Scalable Lookup Algorithms for IPv6

Aleksandra Smiljanić[a*], Zoran Čiča[a]

[a] School of Electrical Engineering, Belgrade University, Bul. Kralja Aleksandra 73, 11120 Belgrade, Serbia

ABSTRACT

IPv4 addresses have been exhausted, and the Internet is moving to IPv6. The large number of IP lookup algorithms has been proposed in the past. In this paper we analyze their scalability, and applicability to IPv6. In particular, we calculate the memory requirements of diverse lookup algorithms, and compare them for different lookup table sizes, assuming the high lookup throughput of one lookup per memory access. BPFL (Balanced Parallelized Frugal Lookup) algorithm requires the smallest external and internal memories for the examined IPv6 lookup tables. In BPFL, the lookup table is defined using the range selector, balanced trees, and bitmaps. Lookup algorithms based on hashing and TCAMs also use memory frugally.

Keywords: address lookup algorithms; IP packet forwarding; Internet router; lookup tables.

1. Introduction

Internet is growing in terms of the number of devices connected, and their speeds. The IP lookup is one of the most challenging tasks of scalable Internet routers. At very high speeds, routers should determine the output port (i.e. the next-hop information) for each incoming packet based on its global IP address.

The number of networked mobile devices is rapidly growing. Also, it is foreseen that the increasing number of appliances will be networked for the remote control. At the same time, IPv4 addresses have been exhausted. Therefore, migration of IPv4 to IPv6 is both the reality, and the necessity. This has been confirmed by the IPv6 Day, 18. June 2011, when major Internet service providers and web sites demonstrated a global scale IPv6 trial. Still, IPv6 lookup tables are small counting a couple of thousands entries, while IPv4 lookup tables count more than 300K entries. It is important to explore if the existing IP lookup algorithms are capable to support the large IPv6 lookup tables which are inevitable in the near future, and, if needed, to propose new scalable solutions.

In this paper, we will give an overview of the principles and techniques that were used for designing a large number of IP lookup algorithms [1-31]. We will focus on their scalability and applicability to IPv6. The lookup algorithms should support large lookup tables with IPv6 addresses which are 4 times longer than IPv4 addresses, and, at the same time, they are supposed to achieve high speeds. Namely, the port bit rates are high with the tendency of growing, while the packets are short. The large portion of the packets on the Internet is as short as 64B, and their duration is 50ns at 10Gb/s, or 12ns at 40Gb/s. So, within these rather short times, it has to be decided where the packet should be routed based on 128bits of its IPv6 address. It is a formidable task, hard to achieve. Fortunately, the latency of the IP lookup is not as important as the lookup throughput, i.e. the number of lookups that can be performed per second. A straightforward way to increase the lookup throughput is by parallelization. Parallelization by rule implies multiple memories. If these memories ought to be on the external chips, the printed circuit boards (PCB) become complex. Desirably, multiple memories should be stored on chip together with the logic, in order to keep the PCB design simple.

[*] Corresponding author. Tel. +381 64 370 9539
E-mail addresses: aleksandra@etf.rs (A.Smiljanić), zoran.cica@etf.rs (Z. Čiča)

Memory required for the IP lookup tables has been a critical hardware resource. The lookup algorithms with the smaller memory requirements are, consequently, more scalable. For this reason, our focus will be on the memory requirements of different lookup algorithms. We will analyze their on-chip memory requirements as well, because the required on-chip memory size determines the level of parallelization that can be achieved by some lookup algorithm. We will examine and compare a large number of lookup algorithms which combine a variety of techniques. The IP lookup algorithms can be divided into three main categories: algorithms based on m-ary trees [1-20], algorithms based on hashing [21-24] and TCAM algorithms [25-31]. Representatives of these three categories will be examined. In addition, our proposed BPFL (Balanced Parallelized Frugal Lookup) algorithm [1-2] will be compared with the previous solutions.

This paper is organized as follows. In Section 2, we will describe different types of lookup algorithms, as well as the techniques which are the building blocks of these algorithms. Formulas for the memory requirements are derived for eight representative lookup algorithms in Section 3. In Section 4, the lookup algorithms are compared in the case of real IPv4 lookup tables, and extrapolated IPv6 lookup tables, in terms of their speed and memory requirements. The complexity of updating will be also discussed. Finally, in Section 5, implementation of BPFL will be presented since it will be shown to have the lowest memory requirements in the case of IPv6. Section 6 concludes the paper.

2. Promising Lookup Algorithms

2.1 Lookup Algorithms based on m-ary Trees

Lookup algorithms based on unibit or multibit trees were widely explored in the literature [1-20]. Unibit, or binary, trees comprise nodes that carry the next-hop information and/or the pointers to the child nodes [15]. In each step of the tree traversal, the next child is determined by the appropriate bit of the IP address for which the next-hop information is searched. The lookup tables based on binary trees consume less memory than the straightforward lookup tables storing $2^{L_{max}}$ entries for all possible prefixes, where L_{max} is the maximum prefix length i.e. the IP address length. However, their memory requirements are not minimal, and can be optimized in various ways. Also, the simplest lookup algorithm is slow since it requires L_{max} memory accesses in the worst case. Path compression can help reduction of the memory requirements, and the speed increase [13, 15]. It stands for the removal of one-way branch nodes of a tree since no decision is made in those nodes. However, improvements provided by the path compression are limited, and the worst-case performance remains the same. Another popular technique for the memory saving is leaf pushing [16], in which the next-hop information is pushed to the leaves. In this way, the node size is reduced since any node stores either the pointers, or the next-hop information, and not both of them. In order to improve the speed of the IP lookup algorithms, the multibit trees were proposed for storing the lookup tables [5,8-9,12,14,17].

In each step of the multibit tree traversal, the child node is determined based on the appropriate part of the given IP address, i.e. the multibit stride. In this way, the worst-case IP address lookup takes the smaller number of accesses for longer strides. In particular, it takes L_{max}/L memory accesses, where L is the stride length in bits. Prefixes which are not divisible by the stride length, L, are typically expanded, and represented by multiple longer prefixes whose length is divisible by L. For example, for strides with length L=4 bits, prefix 01*, should be expanded to prefixes 0000*, 0001*, 0010*, 0011* and its next-hop information will be pushed to these prefixes unless they store the next-hops that correspond to the longer prefix (3, or 4 bits long in this case). So, multibit trees have larger memory requirements than the binary tree while achieving the higher speed. There are different ways to reduce the memory requirements of multibit trees.

One way to reduce the memory of multibit trees is by the node compression as is done in Lulea algorithm [14]. Since consecutive nodes in multibit trees often carry the same next-hop information, this next-hop information can be stored only in the first of these nodes, while the bitmap vector stores the deleted nodes

[14]. The bitmap vector has its element equal to “1”, if the node corresponding to this element is not deleted. The bitmap technique was also analyzed in [1-3,6-8,11]. In FIPL (Fast Internet Protocol Lookup) algorithm, each multibit node contains: the internal bitmap that indicates the prefixes with next-hops within that node, the external bitmap that indicates which children of the node exist, a pointer to the array of next-hops that correspond to the internal nodes, and a pointer to the array of children nodes [5]. The bitmap technique was the main building block of other lookup algorithms as well, such as EHAF (Efficient Hardware Architecture for Fast IP Lookups) and PSTS (Parallel Search Trie-based Scheme) [6,7]. But, these algorithms are not sufficiently scalable to support IPv6, as will be shown later in this paper.

Path compression can be applied to the multibit trees too. Level compressed (LC) tree is obtained by recursive transformation of a binary tree [17]. Starting from its root, the largest full binary subtree is replaced by the one-level multibit tree. The recursive process is then repeated starting from each leaf of the multibit subtrees obtained in the previous phase of transformation. Alternatively, a binary subtree is replaced by the one-level multibit subtree if it has sufficiently high density (the number of nodes divided by 2^D, where D is the subtree depth). In this way, the memory requirements of the multibit trees can be reduced, but still, multiple memory accesses are required in order to find the longest prefix match.

One of the main problems of trees, especially in the IPv6 case, is the storage of very large number of empty nodes needed only for traversing the tree. These empty nodes significantly increase memory requirements. PT (Priority Tree) algorithm overcomes this problem by moving the longer prefixes into empty nodes of levels closer to the root of a binary tree [18]. There are two types of nodes – ordinary node whose prefix corresponds to the node’s path and priority node which stores the prefix of some of its descendants. In this way, empty nodes are avoided and the number of nodes in a tree equals the number of prefixes. As a result, the larger lookup tables can be supported by priority trees. However, due to asymmetry of original binary trees, the depth of priority trees is not significantly reduced, and the lookup time can be reduced only by using multiple memories. CMST (Classified Multi-Suffix Trie) algorithm uses the same idea of pushing prefixes to the upstream empty nodes, only within multibit and not binary trees [19].

The lookup speed of the multibit trees can be increased by the parallelization and pipelining [1-4,10]. Nodes at different levels must be stored in different memories. Search of the longest prefix match still requires multiple memory accesses. However, after the first level is searched and its memory is accessed for some IP address, it can be then searched for the next IP address without waiting for the first search to complete. In this way, the lookup throughput (the number of lookups per second) can be increased by the factor equal to the number of memories. In POLP algorithm (Parallel Optimized Linear Pipeline), the lookup throughput is further increased by evenly distributing subtrees across the multiple pipelines that work in parallel [4].

