MEMORY EFFICIENT IP LOOKUP IN 100 GBPS NETWORKS

Jiří Matoušek

CESNET, z. s. p. o.
Zikova 4,
Praha 6, 160 00,
Czech Republic
imatousek@fit.vutbr.cz

Martin Skačan, Jan Kořenek

IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, 612 66, Czech Republic
xskaca00@stud.fit.vutbr.cz, korenek@fit.vutbr.cz

ABSTRACT

The increasing number of devices connected to the Internet together with video on demand have a direct impact to the speed of network links and performance of core routers. To achieve 100 Gbps throughput, core routers have to implement IP lookup in dedicated hardware and represent a forwarding table using a data structure, which fits into the on-chip memory. Current IP lookup algorithms have high memory demands when representing IPv6 prefix sets or introduce very high pre-processing overhead. Therefore, we performed analysis of IPv4 and IPv6 prefixes in forwarding tables and propose a novel memory representation of IP prefix sets, which has very low memory demands. The proposed representation has better memory utilization in comparison to the highly optimized Shape Shifting Trie (SST) algorithm and it is also suitable for IP lookup in 100 Gbps networks, which is shown on a new pipelined hardware architecture with 170 Gbps throughput.

1. INTRODUCTION

The increasing speed of network links has a direct impact to the design and performance of core routers. To achieve 100 Gbps throughput, core routers need dedicated hardware for IP lookup in a forwarding table. Moreover, the size of forwarding tables is increasing with the amount of devices and networks connected to the Internet [1]. It means that core routers have to perform faster IP look up in larger forwarding tables.

The most demanding part of IP packet forwarding is the Longest Prefix Match (LPM) operation. It implements look-up of the longest prefix from a forwarding table, which corresponds to the destination IP address of a packet. For example, let us consider the prefix set from Fig. 1 and a packet with the 8-bit destination address $IP = 11100010$. In this case, prefixes $P1$, $P4$, and $P7$ correspond to the destination address. However, since the prefix $P7$ is the longest one among these prefixes, it is the only result of the LPM operation.

Core routers supporting 100 Gbps throughput have to be able to perform more than 150 million lookups per second (MLPS). Therefore, a new LPM result has to be provided every 6.72 ns. It is possible to achieve such lookup performance only with hardware implementation of the LPM operation [2]. However, in such a case there is usually a bottleneck in relatively slow access to the external memory, where a prefix set extracted from a forwarding table is stored. This can be solved by storing the prefix set in the easily accessible on-chip memory. Nevertheless, the on-chip memory has a limited capacity, therefore the prefix set has to be represented using a memory efficient data structure.

In this paper we propose a novel memory efficient representation of prefix sets, which can be stored in the on-chip memory with a limited capacity. The proposed representation was designed according to analysis of different prefix sets (real IPv4 and IPv6, generated IPv6). We also propose a pipelined hardware architecture, which utilize the designed prefix set representation and is able to perform more than 150 MLPS.

The rest of the paper is organized as follows. Section 2 contains a brief summary of related LPM algorithms. Section 3 describes performed analysis and its results. The proposed novel prefix set representation is introduced in section 4 and the hardware architecture for its processing is described in section 5. Next section 6 shows results of the performed experiments. Conclusion of the paper and remarks about our future work are in section 7.

2. RELATED WORK

The LPM operation is in many commercial devices implemented using (TCAM) *Ternary Content-Addressable Memory*. Such implementation is able to provide an LPM result in just one clock cycle, but TCAMs are expensive, power-

This work was supported by the grant TAČR TA03010561, the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070, the research program MSM 0021630528, and the grant BUT FIT-S-11-1.

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

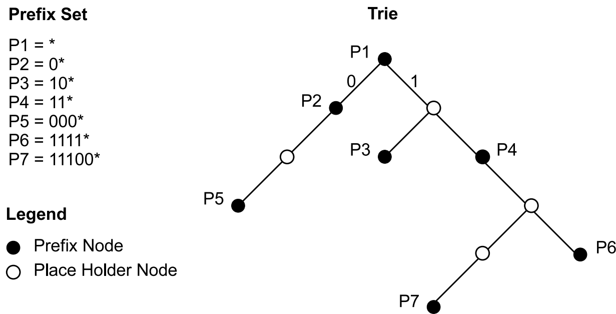

Fig. 1. Sample Prefix Set Represented by Trie

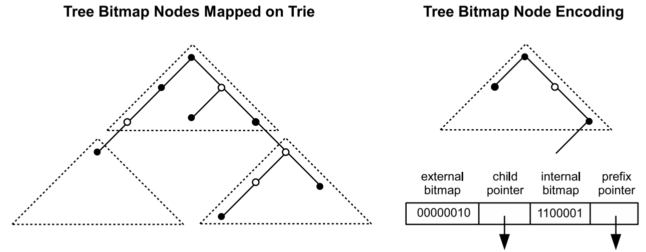

Fig. 2. Tree Bitmap Mapping and Encoding ($SL = 3$)

