic, Aleksandra Smiljanic
Department of Telecommunications
School of Electrical Engineering, Belgrade University
Belgrade, Serbia
zoran.cica@etf.rs

Abstract—**The pool of available IPv4 addresses is being depleted, comprising less than 10% of all IPv4 addresses. At the same time, the bit-rates at which packets are transmitted are increasing, and the IP lookup speed must be increased as well. Consequently, the IP lookup algorithms are in the research focus again because the existing solutions were designed for IPv4 addresses, and are not sufficiently scalable. In this paper, we compare FPGA implementations of the balanced parallelized frugal lookup (BPFL) algorithm, and the parallel optimized linear pipeline (POLP) lookup algorithm that efficiently use the memory, and achieve the highest speeds.**

Keywords - Internet router, IP lookup, FPGA

I. Introduction

Internet is still fast growing network. The number of hosts is still increasing and the IPv4 address space is almost exhausted. Actually, the Internet of "things" is being developed to include a tremendous number of sensors which might be attached to various machines and appliances. As a result of this development, transition to longer IPv6 addresses is inevitable. Packets generated by the increasing number of things on the Internet will be directed through the routers based on their IPv6 addresses. The output port (i.e. next-hop) of each packet is determined based on its IP address using the information from the lookup table according to the specified IP lookup algorithm. The lookup table contains forwarding information for the network addresses that a router learned from other routers in network. As the Internet is growing the lookup tables are getting larger. Classless network addresses are aggregated in these tables in order to consume a minimal amount of memory, and the longest prefix match of the given IP address should be found. The lookup table is typically split between the internal and the external memories. The internal (on-chip) memory is on the same chip as the lookup logic. The on-chip memory has a large throughput, which allows parallelization and pipelining that provide the high lookup speeds. However, the on-chip memory is very limited, and should be, therefore, carefully used. As the IP addresses get longer, the internal memory requirements of the lookup table can become a bottleneck. So, the existing internal memory should be used in a way that would maximize the lookup speed for the largest IP lookup tables.

In the literature, many lookup algorithms were proposed [1-8]. Most of the algorithms are based on trees, since the tree structure is a natural choice for a lookup table. However, search through ordinary binary trees is not fast enough, so there are many techniques to improve the lookup speed. Some of these techniques are the path compression, multibit trees [3], the level compression [8], bitmap techniques [1,3-5], leaf pushing [5,9], the priority tree technique [6], hashing [7] etc. The path compression is a technique where the paths in a tree are compressed if they have no branching. In this way, the number of the memory lookups is decreased. In multibit trees, one node has 2^m children as m bits are used for determining the child in the next level, and this technique reduces the tree depth. The level compression technique replaces the parts of the binary tree that are populated above some threshold with the multibit subtrees to efficiently reduce the depth of the tree. The bitmap technique uses the compact binary presentation of some parts of the tree (a subtree structure is presented with the bitmap vector whose positions correspond to the nodes in the subtree). It is usually combined with the technique that reduces the number of pointers in a multibit tree. Namely, only one pointer is kept in a node and it points to the first element of the vector of pointers to the node's children. Leaf pushing is used to push the next-hop information from the internal nodes of the tree to the leafs of the tree. The priority-tree technique fills empty nodes in the early levels of a binary tree with the longer prefixes. In this way the total number of nodes is reduced. Hashing is a popular technique used to reduce the number of memory accesses and increase the lookup speed. As one can see, there are many techniques that can be used to achieve higher lookup speeds, but usually they come with a price. For example, some internal nodes with the next-hop information can be masked with the leaf pushing technique or the multibit trees, so updates get more complicated than in ordinary binary tree, etc.