Fig.1 shows the POLP architecture in which J ($J \leq Q$) lookups can be performed in one clock cycle. The pipeline selector should be able to switch J lookup requests (i.e. IP addresses) to the pipelines that hold the corresponding subtrees of interest. External memory is associated to each pipeline, and it stores the next hop information for the nodes in this pipeline. Nodes of each pipeline are split among the stage memories, so that these stages can be accessed also in parallel. In this way, the lookup throughput of one pipeline is increased.

2.2 Lookup Algorithms based on Hashing

Lookup algorithms achieve various savings using the fact that a small portion of possible prefixes is actually used. Hashing offers itself as a natural way to exploit this fact [21-24]. Hash is a function that maps a bit sequence into, typically, a shorter bit sequence, so that the probability of collision (that two arguments have the same result) is very low. Lookup algorithms based on hashing are difficult to scale for several reasons. First, hash function assumes the arguments of equal lengths. Secondly, good hash functions that

provide low probability of collision are complex and slow. Finally, the entire prefixes should be stored in hash tables, and they have high memory requirements.

Since the hash functions assume the arguments of equal lengths, only prefixes of equal lengths can be searched within one hash table access. If the prefix of a certain length exists in the hash table, then the search should be continued among the longer prefixes that were not examined, and vice versa. In the worst case, prefixes of $\log_2 L_D$ different lengths should be searched in separate hash tables, where L_D is the number of different prefix lengths. Unfortunately, these search phases cannot be pipelined since the sequence in which the hash tables are accessed depends on the incoming IP addresses.

There are different ways to speed up the search through different hash tables. One is to replicate hash tables onto different memories, and to pipeline searches for the longest match prefixes of different IP addresses using these memories. Obviously, increasing the number of memory chips increases the complexity of the PCB design. Alternatively, the number of the prefix lengths can be reduced, by the prefix expansion [23]. In this case, each prefix is substituted by multiple longer prefixes of the closest longer length that is hashed. As a result, the existing IPv4 lookup tables would be significantly enlarged (at this moment around 5 times), and using several large DRAM memories would be required. This scheme also increases the complexity of the PCB design. In OAL (Offset Addressing Lookup), the lookup comprises two steps [24]. In the first step, the longest matching prefix is found in a binary tree whose nodes are addressed by their offsets with respect to the parent nodes. In the second step, this prefix is hashed, and its hash determines the bucket where the next-hop information resides. The selected bucket is then searched for the next-hop information. In this way, multiple hashing for different prefix lengths during one lookup is avoided. However, in each step multiple memory accesses are needed. Another way to increase the lookup speed is by using Bloom filters [21]. Bloom filters are based on prefix hashes, and, they can determine if a given prefix exists in the table, before this table is accessed. Bloom filters can be wrong with small probabilities, and give false positives.

The best hashing algorithms such as MD, or SHA would spread prefixes evenly over the buckets in the hash table. However, they are complex for implementation, and, therefore slow. Fast hashing algorithms produce collisions with non-negligible probabilities. As a result, multiple prefixes may have the same hash, and would be stored in the same bucket of the hash table. Then, the search of prefixes with the given length might require multiple accesses to the hash table. Different ways were proposed to minimize the effect of collisions on the lookup speed. For instance, it was proposed that buckets should be divided into sections, only one hash is calculated for the given prefix, and the prefix is stored in the section with the smallest number of prefixes in the corresponding bucket [22]. Still, all sections that may contain the given prefix might have to be examined. Counting Bloom filters can be used to determine in advance if some bucket contains the given prefix as was proposed in PHFT (Pruned Fast Hash Table) [21]. Multiple hashes are calculated for each prefix, and the appropriate counters are incremented when the prefix is inserted and vice versa. A prefix is stored in the hash table with the lowest counter. Incremental updating of the PHFT lookup table is a formidable task.

Since multiple next-hops might be stored in some bucket due to the collisions, each next-hop must be associated with its prefix. Consequently, prefixes must be stored in the hash table, which reduces the scalability of the IPv6 lookup algorithms based on hashing.

2.3 TCAM Lookup Algorithms

Ternary Content Addressable Memories (TCAM) allows the most straightforward lookup within one clock-cycle [25-31]. For a given input word, in our case IP prefix, TCAM provides its location as the output. The obtained TCAM location determines the SRAM location from which the router output port could be read. TCAMs store ones, zeros and don't cares which are convenient for storing prefixes of variable lengths [26]. The prefixes should be stored in TCAMs in the decreasing order of their lengths, which makes the

incremental updates of TCAMs complex. TCAMs allow fast lookup, but they are more complex and more power hungry than other types of memories, e.g. SRAMs that are widely used for the IP lookup as well [26]. TCAMs consume two times more transistors than SRAMs, and they consume the power which is on the order(s) of magnitude higher. For these reasons, TCAMs were used mainly in switches and edge routers that do not store the large number of prefixes. However, there are different proposals in the research literature to overcome the limitations of TCAMs and make them more scalable and power-efficient.

TCAM lookup algorithms typically include partitioning of a TCAM into blocks, and activating of only few blocks at the time when the TCAM is searched [25,27,29-31]. Namely, the consumed power is proportional to the number of bits that are searched at some point of time, and can be decreased in this way. TCAM lookup algorithms differ according to the ways in which the prefixes are distributed across the TCAM blocks. They can be divided in two ways, based on: key-IDs, or subtrees. In the first, straightforward case, the key-ID comprises several bits of the IP address, and it determines the address of the TCAM block at which this IP address will be placed [27]. Then, the subtables might have uneven sizes, and, consequently, the TCAM consumes more power because its larger blocks consume more power [29,31].

A comprehensive comparison of the TCAM partitioning schemes based on subtrees is given in [25]. In the simplest scheme, each block (or bucket) of a partitioned TCAM stores prefixes of one subtree. In this way, it is fairly easy to find the bucket of some prefix using its covering prefix. A covering prefix of some bucket is the nearest ancestor of its subtree root which contains a prefix. The longest prefix match of a given IP address can be found in the bucket whose covering prefix is the longest prefix match of this IP address among all the covering prefixes. So, the bucket in the partitioned TCAM can be found by using another TCAM comprising only covering prefixes. The simplest architecture for searching the partitioned TCAM is given in Fig.2. Index TCAM (ITCAM) comprises covering prefixes. Output of the ITCAM is the index of the covering prefix which is the longest prefix match of the given IP address. This index points to the memory location in the index SRAM (ISRAM) memory which determines starting location of the bucket in the data TCAM (DTCAM), and the size of this bucket. Output of DTCAM determines the location in the data SRAM (DSRAM) memory where the next-hop information is stored.

Several algorithms for dividing the tree into subtrees were proposed. Their main goal is to even the bucket sizes as much as possible. First, such a strategy minimizes the size of the largest bucket, and, therefore, it minimizes the power consumption. Then, the easiest way to partition DTCAM is into blocks of equal size. When evenly occupied, the buckets can be filled with the smallest number of null entries so that they all have equal size. Finally, ISRAM will not need the information about the bucket sizes if the buckets are equal. Buckets can be made more even by moving subtrees from bucket to bucket, while, of course, allowing multiple subtrees in one bucket [30]. In this case, multiple covering prefixes would correspond to one TCAM bucket.

Improvements can be also made by moving some information from TCAMs to SRAMs, because SRAMs are cheaper than TCAMs, and consume less power. In the previous schemes, very limited amount of information was stored at the SRAM locations. The ISRAM entry stored the bucket index and its size in the worst case, for which 4 bytes were sufficient. The DSRAM entry stored only the next-hop information for which 2 bytes were enough. On the other side, wide QDRII SRAM memories allow communication of 144 bits in one access. Consequently, more information can be stored at ISRAM and DSRAM locations. One ISRAM entry could store indices and sizes (if they have different sizes) of multiple buckets. Then, the merged subtrees could be represented with fewer covering prefixes in ITCAM. ITCAM size will be reduced in this way. Information related to a certain subtree is extracted based on the suffix added to the covering tree in order to obtain the subtree prefix in question. The structure of ISRAM entries depends on the bucket types (if they have fixed of variable sizes), and on the type of mapping between ISRAM and DTCAM (if multiple entry of ISRAM could point to a single DTCAM location). Similarly, the DSRAM entry could store next-hops of multiple prefixes that form a subtree. In this case, DTCAM would not store all possible

prefixes, but the fewer number of covering prefixes of the small subtrees. The sizes of ITCAM and DTCAM can be reduced roughly 3-8 times depending on the system parameter at the expense of the proportionate SRAM increase, in the described way. The M-12Wb scheme was shown to require the smallest TCAM size [25]. The M-12Wb consequently has also the low power consumption since TCAM is the biggest power consumer. The M-12Wb architecture is shown in Fig.3. In M-12Wb, wide SRAMs are used for both ISRAM and DSRAM (as 2W in its name indicates), and multiple covering prefixes might be mapped to one bucket (as M-1 in its name indicates).

2.4 Balanced Parallelized Frugal Lookup Algorithm

We have proposed BPFL (Balanced Parallelized Frugal Lookup) algorithm as a lookup algorithm that is suitable for IPv4 and IPv6 lookups [1,2]. Its main advantage is very frugal memory usage which allows high level of parallelization and, consequently, high lookup speeds. It reduces the memory requirements using several techniques. First, it does not keep any data about empty subtrees. Then, BPFL minimizes the memory requirements for the subtree presentation. Finally, prefixes of non-empty subtrees are stored in balanced trees which use fixed locations so they do not need pointers.

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 a longest prefix exists. The subtree search engine then finds the longest prefix in the located subtree.