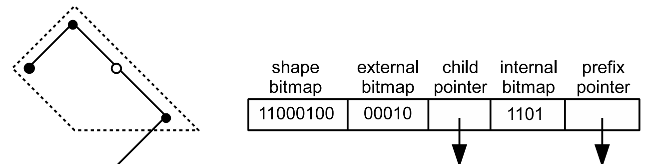

Fig. 3. Shape Shifting Trie Node Encoding ($K = 4$)

hungry and slow in updating their content. Therefore, many algorithmic solutions to LPM have been proposed [3], [4], [5], [6].

A basic data structure utilized in the majority of LPM algorithms is called *trie* [3]. It is a binary tree with prefixes encoded into its structure. The root node of a trie represents the empty prefix. Left and right child nodes of any trie node represent prefixes created from their parent's prefix by appending 0 and 1, respectively. Trie nodes representing prefixes from a prefix set are called prefix nodes, while other trie nodes are referred to as place holder nodes. The representation of the sample prefix set using the trie is shown in Fig. 1. The LPM operation using a trie data structure is performed by traversing a trie from the root to leaves according to bit values of packet's destination address taken from the most significant bit to the least significant bit. The last prefix node visited during such a traversal represents the longest matching prefix.

Adding and removing prefixes from a trie can be done using standard operations on a binary tree. Performing the LPM operation on a trie is also straightforward. However, only one bit of the input can be processed in each step, which means the worst case performance of 32 and 64 steps for IPv4 and IPv6 prefixes, respectively. A trie data structure also has high memory demands, which are caused mainly by the high number of pointers in a trie.

In order to increase lookup performance of trie-based LPM algorithms, multibit tries have been designed. One of the best known multibit trie algorithm is called the *Tree Bitmap* (TBM) [4]. This algorithm represents a set of prefixes using a 2^{SL}-tree, where the parameter SL (i. e. stride length) determines the number of input bits processed in each step of TBM. Mapping of TBM nodes with $SL = 3$ on the trie from Fig. 1 is shown on the left hand side of Fig. 2. On the right hand side of the same figure, there is a sample TBM node and its encoding using two bitmaps and two pointers. The external bitmap contains 2^{SL} bits and it determines the presence of child nodes, while the internal bitmap with $2^{SL} - 1$ bits contains information about prefixes represented by the TBM node. Child and prefix pointers refer to information about child nodes and prefix-related data, respectively.

Use of bitmaps for encoding of a TBM node makes this algorithm easy to implement in hardware. Moreover, the compact representation of a TBM node allows it to be read from a memory in just one clock cycle. The fixed structure of a node is advantageous when updates of a prefix set are performed, however, it may cause high memory overhead in a sparse prefix tree.

Another multibit trie algorithm is called the *Shape Shifting Trie* (SST) [5]. This algorithm is based on TBM, but it tries to overcome its main drawback by introducing adaptive shape of a node, which reduces memory overhead in sparse prefix trees. Adaptive shape is allowed by the shape bitmap consisting of $2K$ bits (see Fig. 3). The parameter K determines the maximum number of underlying trie nodes, that can be represented by a single SST node. SST has exceptionally low memory demands, but its computational complexity is usually unacceptable. Moreover, to the best of our knowledge, there is no hardware architecture implementing SST.

A representation of prefix set with low memory demands and a hardware architecture for its processing, which provides lookup performance higher than 150 MLSP, has been introduced in [6]. We will further refer to this algorithm as the *Prefix Partitioning Lookup Algorithm* (PPLA). PPLA uses a trie data structure, but a trie is utilized only for partitioning a set of prefixes into several disjoint subsets. Each subset is then represented using a separate binary search tree or a 2-3 tree data structure and processed in a separate processing pipeline. This algorithm has good memory efficiency – it needs approximately only one byte of memory to store one byte of IPv4 or IPv6 prefix. However, the proposed representation causes linear growth of mem-