The parallel optimized linear pipeline (POLP) algorithm has been proposed in [2]. The main idea of POLP is to split the original binary tree into non-overlapping subtrees that are distributed across P pipelines which comprise similar numbers of nodes. Each pipeline is split into multiple stages, which comprise similar numbers of nodes. Nodes are associated to the stages so that the parent node must be in a different (and earlier) stage than its children nodes. In POLP, the pipeline is chosen based on the first I bits of the IP address. Then, the longest prefix is searched within the selected subtree. One memory is associated to each stage, and it is accessed when the search reaches that stage. Multiple IP addresses are processed simultaneously using the pipeline technique so that these parallel searches use different stages at any point of time. In this way high throughput is achieved.

This research was partially supported by Serbian Ministry of Science and Technology.

In this paper, we propose the balanced parallelized frugal lookup (BPFL) algorithm. BPFL is an extension of the PFL algorithm [1]. The advantage of BPFL is that it frugally uses the memory resources so the large lookup tables can fit the on-chip memory. As in PFL, the next-hop information is stored in the external memory, while the structure of the lookup table is stored in the on-chip memory. In this way, BPFL allows parallelization and pipelining that achieve faster lookups, as the external memory is accessed only once at the end of the lookup when the next-hop information is retrieved. The on-chip lookup table is organized as a binary tree divided into levels that are searched in parallel. Each level stores only non-empty subtrees. The subtree prefixes are stored in the corresponding balanced trees, unlike the PFL that stores the subtree prefixes in registers. Usage of balanced trees significantly reduces the number of required registers for lookup tables which enables the support for larger lookup tables. If the subtree prefix is found, the subtree bitmap vector is retrieved, and the node with the longest prefix match of the given level is found based on this subtree vector. In BPFL another improvement over the PFL is introduced, as for sparsley populated trees only indices of existing nodes are kept instead of complete bitmap vector. This significantly reduces total memory requirements. Since matches can be found at more than one level, the resulting match is the one found at the highest level. Then, the output port is found at the appropriate location of the external memory. Since the pipelining is used, one IP lookup can be performed per a clock cycle. The memory is used frugally by storing only non-empty subtrees, and by optimizing the bitmap vectors for sparsely populated subtrees. In this way, BPFL supports large IPv4 and IPv6 lookup tables.

In order to provide fast lookups, the IP lookup algorithms must be implemented in hardware. In this paper we present the FPGA implementations of the BPFL and the POLP algorithms. We chose these two algorithms because they consume the small amounts of the internal memory which becomes a critical resource in the case of IPv6 addresses. Finally, we compare the implementations of BPFL and POLP in terms of the resources that they consume, and the speeds that they achieve.

II. BPFL SEARCH ENGINE

The BPFL search engine will be described in this section. First, its architecture comprising multiple levels will be described. Then, the design of each level module will be explained. A module at each level contains two parts, the subtree search engine, and the prefix search engine. The subtree search engine finds the location of the subtree which contains the longest prefix in one of the balanced trees at the given level, provided that such the longest prefix exists. The subtree search engine then finds the longest prefix in the located subtree.

Generally, the IP lookup engine comprises the control logic and the lookup table. The lookup table contains the information about its structure, and the information about the output ports to which the incoming packets should be forwarded. These two types of information can be split, and stored in the internal and the external memories. Typically, the information about the lookup structure is stored in the internal memory so that the lookup speed could be maximized. It is important that this information consumes as little memory as possible, so that it can fit the on-chip memory even for very large lookup tables. For this reason, BPFL is designed so that it frugally uses the memory, and it particularly prudently uses the internal memory. Only the next-hop information with the output ports is stored in the external memory as it is accessed only at the end of the lookup process.