Fig.4 shows the architecture of the BPFL search engine which comprises multiple levels. The number of levels equals L_{max}/D, where L_{max} is the maximum prefix length i.e. the IP address length (32b for IPv4 addresses and 128b for IPv6 addresses), and D is the subtree depth. Module of level i processes subtrees whose depths are equal to D, and which are determined by the $(i\text{-}1)\cdot D$ bits long prefixes. So, module of level i processes only first $i\cdot D$ bits of the IP address, and finds the prefix whose length is greater than $(i\text{-}1)\cdot D$ bits and less than or equal to $i\cdot D$ 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, i.e. 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 deepest level. Finally, the external memory is accessed to retrieve the next-hop information, i.e. the output port to which a packet should be forwarded.

A module at level i contains two parts – the subtree search engine and the prefix search engine as shown in Fig.4. 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 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 Fig.4.

The subtree search engine at level i is shown in Fig.5. It processes K_{Bi} 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 $(i\text{-}1)\cdot D$ 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 $(i\text{-}1)\cdot D$ 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 child, otherwise, the next node is the left child. If the subtree prefix at the node

equals the IP address prefix, the following balanced tree levels are traversed to enable pipelining while forwarding the balanced tree index and the address of the balanced tree node containing the IP address prefix. 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 *Search_j* is used to instruct the search of the succeeding levels. 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'.

The prefix search engine is shown in Fig.6. It consists of the bitmap processor and the internal memory block. The internal memory comprises the subtree bitmaps that indicate non-empty nodes in these subtrees. 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 complete bitmap vector would unnecessarily consume the 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, a pointer to the bitmap vector is determined based on the subtree address obtained from the subtree search engine. The bitmap processor 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 memory address of the next-hop information in the external memory is determined based on the subtree address, the prefix location in the subtree bitmap and the deepest level in which the bitmap is found.

2.5 Modified BPFL Algorithm

In BPFL, only the next-hop information is stored in the external memory [1,2]. In this paper, we propose a modification of BPFL which improves its scalability by increasing the amount of information that could be stored in the external memory. In this modified BPFL, the prefix search engine and the final selector exchanged their places as shown in Fig.7. The proposed modification can significantly reduce the overall internal memory requirements, because the large part of the memory used by the prefix search engine could be moved to the external memory.

3. Performance Analysis of Lookup Algorithms

In this section, we will examine the memory requirements of representative lookup algorithms. Parallelization improves the lookup speed, but it often requires the on-chip memory whose size limits the scalability of a lookup algorithm. For this reason, we will also specify the on-chip memory requirements for the lookup algorithms, as the measure of their scalability. One chip with the control logic and its internal memory (FPGA, ASIC), and up to two external memories are assumed to be available in all cases for the fair comparison. In this way, the PCB design is kept simple. One external memory would be used for the next-hop information, and the other memory would be used for storing the information about the lookup table. The only exception is the M-12Wb algorithm, where additional two TCAM and two SRAM memories must be used.

In all analyzed lookup algorithms, we assume that the next-hop information length is H=16 bits. In these lookup algorithms, D denotes the depth of subtrees, L denotes the prefix length, L_{max} denotes the maximum prefix length (i.e. the IP address length), S denotes the number of subtrees, N denotes the number of prefixes with the next-hop information, P denotes the pointer size and M denotes the memory requirements.

3.1 Efficient Hardware Architecture for Fast IP Lookups

In EHAF, a binary tree for IPv4 addresses is split into four levels of equal depth [6]. Level 1 stores two subtrees of depth D_1=7, and the most significant bit in the IPv4 address determines which of the two subtrees should be chosen. A bitmap vector with B_1=255 bits is associated to each subtree (where B_1 is the number of total nodes in the subtree, i.e. $B_1=2^{(D_1+1)}-1$), and the next-hop information with length H is associated to each node. So, the total number of memory bits at level 1 is:

$$M_{1EHAF}=2\cdot B_1\cdot(H+1) \quad (1)$$

Level 2 stores all subtrees of depth D_2=7, while the most significant D_1+2=9 bits of the IP address are used for choosing one of these subtrees. A bitmap vector of B_2=255 bits is associated to each subtree, and the next-hop information of width H is associated to each node. So, the total number of memory bits at level 2 is:

$$M_{2EHAF}=2^{(D_1+2)}\cdot B_2\cdot(H+1) \quad (2)$$

Level 3 stores only non-empty subtrees of depth D_3=7. A bitmap vector with length $2^{(D_1+D_2+3)}$ is used for marking the existence of subtrees at level 3, because the most significant D_1+D_2+3=17 bits determine the subtree at level 3. A subtree pointer vector stores the pointers to the non-empty subtree bitmaps. The pointer length should be at least $\log_2(S_3)$, where S_3 represents the number of non-empty subtrees at level 3. Again, the subtree bitmap length is B_3=255 bits. So, the total number of memory bits at level 3 is:

$$M_{3EHAF}=2^{(D_1+D_2+3)}+2^{(D_1+D_2+3)}\cdot\log_2(S_3)+S_3\cdot B_3\cdot(H+1) \quad (3)$$

It would be impractical to use the bitmap vector for marking the existence of non-empty subtrees at level 4 since it would be too large. So, the bitmap vector for marking the existence of non-empty groups of subtrees is used. The group of subtrees is non-empty if at least one subtree in the group is non-empty. Subtrees in one group share the common most significant D_1+D_2+3=17 bits. A vector of pointers to the group bitmaps is stored, each having the length of $\log_2(G_4)$ at least, where G_4 is the number of non-empty groups at level 4. A group bitmap vector determines the non-empty subtrees in this group whose maximum number is 256 bits. A subtree pointer vector comprises pointers to the non-empty subtree bitmaps, and it is associated to the group bitmap vector. The subtree pointer length should at least be $\log_2(S_4)$, where S_4 is the number of non-empty subtrees at level 4. So, the total number of memory bits at level 4 is:

$$M_{4EHAF}=2^{(D_1+D_2+3)}+2^{(D_1+D_2+3)}\cdot\log_2(G_4)+G_4\cdot2^{(D_3+1)}+G_4\cdot2^{(D_3+1)}\cdot\log_2(S_4)+S_4\cdot B_4\cdot(H+1) \quad (4)$$

The total memory required by EHAF, M_{EHAF}, is the sum of the memory requirements for each level:

$$M_{EHAF}=M_{1EHAF}+M_{2EHAF}+M_{3EHAF}+M_{4EHAF} \quad (5)$$

To achieve one lookup per clock cycle, slices of memories carrying information about bitmaps and pointers must be accessed independently, and, therefore, they must be placed in the internal memory, since there is at most one external memory available for the lookup table information. We choose to place bitmaps for non-empty subtrees at level 3 in the external memory, since they require the largest capacity. EHAF is not suitable for IPv6 addresses as the bitmap vectors for levels that correspond to the prefixes longer than 32 bits would be too large for practical implementation.

3.2 Parallel Search Trie-based Scheme

In PSTS (Parallel Search Trie-based Scheme), all prefixes are pushed so that their lengths are multiple of eight (8, 16, 24 and 32 bits), [7]. Prefixes of equal length represent one level, and the levels are searched in parallel. Level 1 stores a bitmap vector for L_1=8 bits long prefixes, and the size of the bitmap is 2^{L1} bits. The next-hop information of length H is associated to each node. Therefore, memory requirements for level 1 are:

$$M_{1PSTS}=2^{L_1}\cdot(H+1) \quad (6)$$

Level 2 stores also a complete bitmap vector for L_2=16 bits long prefixes, and the size of the bitmap is 2^{L_2} bits. Two additional bitmaps of the same length are stored, and they are used for determining the existence of groups of prefixes at level 3 and 4 which share the same most significant L_2=16 bits of the IP address. A vector of pointers is associated to each of these bitmaps, where one pointer is P_2 bits long, and it is associated to a group of B_2 bits in bitmap. Therefore, the memory requirements for level 2 are:

$$M_{2PSTS}=(2\cdot 2^{L_2})/B_2\cdot(P_2+B_2)+2^{L_2}\cdot(H+1) \quad (7)$$

Level 3 stores the bitmaps that determine the existence of prefixes with valid next-hop information in non-empty groups of level 3 prefixes that are L_3=24 bits long. Two more vectors are associated to these group bitmaps. The first vector is the next-hop bitmap whose element is set to 1 for the first prefix that was made by pushing of a certain prefix, and it is 0 for other prefixes which were made by pushing this prefix. In this way, the next-hop information corresponding to some prefix is not unnecessarily replicated for all the prefixes obtained by pushing the original prefix. The second vector represents the pointers with length P_3 to the locations with the next-hop information. Memory requirements for level 3 are:

$$M_{3PSTS}=G_3\cdot(2^{(L_3-L_2)}+2^{(L_3-L_2)}+P_3)+N_3\cdot H \quad (8)$$

where G_3 is the number of non-empty groups of prefixes at level 3, and N_3 is the number of prefixes with lengths in the range 17-24 bits.

Level 4 has the same structure as level 3, only the number of prefixes in one group is longer - it is equal to $2^{(L_4-L_2)}$, where L_4=32 bits is the length of prefixes at level 4. So, the memory requirements for level 4 are:

$$M_{4PSTS}=G_4\cdot(2^{(L_4-L_2)}+2^{(L_4-L_2)}+P_4)+N_4\cdot H \quad (9)$$

where G_4 is the number of non-empty groups of prefixes at level 4, and N_4 is the number of prefixes with lengths in the range 25-32 bits.