Table 1. Details of Used Prefix Sets

Prefix Set	Prefixes	Source	Date
IPv4			
rrc00	332 118	http://data.ris.ripe.net/	2010-06-03
IPv4-space	220 779	http://bgp.potaroo.net/	2011-12-21
route-views	442 748	http://archive.routeviews.org/	2012-09-20
IPv6			
AS1221	10 518	http://bgp.potaroo.net/	2012-09-21
AS6447	10 814	http://bgp.potaroo.net/	2012-09-21
Generated IPv6			
rrc00_ipv6	319 998	generated using [7] from rrc00	
IPv4-space_ipv6	150 157	generated using [7] from IPv4-space	
route-views_ipv6	439 880	generated using [7] from route-views	

Table 2. Memory Demands of Different LPM Algorithms

Prefix Set	Prefixes	Memory Demands [Kb]		
IPv4		**Trie**	**TBM (*SL=5*)**	**SST (*K=32*)**
rrc00	332 118	47 639.7	9 689.4	6 930.4
IPv4-space	220 779	24 252.4	5 702.1	4 081.0
route-views	442 748	62 650.5	11 942.1	8 775.0
IPv6		**Trie**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	3 518.3	1 076.9	588.5
AS6447	10 814	3 673.8	1 125.1	617.1
Generated IPv6		**Trie**	**TBM (*SL=4*)**	**SST**
rrc00_ipv6	319 998	307 641.5	87 257.1	N/A
IPv4-space_ipv6	150 157	153 877.3	43 958.7	N/A
route-views_ipv6	439 880	418 663.7	118 889.4	N/A

ory demands with the number of represented prefixes and initial partitioning of a prefix set introduces very high preprocessing overhead.

Linear dependence of memory demands on the number of represented prefixes is one of the most significant issues connected with PPLA. In trie-based LPM algorithms, nodes close to the root of a tree are shared by several prefixes. This property should allow to represent one byte of prefix using less than one byte of memory. Therefore, we focus our analysis on previously introduced trie-based algorithms.

3. ANALYSIS

Basic information about prefix sets extracted from forwarding tables of core routers, which we use in our analysis, are summarized in Table 1. In order to obtain results relevant for many different situations, we use sets of real IPv4 and IPv6 prefixes as well as sets of IPv6 prefixes generated using [7]. Moreover, diversity of data for analysis is increased by using real IPv4 and IPv6 sets from different sources and acquired on different days. Experiments with prefix sets were performed using Netbench tool [8].

The first part of analysis was focused on memory demands of Trie, TBM, and SST algorithms and its results are shown in Table 2. Parameters SL and K of TBM and SST algorithms, respectively, were chosen with respect to the minimum memory demands. As can be seen, K was set to the same value for all prefix sets, while SL was set to a different value for each group of prefix sets. This reflects different density of the prefix tree (smaller value of SL means lower density) between groups of prefix set. Missing results of SST memory demands for generated IPv6 prefix sets cannot be provided because of very high computational complexity of SST.

Table 2 shows, that the lowest memory demands can be achieved when SST is used, while the highest memory demands are connected with the Trie algorithm. Such results would propose SST to be a candidate for further optimization of memory demands. However, as stated in section 2, SST suffers from high computational complexity and there

Table 3. Classification of Nodes From a TBM Representation of route-views (434 552 Nodes, $SL = 3$)

Prefixes	Child Nodes 0	1	2	3	4	5	6	7	8
0	0	**26 829**	**11 859**	**6 876**	**5 422**	**3 679**	**3 547**	**4 297**	**14 138**
1	**278 804**	6 220	4 244	2 840	4 463	1 683	2 416	876	2 051
2	**21 005**	3 270	4 198	1 599	2 688	724	792	393	842
3	**5 716**	1 093	2 000	596	806	293	286	160	306
4	**3 786**	447	543	220	322	106	129	102	267
5	**679**	63	55	22	48	20	25	25	78
6	**298**	30	22	9	23	3	9	6	64
7	**70**	6	3	3	8	4	3	7	46

Table 4. Classification of Nodes From a TBM Representation of AS1221 (25 063 Nodes, $SL = 3$)

Prefixes	Child Nodes 0	1	2	3	4	5	6	7	8
0	0	**11 303**	**1 666**	**812**	**538**	**184**	**145**	**131**	**249**
1	**8 965**	547	142	19	17	3	2	1	1
2	**193**	21	14	4	3	0	1	0	0
3	**50**	3	3	3	1	0	1	0	0
4	**29**	3	1	1	3	1	1	0	0
5	**0**	1	0	1	0	0	0	0	0

is no hardware architecture for this algorithm. Therefore, the next part of our analysis was focused on identification of possibilities for optimization of TBM's memory demands.