Figure 1 shows the architecture of the BPFL search engine which comprises multiple levels. The number of levels equals $L=La/Ds$, where La is the address length, and Ds is the subtree depth. Module of level i processes subtrees whose depths are equal to Ds, and which are determined by the $(i\text{-}1)\cdot Ds$ bits long prefixes. So, module of level i processes only first $i\cdot Ds$ bits of the the IP address, and finds the prefix whose length is greater than $(i\text{-}1)\cdot Ds$ bits and less or equal to $i\cdot Ds$ bits. The inputs of any module are the IP address and the signal *Search* that instructs the module to start looking for the longest prefixes. Modules of all levels pass their search results to the final selector. Here, a result is the location of the next-hop information in the external memory (*Next-hop_addr*). Signal *Match_found* is used to signal the search success. If modules at more than one level signal the successful searches, then, the selector chooses the result calculated at the highest level. Finally, the external memory is accessed to retrieve the next-hop information, i.e. the output port to which the packet should be forwarded.

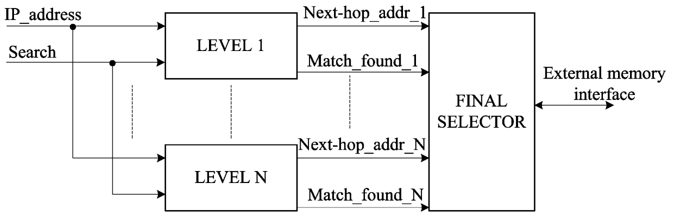

Figure 1. BPFL search engine top-level.

A module at level i contains two parts – the subtree search engine and the prefix search engine as shown in Figure 2. The subtree search engine processes balanced trees comprising the prefixes of non-empty subtrees, in order to find the subtree with the longest prefix of the given IP address, if such a subtree exists. The prefix search engine processes these non-empty subtrees of prefixes in the lookup table, in order to find the longest prefix of the given IP address. If the longest prefix match is found at the given level, the external memory address of the next-hop information is passed to the final selector shown in Figure 1.

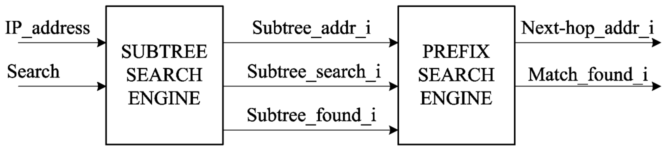

Figure 2. Module at level i.

The subtree search engine is shown in Figure 3. It consists of B non-overlapping balanced trees that store the subtree prefixes. Balanced trees are so called because each node in those trees has equal number of descendant nodes through the left and the right branch. A range of prefixes with the length equal to (i-1)·Ds is associated to one balanced tree at level i. A balanced tree is chosen by the balanced tree selector depending on the range of prefixes to which the IP address belongs. The balanced tree selector gives the address of the root of the selected balanced tree. The selected balanced tree is traversed based on the comparisons of its node entries to the given IP address. If the (i-1)·Ds long prefix of the IP address is greater than the subtree prefix stored at the current node, then, the next node (at the next level of the balanced tree) is the right node, otherwise, the next node is the left node. If the subtree prefix at the node equals the IP address prefix, the following balanced tree levels are just passed to enable pipelining while carrying the address of the balanced tree node with the IP address prefix and the balanced tree index. The subtree address is calculated using this node address. Signal *Found_j* is used to inform the succeeding balanced tree levels that the subtree has been found, while signal *Subtree_found* is used to inform the prefix search engine that the subtree has been found. In order to frugally use the on-chip memory, balanced tree nodes do not store pointers to their children. Instead, locations (addresses) of all nodes in each balanced tree are predetermined. For the sake of simplicity, the left child address is obtained by adding '0' before the parent's address, and the right child address is obtained by adding '1'.