The total memory required for PSTS, M_{PSTS}, is the sum of the memory requirements for each level:

$$M_{PSTS}=M_{1PSTS}+M_{2PSTS}+M_{3PSTS}+M_{4PSTS} \quad (10)$$

Since the information about the lookup table that must be considered in each step is too large, it cannot be pulled from the external memory, but it must be stored in the internal memory. Similarly as EHAF, PSTS is not suitable for IPv6 addresses because bitmaps become too large as the prefix lengths increase.

3.3 Parallel Optimized Linear Pipeline

In POLP subtrees of original binary tree are distributed across Q pipelines, so that they contain similar numbers of nodes [4]. Each pipeline contains K_S stages, where each stage has one memory that stores F nodes of subtrees associated to the corresponding pipeline. Nodes of subtrees in one pipeline are distributed across the stages so that they have similar numbers of nodes. The only rule is that the descendants of one node must be in the following stages to enable the pipelining. Each location in the stage memory contains the pointer that determines the location of the children, and the next-hop information of length H associated to the node. Pointer to the children nodes comprises the stage index ($\log_2 K_S$ bits) and the location in the corresponding stage memory ($\log_2 F$ bits). Based on the first I initial bits of the IP address, the pipeline is determined, along with the location of the subtree's root in the first stage of the pipeline.

Mapping of subtrees to the pipelines is stored in the selector that has 2^I locations, and the pipelines are indexed with $\log_2(Q)$ bits. The total memory requirements M_{POLP} for POLP are:

$$M_{POLP}=Q\cdot K_S\cdot F\cdot(\log_2 K_S+\log_2 F+H)+2^I\cdot\log_2(Q) \quad (11)$$

The next-hop information can be placed at the node's location, or in the external memory where its location would be determined by the node's location. It does not make sense to place any of the stage memories externally, since they are of similar size and their number is too large (around 200 for IPv4).

3.4 Fast Internet Protocol Lookup

In FIPL, m-ary tree structure is used in which the strides of length L=4 bits are used [5]. Each node contains the bitmap of internal nodes (2^L bits), the bitmap that determines the existence of children nodes (2^L bits), a pointer to the next-hop information associated to the internal nodes (P_{NH} bits), and a pointer to the head of the children array (P_{CA} bits). The total memory requirements, M_{FIPL}, are:

$$M_{FIPL}=2^L\cdot K_A\cdot(2^L+2^L+P_{NH}+P_{CA})+K_B\cdot(2^L-1)\cdot H \quad (12)$$

where K_A is the number of the 2^L node arrays, and K_B is the number of the internal bitmaps with at least one valid next-hop information.

3.5 Priority Tree

PT is created from the binary tree by moving the prefixes to the upstream empty nodes [18]. A node in the tree contains pointers to the left and right child, the next-hop information, the prefix and the priority indicator. The priority indicator specifies the type of the node. In the case of a priority node, it stores the prefix that has been moved from some of its descendants. Lookup is performed in the same manner as in the case of a binary tree. The tree is traversed and in the case of a priority node, its prefix is compared to the IP address. If the match is found the lookup ends at that node, otherwise the lookup continues if the corresponding child exist. The number of nodes in a tree is equal to number of existing prefixes. The total memory requirements, M_{PT}, are:

$$M_{PT}=N\cdot(2\cdot\log_2 N+H+1+L_{max}) \quad (13)$$

The number of levels in a priority tree might be as large as in binary tree. Therefore, the PT lookup is slow requiring multiple memory accesses. The speed of one lookup per cycle can be achieved by parallelization when the entire lookup table is stored on chip. An external memory could be used for storing the next-hop information.

3.6 Classified Multi-Suffix Trie

CMST is based on the multibit priority tree [19]. For each child, a CMST node contains two fields. The first field contains the suffix of the longest prefix that is the descendant of the given child and the corresponding next-hop. The second field contains the next-hop information associated to the given child. The CMST node also stores the pointers to the corresponding prefix tree and children array [20]. The prefix tree comprises internal nodes of the multibit tree with prefixes which were not moved. Each node in the prefix tree stores the suffix value, the next-hop and left/right child pointers. Total memory requirements for the CMST tree, M_{CMST}, are:

$$M_{CMST}=\sum_{i=1..Lmax/L} K_i\cdot[2^L\cdot(L_{max}-i\cdot L+H)+2^L\cdot H+P_{PT}+P_{CA}] \quad (14)$$

where K_i is the number of CMST nodes at level i, P_{PT} is the prefix tree pointer length and P_{CA} is the children array pointer length.

Total memory requirements for prefix trees M_{PT} are:

$$M_{PT}=N_{PT}\cdot(L\text{-}1+H+2\cdot P_{PTN}) \quad (15)$$

where N_{PT} is the total number of prefixes stored in prefix trees and P_{PTN} is the length of the pointers to the children in a prefix tree.

Similarly as in the case of PT, the speed of one lookup per cycle is possible to achieve through parallelization when the lookup table is placed on chip, i.e. in the internal memory. The next-hop information could be stored externally.

3.7 Offset Addressing Lookup

In OAL, all prefixes are pushed to the tree's leaves [24]. After that, all internal nodes are indexed from the top-left to the bottom-right. Each internal node contains child indicators and offsets. Child indicator is set when the corresponding child is a leaf. Offset is used for traversing the tree. Address of the left child is calculated as the sum of the parent's address and the offset value. In the case where both children are internal nodes, the right child is at the location which is next after the left child. The search ends at the node whose child is leaf. Next-hops are stored in the next-hop memory divided into K_B buckets. Each bucket contains N_B (prefix, next-hop) pairs. The longest prefix match is passed through K hash functions in parallel to determine the buckets that can store the corresponding next-hop information. A new prefix is added to the shallowest bucket only. However, as the number of pairs in buckets change during time, all buckets must be searched. Total memory required for storing internal nodes of the tree, M_{IN}, is:

$$M_{IN}=K_{IN}\cdot(2+P_{OF}) \quad (16)$$

where K_{IN} is the number of tree's internal nodes and P_{OF} is the number of bits used for offset value.

Total memory requirements for the next-hop memory M_{NH} are:

$$M_{NH}=K_B\cdot N_B\cdot(L_{max}+H) \quad (17)$$

Since the prefixes are found using binary tree, the lookup process is slow unless the parallelization is applied. Parallelization can be implemented by storing the tree in the internal memory.

3.8 Pruned Fast Hash Table

In [21], the PFHT lookup algorithm that uses Bloom filters and hashing tables was proposed and analyzed. For each prefix length L_i, a Bloom filter is configured. When a new prefix with length L_i is added, the corresponding Bloom filter first calculates K_i buckets in the hash table with T_i locations, and, then, increments the counters, C_i bits long, associated to these buckets in the hash table. If one of the counters is 0 for the given IP address, the prefix doesn't exist in the lookup table. The probability of false positive can be made very low if T_i and K_i are large enough to support the given number, N_i, of prefixes with length L_i. Each hash table stores the link lists of the prefixes and their next-hop information, and the pointers of length P_i to these link lists. The total memory required by PFHT, M_{PFHT}, is:

$$M_{PFHT}=\sum_{i=1..Lmax}[(C_i+P_i)\cdot T_i+(L_i+H+P_i)\cdot N_i] \quad (18)$$

Since Bloom filters are accessed simultaneously, they need to be stored internally. Then, the corresponding bucket is accessed using its pointer that could be stored in the external memory with the pointers to all other buckets. The selected bucket might have to be accessed multiple times if more prefixes are stored in it. Since the probability that the number of prefixes in a bucket is larger than 2 is very low, the bucket prefixes with their next-hops could be stored in the second external memory with all other buckets. Multiple prefixes of one bucket could be read with one memory access to wide SRAM, since their number is small. The probability that one lookup is performed per memory access is close to 1.

3.9 Many-1 Two-level TCAM with Wide Memory Lookup

In [25], many TCAM schemes were analyzed and compared in terms of TCAM size and TCAM power consumption. It was concluded that the M-12Wb lookup algorithm requires the TCAM of the smallest size, and that it consumes the least amount of power. Formula for the worst-case TCAM size was given in [25], however, for the real routing tables, the required TCAM size is below this worst-case size. The M-12Wb architecture given in Fig.3 comprises two TCAM memories and two wide SRAM memories (the word that can be read from the wide SRAM in one clock cycle is W=144b long). The first TCAM memory (ITCAM) contains N_C covering prefixes. ISRAM has the same number of locations as ITCAM and it is indexed by ITCAM. It contains suffixes of the corresponding ITCAM prefixes and each suffix points to one bucket in DTCAM. There are K_B buckets of equal size N_B in DTCAM, and multiple ISRAM suffixes can point to each of these buckets. Suffixes and the corresponding next-hops are stored at the DSRAM location that corresponds to one subtree prefix in DTCAM. The total memory requirements of the M-12Wb algorithm are:

$$M_{\text{M-12Wb}}=M_{\text{TCAM}}+M_{\text{SRAM}}=(N_C+K_B\cdot N_B)\cdot L_{max}+(N_C+K_B\cdot N_B)\cdot W \quad (19)$$

We consider M_{TCAM} to be the amount of the on-chip memory since TCAM implements processing for the IP lookup.