TBM analysis was performed by classification of TBM nodes according to the number of child nodes and the number of prefixes represented by a TBM node. Results of this classification for selected IPv4, IPv6 and generated IPv6 prefix sets are shown in Tables 3, 4, and 5, respectively. Even though all tables show classification of TBM nodes with $SL = 3$, presented results can be used for identification of general trends in TBM.

Analysis of the TBM representation of all selected prefix sets shows two significant groups of node. The first group contains leaf nodes (the leftmost column in Tables 3, 4, and 5), while the second group contains internal nodes without prefixes (the first row in Tables 3, 4, and 5). Therefore, efficient encoding of nodes from these two groups will significantly reduce memory demands of TBM.

Table 5. Classification of Nodes From a TBM Representation of route-views_ipv6 (2 239 971 Nodes, $SL = 3$)

Prefixes	Child Nodes								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**1 597 683**	**143 258**	**39 958**	**21 056**	**9 332**	**5 637**	**3 958**	**4 462**
1	**406 100**	3 503	746	263	108	45	15	10	6
2	**2 623**	171	118	40	39	21	8	6	1
3	**475**	37	24	5	7	3	1	3	1
4	**155**	11	7	3	1	1	0	0	0
5	**44**	1	4	3	2	1	0	0	0
6	**12**	0	1	0	0	0	0	0	0
7	**2**	0	0	0	0	0	0	0	0

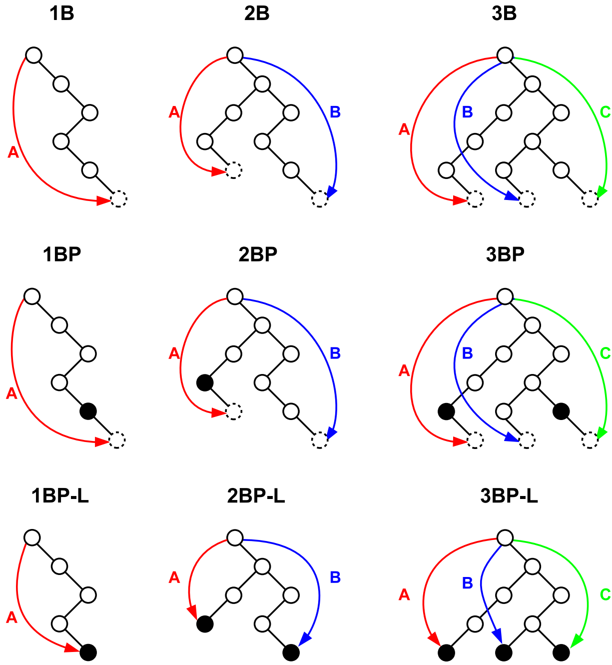

Fig. 4. Newly Proposed Types of Node

4. PREFIX SET REPRESENTATION

Performed analysis has shown two groups of TBM nodes, whose more efficient encoding could reduce memory demands of TBM. To this end, we propose a representation of prefix set using thirteen different types of node. These thirteen types can be divided into two groups – nine newly proposed nodes (see Fig. 4) and four variants of a TBM node. Properties of nodes in Fig. 4 are reflected in their name. A node can encode 1 branch (1B), 2 branches (2B) or 3 branches (3B) of an underlying trie. It can also contain a prefix node (P), but such a prefix node is allowed only in the lowest level of the node and it may not occur in all branches. Presence of a prefix node in the lowest level of all branches is compulsory only in the case of leaf nodes (L). Used TBM nodes include a standard node for $SL = 3$ (TBM3) and leaf TBM nodes for $SL = 3, 4, 5$ (TBM3-L, TBM4-L, TBM5-L).

Table 6. Basic Parameters of Different Types of Node When Aligned to 8-bit and 16-bit Boundary

Node Type	Size Aligned to 8 bits			Size Aligned to 16 bits		
	Branch Length [bits]	**Unaligned Size** [bits]	**Aligned Size** [bits]	**Branch Length** [bits]	**Unaligned Size** [bits]	**Aligned Size** [bits]
1B	24	56	56	17	48	48
1BP	19	72	72	13	64	64
1BP-L	20	48	48	20	48	48
2B	16	72	72	14	64	64
2BP	10	80	80	11	80	80
2BP-L	12	55	56	15	61	64
3B	11	78	80	12	78	80
3BP	5	80	80	6	80	80
3BP-L	7	53	56	9	62	64
TBM3	3	75	80	3	67	80
TBM3-L	3	30	32	3	30	48
TBM4-L	4	38	40	4	38	48
TBM5-L	5	54	56	5	54	64

In order to make hardware implementation of the proposed representation feasible, it is necessary to align the size of node representations to some boundary. A smaller boundary implies smaller memory overhead but higher number of different sizes of node, hence higher utilization of resources for processing such data structures. Therefore, we consider two different alignments (to the 8-bit and 16-bit boundary), which should allow us to achieve a reasonable compromise between memory overhead and resources utilization. We will examine real memory demands and resources utilization for both alignments.