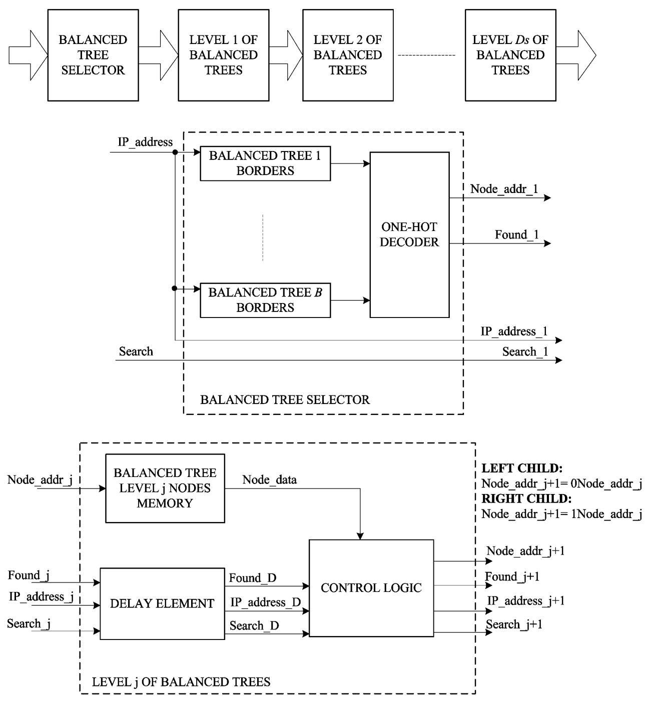

Figure 3. Subtree search engine at level i.

The prefix search engine is shown in Figure 4. It consists of the bitmap processor and the internal memory block. Elements of the complete bitmap vector are ones, if the corresponding nodes in a subtree are non-empty. However, if the subtree is sparsely populated, the bitmap vector would unnecessarily consume large number of memory bits. In this case, it is more prudent to store a list of indices of non-empty nodes. In our design, if the number of non-empty nodes is below the threshold, their indices are kept in the internal memory; otherwise, the complete bitmap vector describing the subtree structure is stored in the internal memory. In both cases, the external subtree address is stored in the internal memory together with its bitmap vector. The next-hop information related to the nodes of some subtree are stored in the external memory starting from the external subtree address. The prefix search engine finds the longest prefix of the IP address within the subtree based on either the list of non-empty nodes or the bitmap vector. The prefix search engine forwards this result to the final selector together with the signalization of the search success, and the external memory address where the next-hop information resides. The external memory address of the next-hop information is determined based on the external subtree address, and the index of the node that corresponds to the longest prefix of the IP address at the level in question.

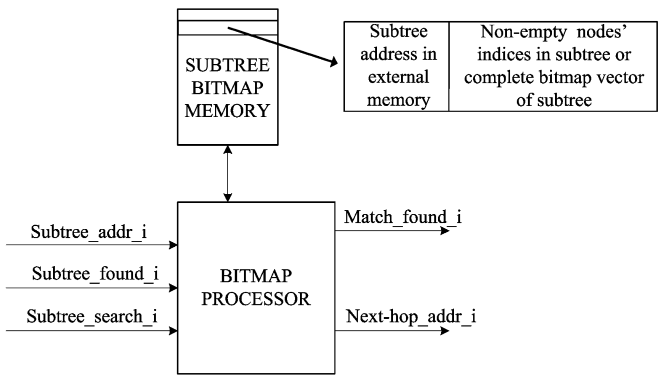

Figure 4. Prefix search engine at level i.

III. POLP SEARCH ENGINE

The POLP search engine will be described in this section. First, the architecture of the search engine will be described. The POLP search engine comprises multiple pipelines. The design of these pipelines will be explained as well.

In POLP, the original binary tree is split into non-overlapping subtrees. These subtrees are then split among multiple pipelines, so that these pipelines contain similar numbers of nodes, as shown in Figure 5. The pipeline is selected by the pipeline selector based on the first I bits of the IP address. Outputs of the pipelines are connected to the final selector. It selects the result from the activated pipeline, calculates the external memory location of the next-hop, and accesses the calculated location to retrieve the next-hop information.

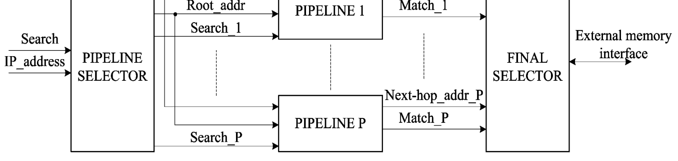

Figure 5. POLP search engine top-level.

