Updating Designed for Fast IP Lookup

Natasa Maksic, Zoran Chicha and Aleksandra Smiljanić
School of Electrical Engineering
Belgrade University, Serbia
Email:maksicn@etf.rs, cicasyl@etf.rs, aleksandra@etf.rs

***Abstract*—Internet traffic is rapidly increasing, as well as the number of users. The increased link speeds result in smaller available time for the lookup and, hence, require faster lookup algorithms. There is a trade-off between complexities of the IP lookups and the IP lookup table updates. In this paper, we propose, implement and analyze lookup table updating for parallel optimized linear pipeline (POLP) lookup algorithm and balanced parallelized frugal lookup algorithm (BPFL). We compare POLP and BPFL update algorithms in terms of their execution times for real-world routing tables. In order to analyze the influence of updates on packet forwarding, we will observe the number of memory accesses when the lookup tables are updated due to the network topology changes. For both lookup algorithms, we measure the memory requirements as well. Our analysis will show that the BPFL update algorithm has the smaller memory requirements, while the POLP update algorithm is faster.**

I. Introduction

IP lookup has always been a critical function of the Internet routers, since routers have to decide very quickly (in several tens of nanoseconds) about the port to which the packet should be forwarded to, based on its IP address. The needed lookup speed can be achieved through the parallelization, and pipelining. But, these techniques require the large number of memories that would be accessed in parallel. This is only practical when internal on-chip memories are used.

A number of distinctive concepts have been developed for the IP lookup design [1]-[6], offering different trade-offs between lookup speed, implementational complexity and memory usage. In this paper, we focus on the BPFL [1]-[2] and POLP [3] algorithms, because of their high lookup speeds. Here, the lookup speed is the number of lookups that can be performed in a second. Both of these algorithms use high levels of parallelization and pipelining to achieve the high lookup speeds. POLP uses simple binary trie which is searched in parallel, while the BPFL groups prefixes into levels based on their lengths and searches the levels in parallel. The BPFL lookup algorithm that we recently proposed minimizes the on-chip memory required for the given speed [1]. BPFL minimizes the memory requirements in several ways: using balanced trees, static addressing and bitmaps. Lower memory requirements allow both the higher scalability and, through the use of on-chip memories, the higher level of parallelization, and, therefore, the higher speed.

It has been recognized that there is a trade-off between the IP lookup and the update of lookup tables. Typically, faster IP lookup algorithms imply more complex update algorithm. Performance of the lookup table updating is often treated in less details than the IP lookup itself. However, efficiency of the update algorithms affects the implementational cost, and, it might cause interruptions of the IP lookup process. In order to fully evaluate our novel BPFL, we will comprehensively analyze its update algorithm, and compare it with the POLP update algorithm.

The update algorithm specifies the preparation of the lookup table used for the IP address lookup. Consequently, the update algorithm is determined by the lookup algorithm. New or deleted routes are provided to the update module by the software modules that execute dynamic routing protocols such as OSPF or BGP, or they are entered by an operator or loaded from configuration files as static routes.

Updating needs to be able to follow the speed of dynamic routing protocols which is several orders of magnitude slower than the speed of packet forwarding through the router. Consequently, lookup and update algorithms are designed so that as much complexity as possible is moved to the update algorithm. Also, because of more relaxed execution timing requirements, the update algorithms are typically implemented in software. Although the timing requirements of the update algorithm are more relaxed than the requirements of the lookup algorithm, they are still very strict and cannot be ignored. Also, interruptions of packet forwarding caused by the lookup table updates should be kept minimal, while maintaining the low memory requirements and the low implementational complexity.

The first update procedure that we propose, implement and analyze in this paper is the update algorithm that corresponds to the BPFL lookup algorithm [1]. We will name it the BPFL update algorithm. The second update algorithm that we will implement and analyze corresponds to the POLP lookup algorithm, presented in paper [3]. Our implementations are written in C++ and have modular design which enables their integration into the routing software packages such as Xorp and Quagga. It can also be easily ported to C for the development of kernel components. Our implementations were included into the OSPF software written by Moy [7] that was used for the network simulations. We have analyzed the performance of the algorithms using the large real-world routing tables.