Basic parameters of different types of node, when aligned to the 8-bit and 16-bit boundary, are summarized in Table 6. The size of a node is determined mainly by the maximum branch length and the presence of child and prefix pointers. The maximum branch length, which is the same for all branches in a node, is shown in Table 6. The prefix pointer encoded on 19 bits is present only in nodes, that can represent prefixes, i. e. nodes with P in their name and TBM nodes. The child pointer is encoded on 23 bits (in the case of alignment to the 8-bit boundary) or 22 bits (in the case of the 16-bit boundary) and it is present in all non-leaf nodes, i. e. nodes without L in their name. Since Table 6 shows aligned as well as unaligned size of each type of node, memory overhead introduced by alignment can be computed as difference of these two values.

The mapping of proposed nodes on a trie is done according to the algorithm in Fig. 5. This algorithm uses, except standard queue operations ENQUEUE and DEQUEUE, three auxiliary functions. MAP_COST returns the cost of mapping of given type of node from the specified position in the trie. The cost is determined using equation (1), where p is the number of covered prefix nodes, n is the number of all covered trie nodes, and $size$ is the size of given type of node. Mapping of the selected type of node to the specified posi-

Input: pointer $root$ pointing to the root node of the trie
Output: pointer $root$ pointing to the root node of the mapped tree

1: $Q \leftarrow \varnothing$
2: **if** $root \neq NULL$ **then**
3: ENQUEUE($Q, root$)
4: **while** $Q \neq \varnothing$ **do**
5: $trie \leftarrow$ DEQUEUE(Q)
6: $max_cost \leftarrow 0$
7: $best_type \leftarrow NULL$
8: **for each** $type \in node_types$ **do**
9: $cost \leftarrow$ MAP_COST($type, trie$)
10: **if** $cost > max_cost$ **then**
11: $max_cost \leftarrow cost$
12: $best_type \leftarrow type$
13: $trie \leftarrow$ MAP($best_type, trie$)
14: **for each** $child \in$ CHILDREN($trie$) **do**
15: ENQUEUE($Q, child$)

Fig. 5. Pseudocode of the Mapping Algorithm

tion in the trie is done using MAP function and CHILDREN returns a list of child nodes of the given node.

$$cost = \begin{cases} \frac{p}{size} & \text{if } \frac{p}{size} > 0 \\ \frac{n}{size} & \text{otherwise} \end{cases} \quad (1)$$

5. HARDWARE ARCHITECTURE

Since the proposed representation of prefix set can be classified as multibit trie approach to LPM, a matching result is in the worst case available after processing of n nodes, where n is the height of the tree, which represents the prefix set. In order to achieve lookup performance of 150 MLSP, it is necessary to employ a processing pipeline, where each processing element (PE) performs one step of the LPM algorithm.

We propose the hardware architecture in Fig. 6 for processing of our representation of prefix set. This architecture consists of two processing pipelines with uniform PEs and dual port memory blocks shared between PEs from corresponding stages. By utilization of the dual port memory, we can achieve double performance of a single pipeline architecture without compromising on memory access. The memory block for each pipeline stage contains two parallel memories, each of which has data width of 80 bits (the maximum size of a node, see Table 6). Use of two parallel memories allows to read the whole node in one clock cycle, even if it is stored in two consecutive data words.