3.10 Balanced Parallelized Frugal Lookup

In BPFL, a binary tree is split into levels, and only non-empty subtrees are stored at each level [1,2]. Levels have the same depth D. Each level uses two memory slices – a slice for storing subtree prefixes in order to find the right one, and a slice for storing subtree bitmaps. These slices are part of the subtree search engine and the prefix search engine, respectively. The subtree prefixes are stored in the nodes of balanced trees. Level i contains K_{Bi} balanced trees, and each balanced tree contains K_{Ni} nodes (i.e. subtree prefixes), where the subtree prefix length equals to $L_i=(i\text{-}1)\cdot D$. Memory slice for the subtree bitmaps is divided in two parts for storing sparse subtree bitmaps and for storing either additional nodes or complete bitmaps. A sparse subtree bitmap comprises only N_L indices of the existing nodes (prefixes) in sparsely populated subtrees and an overflow pointer of length P. The indices are D+1 long. If the number of the existing nodes in the subtree is greater than N_L, but lower than N_H, the overflow pointer points to the indices of additional subtree nodes (in steps of Δ nodes); otherwise, the complete bitmap vector is used. Two thresholds, N_L and N_H, are chosen to allow more subtle adjustment of memory usage to the statistical properties of the lookup tables. Since static addressing is used, no pointers beside the overflow pointers are needed, and the memory requirements are minimized. Memory required for subtree prefixes at level i is:

$$M_{SPi}=K_{Bi}\cdot K_{Ni}\cdot L_i \quad (20)$$

Memory required for subtree bitmaps at level i are:

$$M_{SBi}=K_{Bi}\cdot K_{Ni}\cdot(N_L\cdot(D+1)+P)+S_{Li}\cdot(D+1)+2^{(D+1)}\cdot S_{Hi}$$
$$S_{Li}=\sum_j j\cdot\Delta\cdot S_j \quad (21)$$

where S_{Li} is the number of locations reserved for subtrees that contain more than N_L nodes and less than N_H nodes, S_j is the number of sparsely populated subtrees whose number of nodes is in range $(N_L+(j-1)\cdot\Delta, N_L+j\cdot\Delta)$, and S_{Hi} is the number of densely populated subtrees at level i that contain more than N_H nodes. Selector chooses a result of the deepest level as it represents the best match and accesses the corresponding next-hop information of length H. The memory requirements for the next-hop memory are:

$$M_{NH}=H\cdot\sum_{i=1..Lmax/D}[K_{Bi}\cdot K_{Ni}\cdot N_L+S_{Li}+2^{(D+1)}\cdot S_{Hi}] \quad (22)$$

Total memory requirements of BPFL, M_{BPFL}, are:

$$M_{BPFL}=M_{NH}+\sum_{i=1..Lmax/D}(M_{SPi}+M_{SBi}) \quad (23)$$

To achieve the speed of one lookup per cycle, search at all levels must be performed in parallel, so the information associated to different levels have to be stored internally since at most one external memory would be available for the lookup table.

3.11 Modified BPFL

In modified BPFL, the prefix search engine and the final selector exchanged their places. For such an exchange, some additional information is added to each non-empty subtree. In modified BPFL shown in Fig.6, the selector chooses the subtree of the deepest level that matches the given IP address. However, it may happen that this subtree does not contain the prefix of the IP address with the next-hop information. For this reason, the next-hop information of the closest ancestor of the chosen subtree must be stored in the prefix search engine. The best place to store this information is at the location next to the overflow pointer. Equations (20)-(23) for the calculation of memory requirements still hold, only (21) contains a small modification that includes the described next-hop information of length H:

$$M_{SBi}=K_{Bi}\cdot K_{Ni}\cdot(N_L\cdot(D+1)+P+H)+(D+1)\cdot S_{Li}+2^{(D+1)}\cdot S_{Hi}$$
$$S_{Li}=\sum_j j\cdot\Delta\cdot S_j \quad (24)$$

Modified BPFL enables storing of bitmaps of all levels in one memory (Fig.7), thus bitmaps can be now placed in the available external memory, e.g. the wide SRAM.

4. Performance Comparison

In Table 1, the memory requirements for eleven representative lookup algorithms are compared for realistic IP lookup tables [32]. IPv6 lookup tables were built based on the statistics of IPv4 real lookup tables, as today's IPv6 lookup tables are still small [33-34]. In Table 1, both the total memory requirements and the on-chip memory requirements are provided. The on-chip memory has to be collocated with the control logic on FPGA or ASIC in order to achieve the optimal parallelization. It is important to compare the on-chip memory requirements too as this memory is scarce and can represent a potential bottleneck that would limit the scalability of IP lookup. The best candidates for the external memory are the fast SRAMs that typically can store several megabytes. The on-chip memories have the smaller size since they share the chip resources with the control logic.

EHAF has the low on-chip memory requirements for IPv4 lookup tables, however, its total memory becomes too large as the lookup tables increase. PSTS also has the low memory requirements for the smaller lookup tables, however, as lookup tables become larger, its memory requirements increase unacceptably. These two algorithms are not suitable for IPv6 addressing as the bitmap vectors for deeper levels would be too long for practical implementation.

POLP has a good performance for IPv4 lookup tables and smaller IPv6 lookup tables. The number of pipelines was assumed to be Q=8, and the number of bits for the pipeline selection was set to be I=8 and I=16, in the IPv4 and the IPv6 case, respectively. The main problem of POLP is that the required number of stages in one pipeline is too large, especially for IPv6 addresses, because the nodes at different tree levels are placed into different stages. Possible solution for overcoming this problem would be using of m-ary subtrees instead of binary subtrees, which would reduce the number of stages. Main advantage of POLP is that multiple lookups can be simultaneously processed by different pipelines, and the total throughput can be very high.

FIPL shows a very good performance for IPv4 lookup tables of all sizes, however, it requires too large memory for IPv6 lookup tables. This is due to storing of very large number of nodes that hold no valid information at levels that are closer to root of m-ary tree. Namely, most prefixes have lengths in the range 32-48 bits, so the nodes corresponding to the shorter prefixes are used only for traversing the tree but they contain no valid information. A possible improvement of FIPL in the IPv6 case can be achieved by starting the search from the deeper levels.

PT has the small total memory requirements for IPv4 lookup tables, which are slightly higher for IPv6 lookup tables. As there are no empty nodes in PT, the IPv6 lookup table size does not increase significantly like in POLP and FIPL. However, the high lookup speed can be achieved only through parallelization by storing the lookup table on chip. Consequently, the on-chip memory requirements are much higher for PT than in the case of BPFL and TCAM-based lookup algorithms.

In CMST, we set the stride length to L=4. CMST has acceptable memory requirements for small and medium IPv4 lookup tables, but they become unacceptably high for large IPv4 lookup tables. In the case of IPv6 lookup tables, memory requirements of CMST are unacceptably high for all observed lookup tables. CMST nodes store the large number of bits which increases with the prefix lengths.

OAL has small memory requirements for IPv4 lookup tables, however, its memory requirements increase for IPv6 tables. Also, OAL needs to be parallelized in order to achieve the high speed. For this reason, the lookup tree based on offsets must be stored in the internal memory which is a critical resource. Consequently, OAL requires the large on-chip memory which becomes a bottleneck in the case of IPv6 addresses. Also, multiple buckets have to be accessed in the memory that stores next-hop information so multiple memory accesses are needed in this step, too. These multiple memory accesses impacts the lookup throughput.

The on-chip memory requirements of PFHT are among the lowest ones, but for the large lookup tables it requires the large total memory. PFHT is a promising candidate for further optimization, since it requires the small on-chip memory. PFHT could include prefix expansion in order to minimize the number of prefixes with different lengths, and, therefore, the number of hash tables. Also, the bitmap technique could be used to reduce the size of hash tables. Namely, the subtree prefixes can be hashed instead of prefixes while the subtree internal bitmaps can be placed in the external memory.

The M-12Wb algorithm requires the smallest total memory in the case of IPv4, but it requires the higher total memory than BPFL and modified BPFL in the case of IPv6. For the M-12Wb algorithm, the on-chip memory is in fact the TCAM memory. Compared to BPFL, the M-12Wb algorithm has the lower on-chip requirements for IPv4 addresses and higher for IPv6 addresses. Compared to the modified BPFL, the M-12Wb algorithm requires much larger on-chip memory in both cases. M-12Wb is a good candidate for IPv4 and IPv6 lookup implementation as its memory requirements are reasonable.

In BPFL and modified BPFL, the subtree depth is set to D=8, the thresholds N_L and N_H are set to 8 and 40, respectively, and step Δ is set to 8. BPFL and modified BPFL have the lowest memory requirements for IPv6 compared to all other lookup algorithms, and their lead increases as the lookup table grows. In

particular, modified BPFL requires 2.5 times smaller total memory, and 4 times smaller on-chip memory than other algorithms for the largest IPv6 lookup table considered, which has more than 300K entries. The M-12Wb algorithm requires the smaller total memory in the case of IPv4. On the other side, modified BPFL requires the smallest on-chip memory in the case of IPv4, and, again, this memory saving become more pronounced as the lookup table grows.

Interestingly, the memory requirements of BPFL and modified BPFL are lower in the case of IPv6, which is due to fact that the percentage of sparsely populated subtrees is higher for IPv6 addresses, and they require less memory. But, in the case of IPv6, the larger number of registers is used because of the longer subtree prefixes, and the larger number of subtrees. However, registers are not a critical resource. Also, it can be observed that the modified BPFL requires slightly larger total memory than BPFL. On the other side, the modified BPFL allows the larger part of the memory to be placed externally, because it allows the subtree bitmaps to be placed externally. This can be observed in Table 1 which shows that the on-chip memory requirements are significantly lower when BPFL is modified as proposed in this paper.

Complexity of updating is also important aspect of the lookup algorithms. Since updating is performed less frequently, only when the network topology changes, it is less critical than the IP lookup itself. Also, updating is centralized and can be further optimized by using central processors with multiple cores. Here, we briefly discuss updating for the lookup algorithms under the consideration.