The inputs of the pipeline selector are the IP address and the signal *Search* that activates the search engine as in the case of BPFL. The pipeline selector, then, activates the pipeline that holds the subtree of interest, and passes the relevant part of the IP address (remaining 32-*I* bits) to it as shown in Figure 5. The pipeline selector is a simple memory block addressed with *I* bits. Each location corresponds to one subtree, and contains the pipeline ID that holds the subtree, and its root node location. If the subtree does not exist, then, a specific constant is stored in corresponding location to signal that the subtree does not exist. The pipeline selector also holds the bitmap vectors for subtrees which are shorter than *I* bits, so that the lookup for short prefixes is performed by the pipeline selector. In the case of longer IP addresses, the output of the pipeline selector is passed to the selected pipeline. Activated pipeline searches the selected binary subtree based on the remaining 32-*I* bits. Since the pipelining is used, a new search can begin in the next clock cycle.

Each pipeline consists of *F* stages. Each node is assigned to one stage of the pipeline, while its children cannot be in the same stage to enable pipelining. Nodes are balanced over the stages, so that similar numbers of nodes are assigned to all stages. Children nodes of the given node do not have to be in the next stage. To enable this feature, a pointer in the subtree node holds the stage information where its child is placed, as well as its memory location. To avoid using of two pointers, we place the children of one node into two successive locations so that only one pointer is used. The pipeline structure is given in Figure 6. Each stage sends to the next stage the first 32-*I* bits of the IP address, the position of a bit in the subtree that is next to be processed, the stage and the location of the next node, the current result of the search, and the *Search* indicator that signals that the search is in progress. Signal *Next-hop_addr* carries the address of the next-hop information in the external memory of the best match found in the previous stages. Control signal *Match_found* denotes whether the information in *Next-hop_addr* is valid or not. The last stage in the pipeline will pass only these two signals to the final selector.

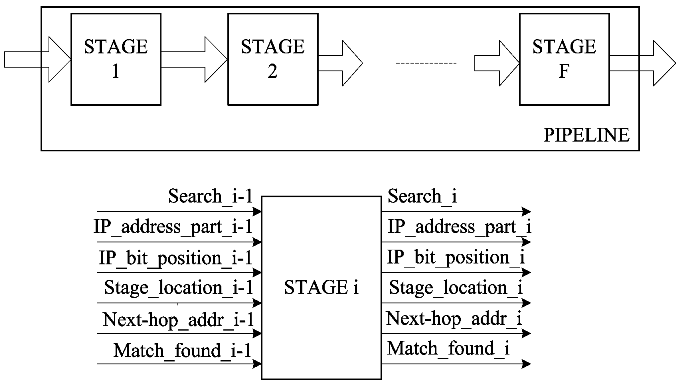

Figure 6. Pipeline structure.

Each stage contains a memory block that holds the nodes of the subtrees assigned to the pipeline under observation. One memory location contains the next-hop bit, the left-child bit, the right-child bit and the children pointer, shown in Figure7. The next-hop bit is set to '1' if the corresponding node holds the next-hop information, otherwise it is set to '0'. The left-child bit and the right-child bit determine the existence of the left child and the right child, respectively. Children pointer contains addresses of the stage and of the memory location of the children nodes. Besides the memory block, each stage contains also the control logic and the delay element. The control logic processes the data read from the memory or passes the signals from the previous stage to the next stage if the next node does not reside in the current stage. If the current stage is the addressed stage, and its node carries the next-hop address, this next-hop information should replace the next-hop address received from the previous stage. The next-hop address in the external memory is calculated based on the stage and the (internal) memory location of the node that carries the best match. The delay element is needed when the current stage is addressed (holds the node of interest) to delay signals from the previous stage as the memory read cycle introduces delay. The delay element is used even when the stage is not addressed to enable efficient pipelining without queues.