A high-level architecture of one PE is also shown in

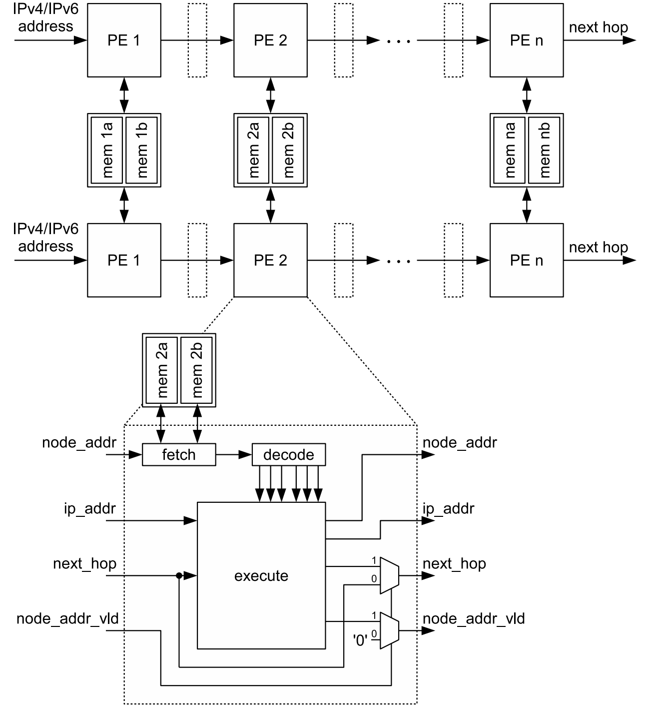

Fig. 6. Double Processing Pipelines With Detail of One Processing Element (PE)

Fig. 6. Processing of a node in PE is like a processing of an instruction in a standard CPU. First of all, PE *fetches* the node from the memory. The representation of the node is then *decoded* and sent in parallel to the *execute* submodule. The internal structure of the execute part is shown in Fig. 7. The main part of execution is done in branch A proc, branch B proc, branch C proc, and TBM node proc submodules. The first three of them are dedicated for processing of corresponding branches of newly proposed types of node (branches are marked by letters A, B, and C in Fig. 4), while the TBM node proc submodule is dedicated for processing of TBM nodes. Since processing of different branches and different types of node is done in parallel, the select branch and the select result modules are used to select correct values for outputs of PE.

Combinatorial logic of fetch and execute submodules of PE is relatively complex. Therefore, in order to achieve desired lookup performance, it is necessary to use intra-stage registers within these two submodules. Each of them contains two sets of internal registers. In total, each PE contains four sets of intra-stage registers, thus processing a node within one PE is done in five clock cycles.

Section 4 describes two variants of node alignment in a memory – to the 8-bit or 16-bit boundary. Both variants can be processed using conceptually the same hardware architecture with only some minor changes in fetch, decode, and execute submodules. Different node alignment has the

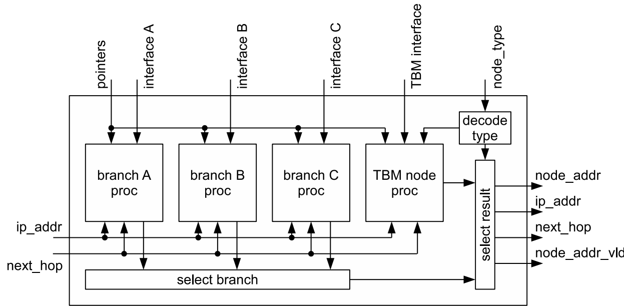

Fig. 7. Internal Structure of PE's Execute Part

biggest influence on data reorder logic in the fetch module. It also has to be reflected in the decode submodule by different interconnection of decoding logic. Relatively the smallest changes have to be done in the execute module, where it is sufficient to change data width of some internal buses.

6. EXPERIMENTAL RESULTS

First of all, we have measured memory demands of the proposed representation of prefix set on our sample IPv4 and IPv6 (both real and generated) prefix sets. The measurement has been done for both variants of node alignment in a memory and its results are presented in Table 7. Results show lower memory demands in the case of 8-bit node alignment. The difference between memory demands of the representation with nodes aligned to 8 bits and 16 bits is the most evident for IPv4 prefix sets. This is because of different density of prefix trees in their leaf part. The prefix tree of IPv4 sets is denser than the tree in the case of IPv6 sets. Therefore, leafs of the IPv4 tree are represented mainly by TBM nodes (which introduce the highest memory overhead when aligned to the 16-bit boundary), while leafs of the IPv6 tree are represented mainly by newly proposed nodes (which introduce almost the same memory overhead for both 8-bit and 16-bit alignment). Table 7 also shows the height of the tree, which represents particular prefix sets.