EHAF and PSTS can perform easy updates as their structure is not highly compressed as it can be observed from their total memory requirements. In EHAF, the bitmaps are updated if they correspond to the given prefix, and possibly bitmaps should be added or deleted in the case of longer prefixes. In PSTS updates are more complicated than in EHAF, because it requires management of the next-hop bitmaps. Namely, prefixes of different length might share parts of the next-hop bitmaps. Updates in FIPL are fairly easy to perform, as FIPL is based on m-ary tree. Inserting of a new prefix, or modifying the existing one, has the same complexity as the IP lookup. Multiple ancestors of the given prefix might have to be deleted when it is to be deleted, and this may require backtracking through the m-ary tree.

In POLP, updates are generally similar to the updates that correspond to other lookup algorithms based on binary trees. However, updates in some cases can be complex. For example, when new nodes are to be added as a result of the prefix insertion, and the number of stages with empty space is too low to store all these nodes, the existing nodes should be moved to free up sufficient number of stages to fit the new nodes.

In PT and CMST updates are not complex. They are very similar to the updates in lookup algorithms based on binary/multibit trees. The main difference in the case of inserting is that some priority prefix can be swapped with the inserted prefix. After that, the priority prefix that was ejected from the node is inserted into the tree in a recursive manner. In the case of a prefix deletion, a node becomes empty so some prefixes might have to be pushed upwards to fill the empty node. Both operations have similar complexity as in the binary tree case.

In OAL updating of a tree based on offsets is complex. In OAM, entries correspond to indices of internal nodes. Therefore, in the case when new internal node is added or when the existing internal node is deleted, indices of multiple internal nodes are changed. This leads to movement of large number of entries in the memory based on offsets.

In PFHT, the updates can also be complex in some cases. In PFHT, a prefix is stored in the bucket whose counter is minimal among the buckets that are calculated by hashing. Deletion of a prefix is complex, because the counters of the corresponding buckets need to be decremented. Certain prefixes would need to be moved to some of these buckets because their counters become minimal. Consequently, each prefix should be stored in all of its calculated buckets, which implies very high memory requirements. Alternatively, more complex updating schemes must be applied.

In M-12Wb, updating is complex since it involved the selection of the covering prefixes in an optimal way. In addition, prefixes needs to be stored in TCAMs in a sorted order. Insertion or deletion of certain prefixes might require a rearrangement of large number of prefixes.

BPFL and MBPFL also in most of the cases have easy updates. However, in the case when some subtree is added or deleted, a large number of nodes might have to be moved across its balanced tree in order to keep it balanced.

5. BPFL Implementation

Encouraged by its advanced scalability, we implemented the BPFL lookup algorithm on the FPGA chip. FPGA chips are popular because of their programmability, and because of the high-level of parallelization that they can provide. The FPGA chip used for this implementation was the Altera's Stratix II EP2S180F1020C5 [35]. Tables 2 and 3 show the chip resources required for the BPFL implementation in the case of IPv4 and IPv6, respectively. In both implementations, the subtree depth is D=8. One can observe that the chip resources are fully utilized only in the case of the largest IPv6 lookup table. So, IPv4 lookup tables with more than 310K entries can be supported by the selected FPGA chip, while the IPv6 lookup tables with more than 310K entries would require more advanced FPGA chips which are available. It is clear that the higher portion of chip resources are used in the IPv6 case, due to the longer IPv6 addresses. But, the required external memory is smaller in the IPv6 case because more subtrees have the lower density, while 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, which is sufficient to support 40Gb/s links. More advanced FPGAs could achieve even higher throughputs.

6. Conclusion

In this paper, a wide range of IP lookup algorithms have been analyzed and compared in terms of their scalability. A fair comparison has been made assuming that one lookup per memory access should be achieved in order to support high bit-rates. Also, a reasonable complexity of the printed circuit board design was assumed for all the examined lookup algorithms. As a result of the presented analysis, the optimized TCAM lookup algorithm, hashed-based PHFT lookup algorithm, and BPFL were shown to be the most promising candidates for IPv6. Modified BPFL has turned out to be attractive since its memory requirements are consistently lowest in the case of IPv6, and it particularly frugally uses the on-chip memory. This lead of the modified BPFL increases with the size of the IPv6 lookup tables, which is the confirmation of its scalability. PHFT also has the low on-chip memory requirements and has a potential for further optimization. Results of the presented analysis allowed us to recognize the lookup algorithms and techniques which should be explored for further development.

Acknowledgment

This work was supported by the Serbian Ministry of Education and Science, Telekom Srbija and Informatika.

References

[1] Z. Čiča, A. Smiljanić, Balanced Parallelised Frugal IPv6 Lookup Algorithm, IET Electronics Letters, 47(17) (2011) 963-965.

[2] Z. Čiča, L. Milinković, A. Smiljanić, FPGA Implementation of Lookup Algorithms, in: Proceedings of International Conference on High Performance Switching and Routing 2011, Cartagena, Spain, July, 2011.
[3] Z. Čiča, A. Smiljanić, Frugal IP Lookup Based on Parallel Search, in: Proceedings of International Conference on High Performance Switching and Routing 2009, Paris, France, June, 2009.
[4] W. Jiang, Q. Wang, and V. K. Prasanna, Beyond TCAMs: An SRAM-based Parallel Multi-Pipeline Architecture for Terabit IP Lookup, in: Proceedings of IEEE INFOCOM 2008, Phoenix, USA, April, 2008.
[5] D. Taylor, J. Lockwood, T. Sproull, J. Turner, D. Parlour, Scalable IP Lookup for Programmable Routers, in: Proceedings of IEEE INFOCOM 2002, New York, USA, June, 2002.
[6] 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, 150(1) (2003) 43-52.
[7] R. Rojas-Cessa, L. Ramesh, Z. Dong, L.Cai, N. Ansari, Parallel-Search Trie-based Scheme for Fast IP Lookup, in: Proceedings of GLOBECOM 2007, Washington D.C., USA, November, 2007.
[8] W. Eatherton, Z. Dittia, G. Varghese, Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates, ACM SIGCOMM Computer Communication Review, 34(2) (2004) 97-122.
[9] N. Yazdani, P. Min, Fast and Scalable Schemes for the IP Address Lookup Problem, in: Proceedings of IEEE Conference on High Performance Switching and Routing 2000, Heidelberg, Germany, June, 2000.
[10] S. Kumar, M. Becchi, P. Crowley, J. Turner, CAMP: Fast and Efficient IP Lookup Architecture, in: Proceedings of ANCS ’06, San Jose, USA, December, 2006.
[11] H. Song, J. Turner, J. Lockwood, Shape Shifting Tries for Faster IP Route Lookup, in: Proceedings of ICNP 2005, Boston, USA, September, 2005.
[12] S. Sahni, K.S. Kim, Efficient Construction of Multibit Tries for IP Lookup, IEEE/ACM Transactions on Networking, 11(4) (2003) 650-662.
[13] R. Sangireddy, N. Futamura, S. Aluru, A. Somani, Scalable, Memory Efficient, High-Speed IP Lookup Algorithms, IEEE/ACM Transactions on Networking, 13(4) (2005) 802-812.
[14] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small Forwarding Tables for Fast Routing Lookups, in: Proceedings of ACM SIGCOMM ’97, Cannes, France, September, 1997.
[15] M. Ruiz-Sanchez, E. Biersack, W. Dabbous, Survey and Taxonomy of IP Address Lookup Algorithms, IEEE Network, 15(2) (2001) 8-23.
[16] V. Srinivasan and G. Varghese, Fast Address Lookups using Controlled Prefix Expansion, in: Proceedings of ACM Sigmetrics 98, Madison, USA, June, 1998.
[17] S. Nilsson, G. Karlsson, IP-Address Lookup Using LC-Tries, IEEE Journal on Selected Areas in Communications, 17(6) (1999) 1083–1092.
[18] H. Lim, C. Yim, E. Swartzlander, Priority Tries for IP Address Lookup, IEEE Transactions on Computers, 59(6) (2010) 784-794.
[19] S.Y. Hsieh, Y.C. Yang, A Classified Multi-SuffixTrie for IP Lookup and Update, IEEE Transactions on Computers, April, 2011.
[20] M. Berger, IP Lookup with Low Memory Requirement and Fast Update, in: Proceedings of HPSR 2003, Torino, Italy, June, 2003.
[21] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing, in: Proceedings of SIGCOMM ‘05, Philadelphia, USA, August, 2005.
[22] A. Broder, M. Mitzenmacher, Using Multiple Hash Functions to Improve IP Lookups, in: Proceedings of IEEE INFOCOM 2001, Anchorage, USA, April, 2001.
[23] M. Bando, N.S. Artan, J. Chao, FlashLook: 100-Gbps Hash-Tuned Route Lookup Architecture, in: Proceedings of International Conference on High Performance Switching and Routing 2009, Paris, France, June, 2009.
[24] K. Huang, G. Xie, Y. Li, A.X. Liu, Offset Addressing Approach to Memory-Efficient IP Address Lookup, in: Proceedings of INFOCOM 2011, Shanghai, China, April, 2011.