This paper is organized as follows. The second section briefly describes BPFL and POLP lookup algorithms. In the third section, the BPFL update algorithm and the POLP update algorithm are presented. In the fourth section, we measure the memory requirements of these two lookup algorithms.

Execution times of the update algorithms are measured, and the results are presented in the fifth section. The sixth section examines the number of accesses to the lookup memory for the analyzed algorithms. The paper is concluded in the seventh section.

II. IP Lookup Algorithms

In this section we will give brief descriptions of the BPFL and POLP lookup algorithms. We will focus on the structures of the lookup tables in these algorithms, since they determine the corresponding update algorithms.

A. Balanced Parallelized Frugal Lookup Algorithm

The structure of the BPFL lookup table is shown in Fig. 1. In BPFL, prefixes are split into L levels based on their lengths. Each level contains prefixes whose lengths are between $(i-1)\cdot D_s$ and $i\cdot D_s$. Here, $D_s = L_a/L$, where L_a denotes the number of bits in the IP address. At each level, information about prefixes is kept in balanced trees and subtrees.

At level i, nodes of balanced trees contain prefixes with lengths $p_i = (i-1)\cdot D_s$. A subtree corresponds to each node of a balanced tree, and its structure is stored at the memory location determined by that node. The remaining prefix bits of the IP address determine the node of this subtree. The output port is determined by the level, and the node of the balanced tree and the subtree that correspond to the given IP address. If the match is found at multiple levels, the one from the highest level is taken as a solution since it is longest.

A sorted array keeps the prefix ranges associated to the balanced trees, as depicted in Fig. 1. All balanced trees have equal depths, equal to D_b. The memory location of the balanced tree root is determined by the prefix range to which the p_i long prefix of the given IP address belongs. Then, the selected balanced tree is searched for the subtree prefix that matches the IP address. At each node of the balanced tree, the IP address prefix is compared with the subtree prefix of that node. If the IP address prefix is larger, the search is continued through the left branch, and vice versa. When the subtree prefix equals the IP address prefix, the search is completed, and the subtree address is determined based on the balanced tree node with the subtree prefix. The balanced tree search is pipelined since each level of the balanced tree is stored in separate memory.

Subtrees are D_s bits deep. We will consider the case in which $D_s = 8$. Indices of existing prefixes are kept in an array if their density is below a certain threshold. If the density of IP addresses is above the given threshold, the subtree is represented by the bitmap vector with ones denoting existing prefixes.

BPFL saves the internal memory as a critical resource in several ways. Firstly, no information is stored about empty subtrees without prefixes, which most of the previously proposed lookup algorithms do. Secondly, nodes in balanced trees have fixed memory locations, which saves the memory since the pointers are not needed. Thirdly, the subtree structures are stored in an optimized way, so that their memory consumption is minimized. With these techniques, BPFL requires significantly less memory than other analyzed lookup algorithms for both IPv4 and IPv6 lookup algorithms [2]. The BPFL savings of memory increase with the lookup table size. BPFL was shown to require almost 8 times less memory than the POLP lookup algorithm, for large IPv6 table with 300K entries [1]. Thanks to this property, the entire structure of very large lookup tables as specified by BPFL can fit on a single state-of-the-art FPGA chip. Consequently, a high lookup speed can be achieved when BPFL is used, due to the high level of parallelization that is provided on chip.

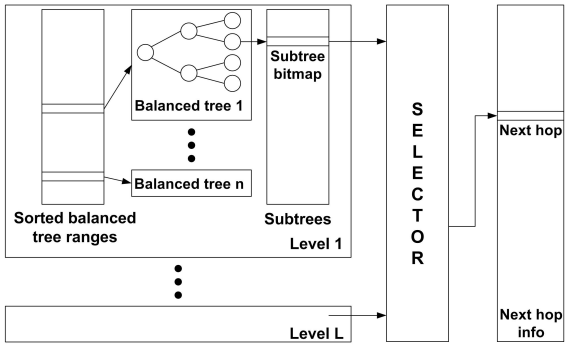

Fig. 1. BPFL data structures

B. Parallel Optimized Linear Pipeline

The POLP lookup algorithm has been introduced in [3]. The POLP specifies the lookup table structure that is depicted in Fig. 2. The POLP algorithm uses the leaf-pushed unibit-trie. The lookup process is parallelized and pipelined. Multiple IP addresses can be searched in parallel through distinct pipelines.