Both variants of the proposed architecture have been implemented for a Xilinx Virtex-6 XC6VSX475T FPGA. Utilization of resources and maximum frequency after place & route using Xilinx ISE 14.3 are shown in Table 8. As can be seen, the main difference between two proposed architectures is in the number of utilized LUTs, where the 16-bit architecture shows better results. This is mainly due to smaller data width of some buses in this architecture. The number of utilized registers is practically the same and maximum frequency is little higher in the case of the 8-bit architecture. Table 8 contains information about utilization of resources by one PE, the complete processing pipeline and also by the whole proposed architecture, which consists of two processing pipelines. Even though the length of each pipeline

Table 7. Memory Demands and Tree Height of the Proposed Representation on Different Prefix Sets

Prefix Set	Prefixes	Memory Demands [Kb] 8-bit Alignment	16-bit Alignment	Tree Height
IPv4				
rrc00	332 118	6 330.8	7 287.6	12
IPv4-space	220 779	3 571.4	4 297.4	12
route-views	442 748	7 779.8	9 039.6	12
IPv6				
AS1221	10 518	475.8	489.0	18
AS6447	10 814	493.8	506.6	23
Generated IPv6				
rrc00_ipv6	319 998	21 264.3	21 373.2	21
IPv4-space_ipv6	150 157	10 412.2	10 421.4	18
route-views_ipv6	439 880	29 039.5	29 207.4	20

Table 8. Resources Utilization and Maximum Frequency of Proposed Hardware Architecture (Xilinx ISE 14.3, Virtex-6 XC6VSX475T)

8-bit Alignment	LUTs (% of All)	Registers (% of All)	Frequency [MHz]
1 PE	3 647 (1.23 %)	1 825 (0.31 %)	127.162
1 pipeline (23 PEs)	83 881 (28.19 %)	41 957 (7.05 %)	127.162
2 pipelines (46 PEs)	167 762 (56.37 %)	83 950 (14.11 %)	127.162
16-bit Alignment	**LUTs (% of All)**	**Registers (% of All)**	**Frequency [MHz]**
1 PE	3 194 (1.07 %)	1 817 (0.31 %)	123.183
1 pipeline (23 PEs)	73 462 (24.69 %)	41 791 (7.02 %)	123.183
2 pipelines (46 PEs)	146 924 (49.37 %)	83 582 (14.04 %)	123.183

(23 PEs) allows processing the prefix set represented by the highest tree (real IPv6 set AS6447, see Table 7), the whole architecture fits into the target FPGA.

Since resources utilized by both variants of the hardware architecture are significantly lower than resources available in the target FPGA, selection of "better" variant is governed mainly by their memory demands, whose optimization is the main objective of this work. Therefore, we select the representation of prefix set with nodes aligned to the 8-bit boundary.

The selected variant can also operate on a little higher frequency, which implies higher lookup performance. Both processing pipelines are able to provide one matching result in each clock cycle, which translates into total lookup performance of almost 255 MLPS. Thus, the proposed solution is able to support throughput of 170 Gbps. Frequency of the proposed solution also determines, together with the number of pipeline stages, the overall latency. As stated in section 5, each PE consists of five pipeline stages. Therefore, the whole pipeline contains $5 \times 23 = 115$ stages. Since pro-

Table 9. Memory Demands of the Proposed Representation of Prefix Set and its Comparison to TBM and SST

Prefix Set	Prefixes	Memory [Kb]	Savings	
IPv4		**New Nodes**	**TBM (*SL=5*)**	**SST (*K=32*)**
rrc00	332 118	6 330.8	34.67 %	8.65 %
IPv4-space	220 779	3 571.4	37.37 %	12.49 %
route-views	442 748	7 779.8	34.85 %	11.34 %
IPv6		**New Nodes**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	475.8	55.82 %	19.16 %
AS6447	10 814	493.8	56.11 %	19.98 %
Generated IPv6		**New Nodes**	**TBM (*SL=4*)**	**SST**
rrc00_ipv6	319 998	21 264.3	75.63 %	N/A
IPv4-space_ipv6	150 157	10 412.2	76.31 %	N/A
route-views_ipv6	439 880	29 039.5	75.57 %	N/A

Table 10. Memory Efficiency Ratio (Bytes of Memory/Bytes of Prefixes) of the Proposed Representation of Prefix Set, TBM and SST

Prefix Set	Prefixes			
IPv4		**New Nodes**	**TBM (*SL=5*)**	**SST**
rrc00	332 118	0.610	0.934	0.668
IPv4-space	220 779	0.518	0.826	0.592
route-views	442 748	0.562	0.863	0.634
IPv6		**New Nodes**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	0.724	1.638	0.895
AS6447	10 814	0.731	1.665	0.913
Generated IPv6		**New Nodes**	**TBM (*SL=4*)**	**SST (*K=32*)**
rrc00_ipv6	319 998	1.063	4.363	N/A
IPv4-space_ipv6	150 157	1.109	4.684	N/A
route-views_ipv6	439 880	1.056	4.324	N/A

cessing in one stage takes 7.86 ns, the overall latency of the proposed solution is 903.90 ns. The overall latency also determines the size of the buffer for packets waiting for the LPM result, which has to be at least 8.6 KB for 100 Gbps Ethernet link.