[25] W. Lu, S. Sahni, Low-Power TCAMs for Very Large Forwarding Tables, IEEE/ACM Transactions on Networking, 18(3) (2010) 948-959.
[26] B. Agrawal, T. Sherwood, Modeling TCAM Power for Next Generation Network Devices, in: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS 2006, Austin, USA, March, 2006.
[27] D. Lin et al., Route Table Partitioning and Load Balancing for Parallel Searching with TCAMs, in: Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium 2007, Long Beach, USA, March 2007.
[28] V.C. Ravikumar, R.N. Mahapatra, L.N. Bhuyan, EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup, IEEE Transactions on Computers, 54(5) (2005) 521-533.
[29] X. Zhang, B. Liu, W. Li, Y. Xi, D. Berminghem, X. Wang, IPv6-oriented 4*OC768 Packet Classification with Deriving-Merging Partition and Field-Variable Encoding Algorithm, in: Proceedings of INFOCOM 2006, Barcelona, Spain, April, 2006.
[30] F. Zane, G. Narlikar, A. Basu, CoolCAMs: Power-Efficient TCAMs for Forwarding Engines, in: Proceeding of INFOCOM 2003, San Francisco, USA, March/April, 2003.
[31] K. Zheng, C. Hu, H. Liu, B. Liu, An Ultra-high Throughput and Power Efficient TCAM-based IP Lookup Engine, in: Proceedings of INFOCOM 2004, Hong Kong, China, March, 2004.
[32] http://data.ris.ripe.net
[33] M. Wang, S. Deering, T. Hain, L. Dunn, Non-random Generator for IPv6 Tables, in: Proceedings of IEEE Symposium on High-Performance Interconnects 2004, Stanford, USA, August, 2004.
[34] http://bgp.potaroo.net
[35] http://www.altera.com

Table 1
Comparison of memory requirements of IP lookup algorithms

Lookup Algorithm	Lookup Table Size	IPv4		IPv6	
		Total Mem. Requirements [MB]	On-chip Mem. Requirements [MB]	Total Mem. Requirements [MB]	On-chip Mem. Requirements [MB]
EHAF	70K	3.53	0.54	-	-
	140K	5.33	0.77	-	-
	310K	9.64	1.17	-	-
PSTS	70K	1.62	1.36	-	-
	140K	5.91	5.53	-	-
	310K	12.02	11.34	-	-
POLP	70K	1.69	0.87	1.25	0.63
	140K	2.93	1.51	7.32	3.98
	310K	5.49	2.91	13.37	7.27
FIPL	70K	1.34	0.16	15.24	12.21
	140K	2.22	0.28	27.00	21.61
	310K	3.90	0.41	42.51	33.16
PT	70K	0.65	0.49	0.93	0.63
	140K	1.35	0.98	1.88	1.33
	310K	2.98	2.11	4.16	3.07
CMST	70K	1.36	1.02	6.27	5.8
	140K	2.85	2.01	13.41	12.57
	310K	6.32	3.96	29.95	28.33
OAL	70K	1.09	0.32	2.70	1.64
	140K	2.23	0.58	5.36	3.14
	310K	4.19	1.11	10.99	5.75
PFHT	70K	2.04	0.26	2.24	0.26
	140K	4.27	0.51	4.66	0.50
	310K	9.65	1.10	10.50	1.09
M-12Wb	70K	0.42	0.05	0.84	0.19
	140K	0.81	0.11	1.72	0.39
	310K	1.76	0.23	4.02	0.92
BPFL	70K	0.67	0.15	0.52	0.26
	140K	1.46	0.29	1.11	0.47
	310K	3.26	0.47	1.54	0.76
MBPFL	70K	0.68	0.02	0.54	0.08
	140K	1.47	0.03	1.14	0.15
	310K	3.28	0.04	1.59	0.23

Table 2
BPFL hardware resource usage in the case of IPv4

Table Size	LE	FPGA memory [Mb]	External memory [MB]	f_{max} [MHz]
70K	15.1K (11%)	1.93 (21%)	0.61	119.0
140K	19.4K (13%)	2.80 (30%)	0.80	113.6
310K	27.9K (19%)	5.60 (60%)	1.68	96.1

Table 3
BPFL hardware resource usage in the case of IPv6

Table Size	LE	FPGA memory [Mb]	External memory [MB]	f_{max} [MHz]
70K	52.1K (36%)	4.03 (43%)	0.32	110.5
140K	61.1K (43%)	6.33 (67%)	0.76	106.3
310K	89.7K (63%)	8.86 (94%)	0.95	99.9