In POLP, the leaf-pushed unibit-trie that stores all routes is divided into subtries differentiated by the first I bits. These subtries are assigned evenly to the pipelines, so that the pipelines have the similar numbers of nodes. Subtries are first sorted in a descending order with respect to their numbers of nodes, and, then, the subtries are assigned in this order. Each subtrie is assigned to the pipeline with the smallest number of nodes, when its term to be assigned comes. A selector determines the pipeline and the subtrie root, based on the first I bits of the IP address.

Multiple subtries of one pipeline can be searched in parallel, only if they access different memories at any point of time. Nodes in the pipelines need to be stored in different memories to allow such pipelining. A stage comprises nodes stored in one memory. In the most straightforward design, nodes of the same level in all subtries belong to the same stage. Alternatively, prefixes can be evenly distributed across the stages so that they require the memories of the same size.

Obviously, POLP was designed to achieve a high level of parallelization. However, pointers to the children nodes have higher memory requirements than the bitmap representation used in BPFL. Such a large memory required by POLP cannot be placed on chip for large lookup tables [1]. Using multiple external memories significantly complicates the design.

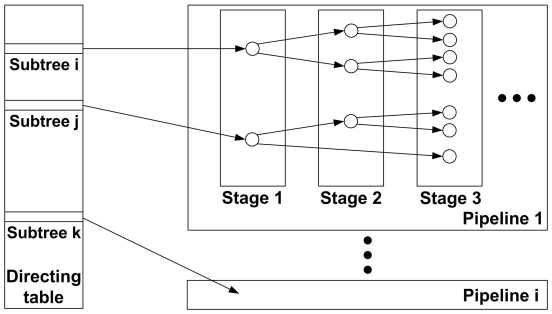

Fig. 2. POLP data structures

III. Algorithms for the Lookup Table Updating

The BPFL lookup algorithm has an obvious advantage over the previous proposals in terms of scalability, since it requires significantly lower on-chip memory [1]. It is important to analyze if the BPFL update algorithm is practical as well. In this section we will describe the BPFL update algorithm, as well as the POLP update algorithm.

A. BPFL Update Algorithm

In BPFL, initially, the ranges related to the balanced trees of level i are equal to the $(i-1) \cdot D_s$ long prefixes of the incoming routes as long as their number is smaller than the maximum number of balanced trees on that level. Afterwards, the prefix is added either to the closest range, and it becomes one of its bounds, or to the range to which it belongs. The range will also change if its balanced tree is overflowed.

Algorithm 1 describes a route addition in BPFL. Function $find\_tree()$ performs a binary search for the balanced tree to which a new route should be added. It chooses either the balanced tree whose prefix range includes the new prefix, or the balanced tree with the closest range. If the chosen tree is not full, the new route is added to it. Then, if the neighbouring balanced tree has at least two empty spaces more than the balanced tree of the prefix, the corresponding boundary node from the original tree is moved to this neighbouring tree, in order to maintain uniform distribution of the nodes across the balanced trees. If the balanced tree of a new prefix is full, one of its boundary nodes is moved to the neighbouring tree. The neighbour is selected by function $min\_side()$ that returns the neighbouring tree on the less populated side of the given tree. In this way, nodes are distributed evenly across the balanced trees, and the later rearrangements of the balanced trees are reduced. Function $move\_node()$ recursively moves nodes to the neighboring trees until it finds the tree which is not full. If it finds such a tree, the closest prefix is added to it.

Function $add\_prefix()$ in Algorithm 1 adds the route into the balanced tree. Balanced trees are complete trees, namely, all levels of a balanced tree are completely filled, except possibly the last level that is filled from left to right. Therefore, addition and deletion of nodes involve moving nodes until the tree is filled in a described way. After the addition of a node to the balanced tree, the corresponding subtree at the location determined by the added node is updated and the corresponding next-hop information is stored. At the end of the route addition, function $update\_tree\_ranges(tree)$ is needed to set new bounds for the balanced tree ranges.