Comparison of our prefix set representation, TBM, and SST in terms of memory demands is provided in table 9. Except memory demands of our solution, we show its savings compared to other LPM algorithms. The proposed prefix set representation overcomes both TBM and SST, but reduction of memory demands is higher for TBM (between 34.67 % and 76.31 %) than for SST (between 8.65 % and 19.98 %). Moreover, it is shown that the sparse prefix tree of IPv6 prefix set allows higher savings, which is due to higher utilization of memory efficient newly proposed types of node (see Fig. 4).

In order to compare memory efficiency of the proposed prefix set representation with PPLA, we provide a memory efficiency ratio (bytes of memory required to store one byte of prefix) of our solution on different prefix sets in Table 10. The value of this parameter is shown also for TBM and SST. According to [6], the average memory efficiency ratio of PPLA on generated IPv6 prefix sets is 1.01 when a 2-3 tree data structure is used. Therefore, our solution is comparable to PPLA on generated IPv6 prefix sets. However, our prefix set representation is significantly better than PPLA on IPv4 prefix sets, where [6] reports the average memory efficiency ratio of 1.00. Moreover, both TBM and SST, which were not taken into account in [6], shows better memory efficiency than PPLA on IPv4 sets. Memory efficiency of our solution and PPLA on real IPv6 prefix sets cannot be compared, because this value is not reported in [6].

Since both our solution and TBM are based on the trie, they should achieve better (i.e. lower) memory efficiency ratio on prefix sets with high number of prefixes (generated IPv6), than on prefix sets with a small number of prefixes (real IPv6). However, according to the results presented in Table 10, this is not true in our case. The most probable explanation of this situation is that IPv6 prefix sets generator [7] does not model the process of assigning IPv6 addresses correctly. We have used this generator in order to be able to compare our results with results presented in [6].

7. CONCLUSION AND FUTURE WORK

The paper proposed a novel representation of IP prefix sets using thirteen different types of node designed for a memory efficient representation of the most common situations in a prefix tree. This prefix set representation has significantly lower memory demands than TBM and it also overcomes the SST algorithm. Moreover, the proposed representation shows better memory efficiency than PPLA on real IPv4 prefix sets and comparable results on generated IPv6 prefix sets. Memory efficiency of the proposed representation and PPLA on real IPv6 prefix sets cannot be compared.

We also introduced a pipelined hardware architecture, which utilizes the proposed prefix set representation. The architecture was implemented on Xilinx Virtex-6 FPGA with 170 Gbps throughput.

As future work, we want to optimize resources utilization and lookup performance of the proposed architecture. We would also like to utilize dynamic partial reconfiguration for allocation of memory blocks to particular pipeline stages according to the actual prefix set.

8. REFERENCES

[1] (2013, Jan.) IPv6 / IPv4 Comparative Statistics. [Online]. Available: http://bgp.potaroo.net/v6/v6rpt.html

[2] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms,” *IEEE Network*, vol. 15, no. 2, pp. 8–23, Mar. 2001, ISSN 0890-8044.

[3] E. Fredkin, “Trie Memory,” *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, Sept. 1960, ISSN 0001-0782.

[4] W. Eatherton, G. Varghese, and Z. Dittia, “Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates,” *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, Apr. 2004, ISSN 0146-4833.

[5] H. Song, J. Turner, and J. Lockwood, “Shape Shifting Tries for Faster IP Route Lookup,” in *Proc. of the 13th IEEE International Conference on Network Protocols (ICNP’05)*. IEEE Computer Society, 2005, pp. 358–367, ISBN 0-7695-2437-0.

[6] H. Le and V. K. Prasanna, “Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning,” *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1026–1039, July 2012, ISSN 0018-9340.

[7] M. Wang, S. Deering, T. Hain, and L. Dunn, “Non-random Generator for IPv6 Tables,” in *Proc. of the 12th Annual IEEE Symposium on High Performance Interconnects, 2004*. IEEE Computer Society, Aug. 2004, pp. 35–40, ISBN 0-7803-8686-8.

[8] V. Pus, J. Tobola, V. Kosar, J. Kastil, and J. Korenek, “Netbench: Framework for Evaluation of Packet Processing Algorithms,” in *Seventh ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS’11)*. IEEE Computer Society, Oct. 2011, pp. 95–96, ISBN 978-0-7695-4521-9.