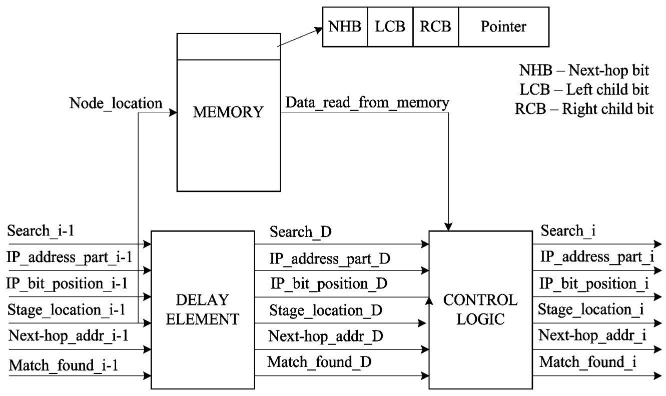

Figure 7. Stage *i* structure.

IV. Performance Analysis

In this section, we analyze the performance of the BPFL and POLP implementations. In the analysis we used three realistic lookup tables of different sizes (71K, 143K and 309K entries) to evaluate the performance of these two lookup algorithms for smaller and larger tables. The FPGA chip used for implementation is the Altera's Stratix II EP2S180F1020C5 chip [10]. The SRAM memory is used as the external memory. Since the existing IPv6 lookup tables are still small [11], we anticipated future large IPv6 lookup tables according the methodology proposed in [12]. The IPv6 lookup tables are derived from the existing IPv4 lookup tables [13]. Length of each prefix in the IPv4 lookup table is doubled, and 25% of them are moved to the closest odd number. Bits are added to the IPv4 addresses so that the desired density of non-empty subtrees in the IPv6 lookup table is achieved. The density of subtrees is set to be 2-4 times lower than in the IPv4 lookup tables because they occupy the larger address space.

TABLE I
Resource Usage for BPFL in the Case of IPv4

Table Size	LE	Memory [Mb]	SRAM [MB]	f_{max} [MHz]
71K	15.1K (11%)	1.93 (21%)	0.61	119
143K	19.4K (13%)	2.8 (30%)	0.8	113.6
309K	27.9K (19%)	5.6 (60%)	1.68	96.1

TABLE II
RESOURCE USAGE FOR BPFL IN THE CASE OF IPv6

Table Size	LE	Memory [Mb]	SRAM [MB]	f_{max} [MHz]
71K	52.1K (36%)	4.03 (43%)	0.32	110.5
143K	61.1 (43%)	6.33 (67%)	0.76	106.3
309K	89.7 (63%)	8.86 (94%)	0.95	99.9

Tables I and II show the chip resources required for the BPFL implementation in the case of IPv4 and IPv6 lookup tables, respectively. In both tables, the stride length is *Ds*=8, so that the IPv4 tables have up to four levels, while the IPv6 lookup tables have up to eight levels. One can observe that the chip resources are fully utilized only in the case of the largest IPv6 lookup table. So, the larger IPv4 tables can be supported by the selected FPGA chip, while for the larger IPv6 tables a more advanced FPGA chip would be needed. It can be observed that more internal chip resources are used in the IPv6 case, due to the longer IPv6 addresses. But, the SRAM memory requirements are lower in the IPv6 case, because more subtrees have the lower density, and, their storage is more efficient. Since pipelining is used, one lookup can be performed per one clock cycle. So, even in the worst tested case, the throughput of around 96 millions lookups per second was achieved.