Algorithm 1: Route addition in BPFL

$tree \leftarrow find\_tree(prefix)$
if $tree\_full(tree) = false$ **then**
  $add\_prefix(tree, prefix)$
  for nbr in $neighbor\_tree(tree)$ **do**
    if $node\_num(tree) > node\_num(nbr) + 1$ **then**
      $move\_node(tree, nbr)$
    end if
  end for
else
  if $move\_node(tree, min\_side(tree)) = true$ **then**
    $add\_prefix(tree, prefix)$
  end if
end if
$update\_tree\_ranges(tree)$

Modification of an entry in the lookup table is similar to the lookup process. First, based on the network prefix, the level of the prefix is determined. Then, the location of the next-hop information that corresponds to the given prefix (IP network address) is found according to the BPFL lookup algorithm, and, finally, this next-hop information is changed. In order to delete some entry of the lookup table, first, the location of the next-hop information needs to be found. Then, the next-hop information is deleted, as well as the corresponding entry from its subtree. If this is the last entry of the subtree, then the subtree prefix should be removed from the corresponding node of the balanced tree. In that case, the nodes of this balanced tree need to be rearranged in order to maintain its completeness.

B. POLP Update Algorithm

In [3], a heuristic has been proposed for distributing the nodes as evenly as possible across the stages, taking into account their heights (largest distance from the leaves) and the numbers of their descendants. Nodes are assigned to the stages, so that the child nodes are always in one of the stages after the parent nodes. Stage 1 stores the information about root nodes of subtries. Nodes are stored in the first available empty stage so that their number is smaller than the quotient of the remaining nodes and the remaining number of stages.

Every time the trie is changed according to the POLP algorithm, both the assignment of subtries to the pipelines, and the assignment of subtrie nodes to the stages need to be revised. In the case of frequent changes in the network, the calculation of these assignments according to the algorithm proposed in [3] would require high processing power. In this algorithm, subtries are assigned to the pipelines in the decreasing order of their sizes, and nodes are assigned to the stages in the decreasing order of their heights. However, prefixes are added or deleted to the lookup tables as the network topology changes, and not in these particular order.

For this reason, a more efficient POLP update algorithm needs to be designed which will be better suited for incremental network topology changes. One solution for this problem is presented in [8], in which incremental update algorithm for uniform node distribution across the stages is analyzed.

Since the algorithms for uniform node distribution across the stages have been examined in the literature, we turn our attention to non-uniform node distribution, in which attempt is made to assign nodes of the same level to the same stage. This reduces pipeline conflicts which occur when two lookup processes try to access the same stage memory, resulting in faster lookup. Stage memories are adjusted to the prefix length distribution on the Internet [9].

Addition of a new prefix is based on the trie traversal according to the bits of the new prefix. If a leaf is encountered, it is remembered for the purpose of leaf pushing, and new nodes are created until prefix bits are exhausted. If the depth of a newly created node is I, a new subtrie is created and assigned to the pipeline with the smallest number of nodes. After the creation of new nodes, leaf pushing is performed starting from remembered node. If the nodes for all bits were already created, leaf pushing would be performed starting from the last node on the traversal path.

Algorithm 2: Addition of a node to the pipeline

$free\_slot = -1$
if $depth = I$ **then**
  $stage\_num = 1$
  $free\_slot = free\_stage\_slot[1]$
else
  $parent\_stage = node.parent.stage\_num$
  for $i = parent\_stage + 1$ **to** $stages\_num$ **do**
    if $free\_stage\_slot[i] > 0$ **then**
      $stage\_num = i$
      $free\_slot = free\_stage\_slot[i]$
      break
    end if
  end for
end if
if $free\_slot > 0$ **then**
  $node.stage\_num \leftarrow stage\_num$
  $node.stage\_slot \leftarrow free\_slot$
  $find\_free\_slot(stage\_num)$
  $node.write\_parent()$
  $node\_num = node\_num + 1$
end if

Addition of new node to the pipeline is performed using Algorithm 2. For every stage, one empty slot is kept in the $free\_stage\_slot$ array, if it exists. A node is assigned to the first stage with an empty slot starting after the stage of its parent node. A new free slot in the selected stage is found using the function $find\_free\_slot()$. Function $write\_parent()$ writes the information about new node's location into the parent's stage slot. The number of nodes $nodes\_num$ in the corresponding pipeline is incremented, and the sorted list of pipelines is updated accordingly.