Fig. 1.
POLP architecture

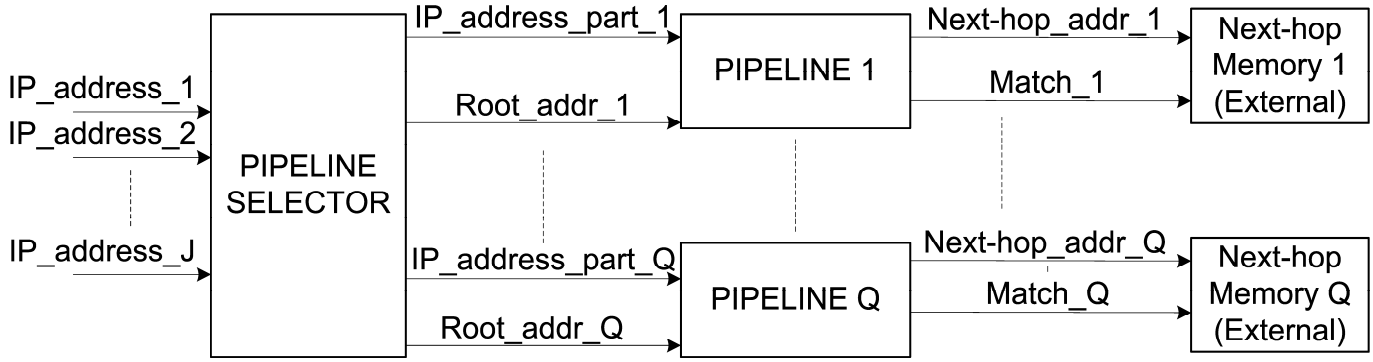

Fig. 2.
Partitioned TCAM architecture

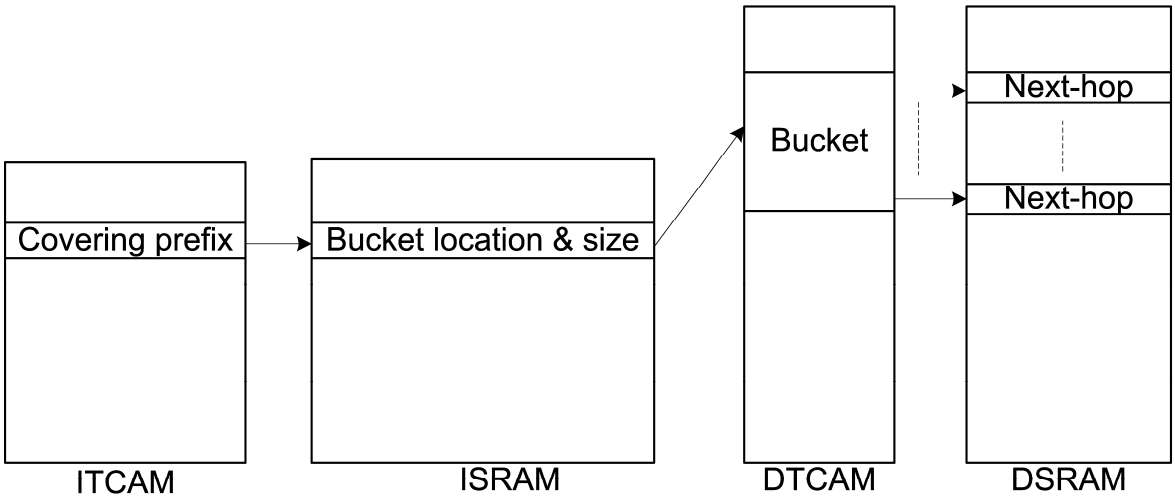

Fig. 3.
M-12Wb architecture

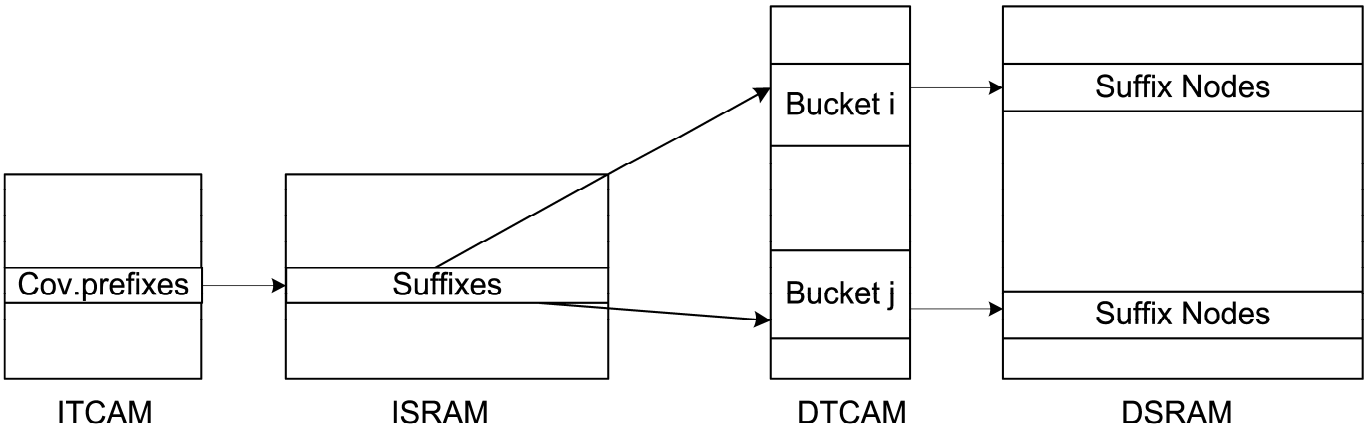

Fig. 4.
BPFL architecture

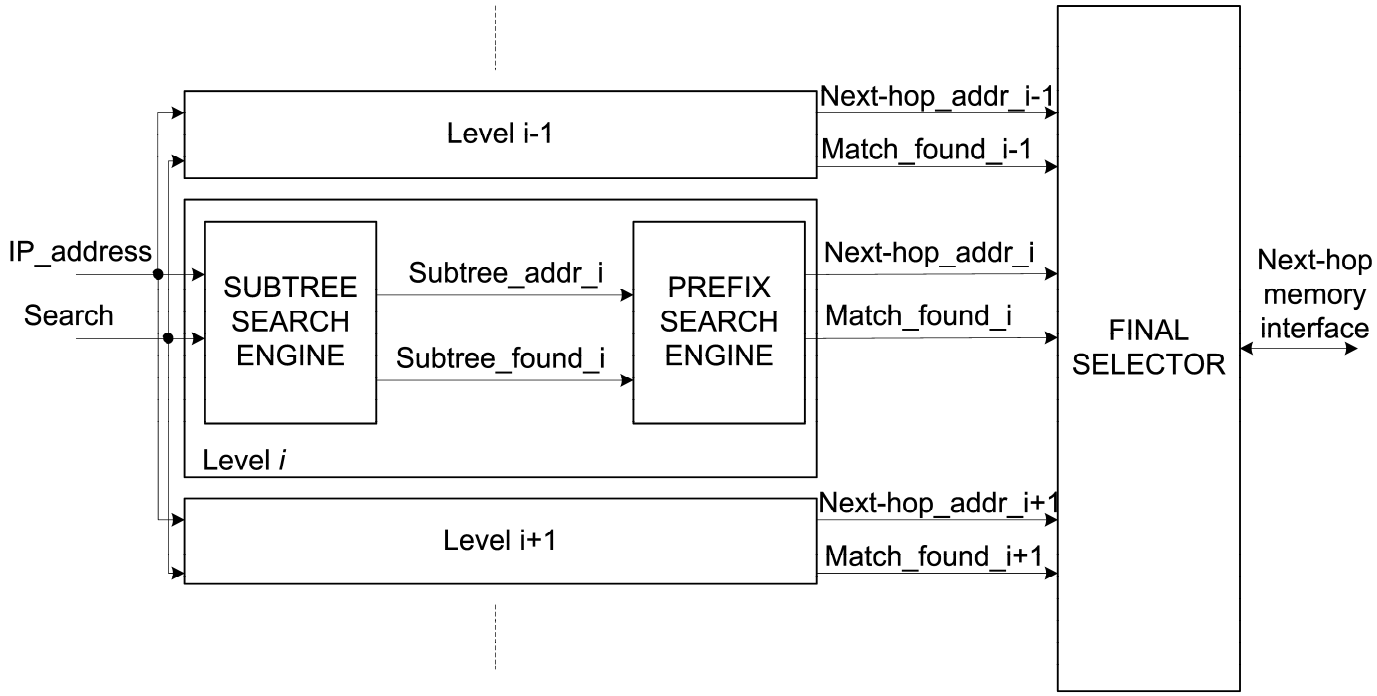

Fig. 5.
Subtree search engine at level *i*

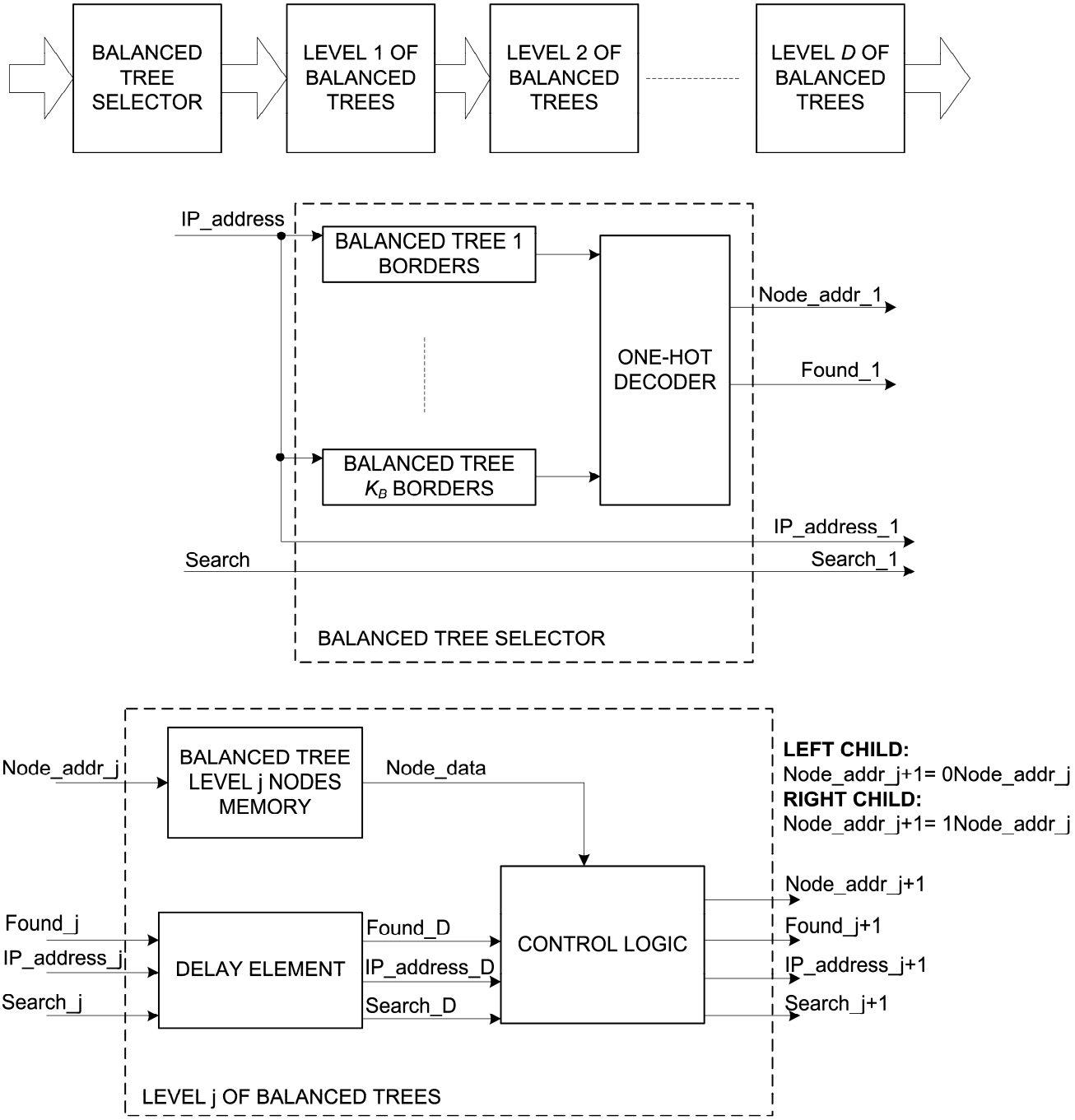

Fig. 6.
Prefix search engine at level *i*

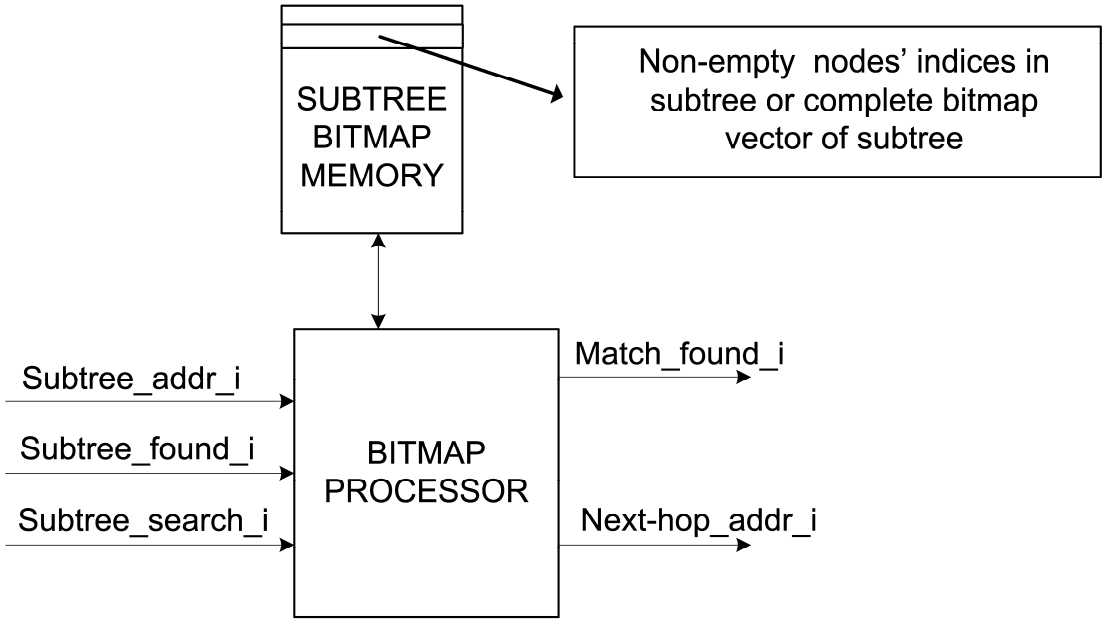

Fig. 7.
Modified BPFL architecture