TABLE III
RESOURCE USAGE FOR POLP IN THE CASE OF IPv4

Table Size	LE	Memory [Mb]	SRAM [MB]	f_{max} [MHz]	P	No of FPGA
71K	9.8K (7%)	8.81 (94%)	0.48	110.6	10	2
143K	6.6K (5%)	9.17 (98%)	0.76	107.6	9	3
309K	6.6K (5%)	9.17 (98%)	1.18	107.6	15	5

TABLE IV
RESOURCE USAGE FOR POLP IN THE CASE OF IPv6

Table Size	LE	Memory [Mb]	SRAM [MB]	f_{max} [MHz]	P	No of FPGA
71K	29.3K (20%)	3.4 (36%)	0.43	146.5	6	2
143K	10.4K (7%)	9.08 (97%)	1.9	123.2	9	9
309K	10.4K (7%)	9.08 (97%)	3.08	123.2	15	15

Tables III and IV show the chip resources required for the POLP implementation for IPv4 and IPv6 lookup tables, respectively. In our design we did not use *I*=8, like in [2]. We used *I*=16 to lower the total number of the stage memories as they are limiting the scalability of the POLP implementation on the FPGA chips. Because of its large memory requirements, the complete POLP design cannot fit one FPGA chip. For example, in the IPv4 case, one pipeline must contain at least *F*=17 stage memories when *I*=16 to fit the deepest possible subtrees. Size of the stage memory decreases when the number of pipelines increases, because the nodes are balanced over the pipelines and the stages. By changing the number of pipelines, the stage memories can be adjusted to better fit the available FPGA memory blocks. For each lookup table, we found the optimal number of pipelines in the POLP design for which the minimal number of the FPGA chips is needed, as shown in Tables III and IV. In both tables, the required chip resources are given per one FPGA chip, and the last two columns show the total number of pipelines, *P*, and FPGAs. The capacities of the external SRAM memories required for different lookup tables are also shown in tables.

It can be seen from Tables I, II, III, and IV, that the BPFL algorithm uses much smaller internal memory than the POLP algorithm. In addition, the memory blocks are better utilized in the case of the BPFL algorithm. As a result, the total on-chip memory requirements are significantly lower for BPFL than for POLP, so that the BPFL design can fit one FPGA chip, while the POLP design requires multiple FPGA chips which makes it costly and impractical. BPFL uses more logical elements due to the balanced tree selector that requires a large number of registers and comparators. However, logic elements required for the BPFL design can fit a single FPGA chip even for the largest IPv6 lookup table. The speeds of both algorithms are similar. The pipelines in POLP can serve multiple ports simultaneously, which would reduce the required number of FPGA chips per port. But then, a portion of the internal memory must be allocated to each pipeline for keeping the search requests. Also, in order to achieve full parallelization, the external SRAM memory should be allocated to each pipeline, which stores the next-hops corresponding to the prefixes in that pipeline. Since the POLP design requires multiple chips, it increases the complexity of the board design, and the overall router's cost.

TABLE V
COMPLEXITY OF THE WORST CASE UPDATE

Table Size	IPver	BPFL	POLP
71K	IPv4	8K	28K
	IPv6	16K	42K
143K	IPv4	8K	48K
	IPv6	33K	80K
309K	IPv4	16K	48K
	IPv6	65K	80K

Updates of the lookup tables are typically processed at the control plane of the router, and then, modified lookup tables are downloaded to the packet processors which are implemented in hardware. In this way, a complex update logic at the data plane for each search engine is avoided. The central processor can process the updates sequentially since they are much less frequent than the lookups. Updates in both algorithms typically have a moderate complexity. In BPFL, an update is easy in the case when a subtree already exists, and only its bitmap needs to be changed. When a subtree itself needs to be added or deleted, a balanced tree might need to be restructured, i.e. the nodes in the balanced tree might need to be moved to new positions. In the case of fully populated balanced trees, one fourth of nodes need to be moved on average. In the case of POLP, adding a new node can trigger a migration of multiple nodes to the earlier stages in order to provide sufficient number of stages for the path that includes the newly added node. The complexity in this case depends on the subtree size. Also, when the first prefix of some subtree needs to be added to the lookup table, it would require *F*/2 memory accesses on average, where *F*/2 is the number of stages. Finally, in order to keep the pipelines balanced, subtrees might need to be moved from pipeline to pipeline. The processing complexities in mentioned realistic cases, are similar for POLP and BPFL algorithms.