Route deletion traverses the leaf-pushed trie based on the destination network prefix that should be deleted. When the node is found, all leaf nodes which correspond to this prefix will be deleted.

If a new route cannot be accommodated using described procedure, and a skipped stage with an empty slot exists on a path of this route in its pipeline, we apply node shifting procedure. Nodes are shifted up, if possible, towards the emptied slot in the skipped stage.

If node shifting cannot provide addition of a new route, we apply recalculation of the pipeline, using procedure similar to one described in [3], with the difference that stage is filled until it reaches the quotient of the remaining nodes and the remaining number of free slots multiplied by memory size of the current stage. If the pipeline recalculation fails, a complete recalculation of all pipelines is performed. It should be noted that, longer prefixes could be stored with higher probability by using additional stages after $(32 - I)^{th}$ stage.

IV. Memory Requirements

In this section, we will confirm the main advantage of BPFL which is its frugal memory consumption, using our developed algorithm for updating the BPFL lookup tables. We used a set of 21 real lookup tables, different from the ones used in [1].

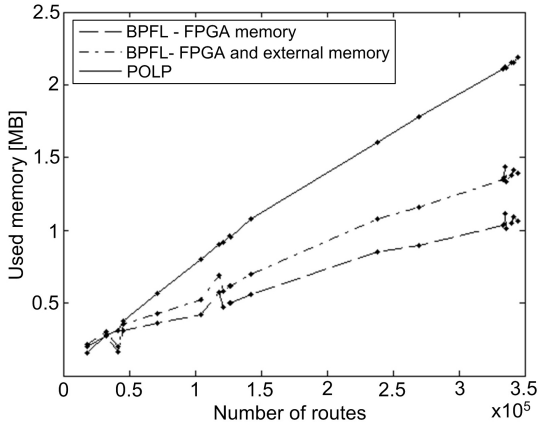

Fig. 3. Measured memory for real-world routing tables

We assessed the memory requirements of POLP and BPFL using real-world routing tables collected from [10]. We measured the number of nodes in the leaf-pushed trie and the number of nodes in each of the balanced trees. Based on that information, we calculated used memory as a function of the number of routes shown in Fig. 3. The dashed line represents the memory used by BPFL for storing the lookup table structure, and the dashed-dot line represents the total memory used by BPFL, which includes the next-hop information. The full line represents memory used by nodes of the POLP algorithm, for the case in which 18 bits are used to address the children. The next hop information is stored in the leaf nodes. The results in Fig. 3 show that BPFL uses scarce on-chip memory much more frugally than POLP which confirms the results in [1].

V. Execution Time

The performance of BPFL and POLP updating will be compared in terms of their execution times.

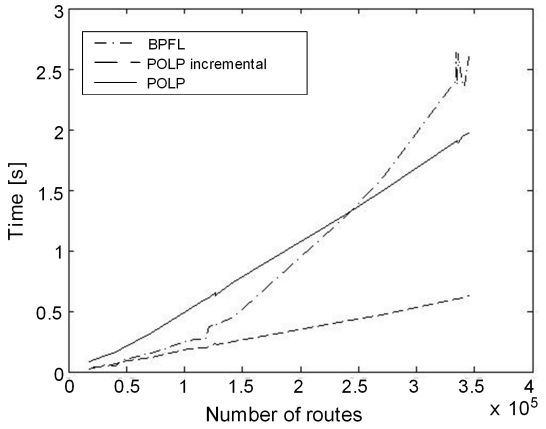

Fig. 4. Execution time

The dashed-dot line and the dashed line in Fig. 4 show execution time for iterative addition of entire real-world routing tables of different sizes for BPFL and POLP update algorithms, respectively. All prefixes from the routing table were added one by one, and lookup tables were updated after the addition of each route. The full line in Fig. 4 shows the time required to build the POLP lookup table as in [3].

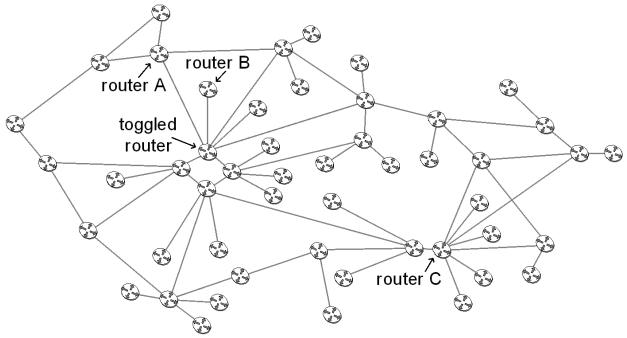

Fig. 5. Simulation network

It can be observed from Fig. 4 that incremental POLP update algorithm uses less time to add all prefixes to the routing tables than the BPFL algorithm. Faster increase of BPFL execution time occurred because larger routing tables resulted in larger number of operations needed to maintain balance property of the tree. Also, when a node needs to be added in a full tree, transfer of nodes between trees additionally increased execution time, as described in Section 3. However, times required for building very large lookup tables is acceptable for both the POLP and the BPFL update algorithms.

VI. Lookup Disruption by Updates

Important performance measure of the lookup table update algorithm is the number of memory accesses that it requires. While the memory content is updated, lookup for the incoming packets could be affected. So, the lookup update algorithm is better if it requires smaller number of memory accesses. In order to examine the number of memory accesses that the POLP and BPFL update algorithms require, we integrated our implementations of these algorithms into the OSPF program created by John Moy [7]. For simulation of the OSPF network, we used ospfd_sim program which is the part of the OSPF implementation [7]. Simulation network is shown in Fig. 5. Topology of the simulation network is inspired by the Czech educational and scientific network CESNET [11]. The simulation network consists of fifty routers, with a local area network attached to each router. Routers were started at the beginning of the experiment, and, in the 30^{th} second of the measurement, router depicted as "toggled router" was turned off. In the 50^{th} second of the measurement this router was started again. Fig. 6 and Fig. 7 show the numbers of accesses to the lookup memories of the three routers, depicted with letters A, B and C. It can be noticed, that the accesses occurred at the beginning of the experiment, when the routers were started, and than again after the 30^{th} and 50^{th} second of the measurements, when the "toggled" router was turned off and than turned on again.

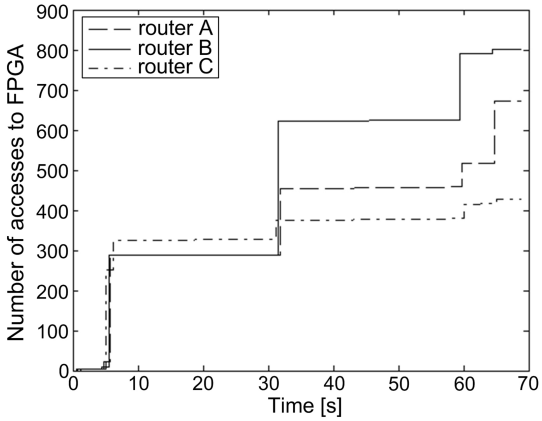

Fig. 6. Number of accesses to lookup memory for BPFL update algorithm

Fig. 6 shows the number of memory accesses when the BPFL algorithm is applied. The number of memory accesses is depicted for three routers in the simulation network, which are marked in Fig. 5. Router B had the larger number of memory accesses in the second half of the measured time, because it was isolated from the rest of the network. After the depicted router had been turned off, routers A and C had the smaller number of changes in the lookup memory.

Fig. 7 shows the number of memory accesses when the POLP algorithm is applied. As in the case of BPFL, router B had the largest number of memory accesses. It can be observed that the POLP update algorithm requires more memory accesses than the BPFL algorithm. This behavior can be explained by the fact that for small routing tables new routes create the larger number of nodes in the trie structure used by POLP. Namely, balanced trees used in the BPFL algorithm have the low number of nodes, and their processing requires

TABLE I
NUMBER OF MEMORY ACCESSES

Number of prefixes		32505	41328	45184	118190	121613	127157	339585	341138	344858
Mean value	BPFL	2.37	3.26	2.17	2.59	1.50	1.57	3.60	2.81	4.58
	POLP	2.43	2.09	1.72	2.09	1.27	1.37	1.37	1.68	1.68
Maximum	BPFL	262	451	266	452	114	192	390	513	579
	POLP	32	87	11	67	22	46	18	61	61
Std. deviation	BPFL	14.93	25.37	13.98	19.41	6.59	8.89	29.25	27.42	37.76
	POLP	2.76	5.38	1.31	4.76	1.26	2.88	1.95	4.75	4.42