The worst case update of the lookup table in BPFL is when there is only one empty node in the subtree search engine so that all the nodes must be moved when a new prefix is to be added at the given level. The processing complexity in this worst case comprises $(2^{D_b}-1)\cdot B$ memory accesses, where D_b is the balanced tree depth in the level with the largest number of nodes, and B is the total number of balanced trees in the same level. The worst case for updates in POLP is when two largest subtrees need to exchange their places in two different pipelines so that a new prefix could be added to one these subtrees. We estimate the size of these large subtrees to be $(F\text{-}\log_2 N_n+1)\cdot N_n$. Namely, the upper levels of these trees double in each stage, until they reach the size of the stage memory; then, the lower levels comprise N_n nodes. So, the worst case processing complexity comprises $2\cdot(F\text{-}\log_2 N_n+1)\cdot N_n$ memory accesses when POLP is used. It should be noticed that in both algorithms, the probability of the described worst cases is very low. Table V gives the worst case complexity for the lookup tables used in this paper. It can be seen that BPFL has the lower worst case complexity of the lookup table updates. However, the described worst cases are not very likely, while in more probable cases the update complexities are significantly lower for both algorithms.

V. CONCLUDING REMARKS

In this paper we presented FPGA implementations of two lookup algorithms, POLP and BPFL, and compared them. FPGA chips are attractive devices for implementation of the data plane functionalities because of their speed and flexibility. Both algorithms achieve high lookup speeds. However, our BPFL algorithm more scalable than the recently proposed POLP algorithm. Namely, POLP requires multiple chips even for the lookup tables of a moderate size, while BPFL requires only one FPGA chip for the largest lookup tables on the Internet. Since the implementation complexity of the BPFL algorithm is fairly low, it is a promising lookup algorithm which can support the IPv6 addresses that are spreading on the Internet.

REFERENCES

[1] Z. Čiča, A. Smiljanić, "Frugal IP Lookup Based on Parallel Search, " *15th International Workshop on High Performance Switching and Routing 2009*, Paris, France, June 2009.

[2] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," *Proc. of IEEE INFOCOM*, April, 2008.

[3] D. Taylor, J. Lockwood, T. Sproull, J. Turner, D. Parlour, "Scalable IP Lookup for Programmable Routers, " *Proc. IEEE INFOCOM 2002*, vol.21, no.1, pp.562-571, June 2002.

[4] D. Pao, C. Liu, A. Wu, L. Yeung, K.S. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup, " *IEE Proceedings on Computers and Digital Techniques*, vol.50 , no.1, pp.43-52, January 2003.

[5] R. Rojas-Cessa, L. Ramesh, Z. Dong, L.Cai, N. Ansari, "Parallel-Search Trie-based Scheme for Fast IP Lookup," *GLOBECOM 2007*, pp. 26-30, November 2007.

[6] H. Lim, J. H. Mun, "An Efficient IP Address Lookup Algorithm Using a Priority Trie," *GLOBECOM 2006*, pp. 1-5, November 2006.

[7] M. Bando, N.S. Artan, J. Chao, "FlashLook: 100-Gbps Hash-Tuned Route Lookup Architecture, " *15th International Workshop on High Performance Switching and Routing 2009*, Paris, France, June 2009

[8] S. Nilsson, G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE JSAC*, vol. 17, no. 6, pp. 1083–1092, June 1999.

[9] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," *Proc. ACM Sigmetrics 98*, June 1998.

[10] www.altera.com

[11] http://bgp.potaroo.net

[12] M. Wang, S. Deering, T. Hain, L. Dunn, "Non-random Generator for IPv6 Tables," *Proc. of IEEE. Symposium on High-Performance Interconnects*, pp. 35-40, August 2004.

[13] http://psp1.iit.cnr.it/~mcsoft/ast/ast_data.html