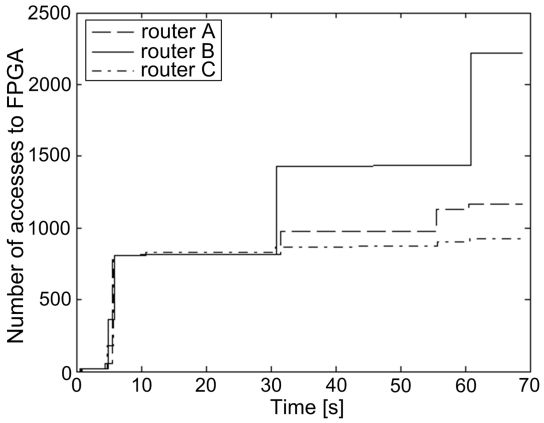

Fig. 7. Number of accesses to lookup memory for POLP update algorithm

the smaller number of memory accesses.

For a number of real-world routing tables [10] we measured the number of memory accesses for both the POLP and the BPFL algorithm when the randomly chosen routes were deleted. The mean value, the maximum and the standard deviation of the number of FPGA accesses are shown in Table I. We can observe that the mean values are moderate and similar for both algorithms. The maximum number of accesses is larger for the BPFL algorithm because it requires the large number of memory accesses when the last route from a subtree is deleted. In that case, the corresponding node of the balanced tree needs to be deleted, and the balanced tree needs to be rearranged so that its completeness is maintained. In addition, a node might have to be moved from the neighbouring tree if it has at least two nodes more than the tree whose node has been deleted.

In order to precisely assess the disruption of packet forwarding because of the lookup table updates, work of the lookup engine should be analyzed. If the lookup engine has modules that are executing in parallel, and if these modules have separate memories, updating the lookup table stored in one of these memories does not directly affect the lookup into the other memory. For example, access to the memory of one level in the BPFL algorithm does not disturb the lookup process at the other level. However, the IP lookup cannot complete until the result from the affected level is obtained. Similarly, access to the memory of one stage in the POLP lookup engine will affect the lookup procedure only in its pipeline. On the other hand, pipelines are completely independent of each other.

VII. CONCLUSION

In this paper, the BPFL and POLP update algorithms were presented. The POLP update algorithm is faster for large routing tables. However, the update times have been shown to be acceptable for both algorithms assuming the large routing tables. We also measured the number of memory accesses required by the BPFL and POLP update algorithms in the smaller networks, and observed that the BPFL update algorithm performs better for the smaller routing tables. Finally, we measured the memory requirements of these algorithms, and confirmed that the BPFL algorithm had the smaller memory requirements, and that its memory savings increase with the routing table size.

ACKNOWLEDGMENT

This work is funded by the Serbian Ministry of Education and Science and companies Informatika and Telekom Srbije.

REFERENCES

[1] Z. Čiča, Luka Milinković and A. Smiljanić, "FPGA Implementation of Lookup Algorithms", *12th International Workshop on High Performance Switching and Routing 2011*, Cartagena, Spain, July 2011.
[2] Z. Čiča and A. Smiljanić, "Balanced Parallelised Frugal IPv6 Lookup Algorithm", *IET Electronics Letters*, Vol. 47 , No. 17, pp.963-965, August 2011.
[3] 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.
[4] 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.
[5] D. Pao, C. Liu, A. Wu, L. Yeung, K.S. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup", *IEEE Proceedings on Computers and Digital Techniques*, Vol. 50 , No. 1, pp.43-52, January 2003.
[6] 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
[7] *OSPFD Routing Software Resources* [Online]. Available: www.ospf.org
[8] W. Jiang and V. K. Prasanna, "Towards Practical Architectures for SRAM-based Pipelined Lookup Engines", *Proc. of IEEE INFOCOM*, March 2010.
[9] M. A. Ruiz-Sanchez, E. W. Biersack, W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", *IEEE Network*, March/April 2001.
[10] http://data.ris.ripe.net/
[11] http://www.ces.net/network/