

# An IP Packet Forwarding Technique Based on Partitioned Lookup Table

Mohammad J. Akhbarizadeh and Mehrdad Nourani

Center for Integrated Circuits & Systems

The University of Texas at Dallas

Richardson, TX 75083

**Abstract**—In this paper, we present an efficient IP packet forwarding methodology and architecture. This is achieved by partitioning the lookup table into the smaller ones for each output port and allowing a forwarding engine to process them in parallel. This effectively reduces the complexity of finding “the longest prefix match” problem to “the first prefix match” problem.

## I. INTRODUCTION

INTERNET consists of a mesh of interconnected IP (Internet Protocol) routers and the end-hosts connected to some of these routers. An IP router is a special purpose packet switch which requirements are described in RFC 1812 (i.e. requirements for IP version 4 (IPv4) routers) [1]. Briefly, IP routing means to forward a received IP packet to one or more router outputs (which are connected to next hops) based on the packet’s destination IP address. Each Internet router is responsible for this operation when packet switched IP communication is considered. In classful routing, the *netids* (the host identification part of an IP address) are of fixed size: 8 bits (class A), 16 bits (class B) or 24 bits (class C) and thus, the LUT (Lookup Table) search for netid matching is pretty easy [2]. The advent of Classless Inter-Domain Routing (CIDR) in 1993 [3] brought many advantages to the Internet with the expense of making the LUT search for variable length netid (prefix) matching a complex task. This paper addresses routing lookup problem for CIDR which is one of the most time and resource consuming jobs of a router.

### A. Prior Work and Main Contribution

Searching a lookup table (LUT) is a relatively exhausted field of research. Many lookup algorithms try to devise a data structure that takes advantage of the binary search tree methods which is among the mature search algorithms. A binary search tree can not be directly used due to the nature of the longest match problem. Thus, variations of the binary search methods have been presented in the literature including radix tree (a binary tree with labeled branches, also called trie) and Patricia trie (a trie with no 1-degree node) [4][5], dynamic non-recursive Patricia trie [6], Multi-ary trie [7] and the level-compressed (LC) trie [8]. Some researchers though have come up with heuristics to put prefixes together in certain way that facilitates the use of binary search. For example, binary search on trie levels and binary search on the prefix ranges are presented in [9] and [10], respectively. Unfortunately, these approaches usually suffer from large storage requirement or poor updating feature. In addition to the algorithmic and software approaches on LUT search, some researchers offered hardware solutions. For example, [12] proposes hardware implementation of tries of high radix for exact matching and [13] uses internal SRAMs for the entire routing table. Also, using content-addressable memory

(CAM) and utilizing the cache are proposed in [14] and [13], respectively. Few researchers even suggest protocol changes to avoid the LUT search problem all together [15]. Moreover, some researchers suggest partitioning LUT to reduce lookup time complexity. For example, [9] implies that LUT be partitioned into separate tables based on prefix length. A good survey of these methods can be found in [16] [17].

The main contribution of this paper is twofold. First, we present Ip packet forwarding based on partitioned lookup table (IFPLUT) method which to a large extent reduces the complexity of the route lookup operation and parallelizes the search process. Second, we offer an efficient architecture to realize this methodology. The flexibility of this architecture allows IF-PLUT to be easily integrated within routing SoCs (system-on-chip) and generic network packet processing units.

### B. Assumptions, Notations and Definitions

In this paper, we focus on the unicast (single-source, single-destination) routing. A route prefix is a variable length bit string. The length of this string can vary from 1 to 32 for IPv4 and 1 to 128 for IPv6. When necessary the prefix length can be shown using 5 and 7 bits for IPv4 and IPv6, respectively. In this paper, all IP addresses are given based on IPv4 unless we explicitly say otherwise.

Suppose  $P = \{p_1, p_2, \dots, p_M\}$  is the set of  $M$  prefixes collected by a backbone router. Assuming there are  $N$  output ports (e.g. the line cards of router), let  $Q = \{1, 2, \dots, N\}$  denote the set of these port indices. A set of all  $(p_i, q_i)$  pairs, shown as  $L$ , indicates a many-to-one function that maps  $P$  into  $Q$ . For the purpose of this paper, a forwarding table or lookup table is an organized set of  $e_i = (p_i, q_i)$  pairs, where  $p_i \in P$  and  $q_i \in Q$ . When further clarification is needed, each prefix  $p_i$  is also shown using symbolic representation of  $IPN_i/len_i$ , where  $IPN_i$  and  $len_i$  are the IP address (the prefix bit string padded with zeroes) and the length of the prefix, respectively. For example, consider the LUT shown in Table I. There are  $M = 16$  prefixes and  $N = 3$  output ports. Prefixes are shown both in the symbolic bitstream way in the second column and in  $IPN/len$  representation in the third column. In the second column, the leftmost  $len$  bits are shown as they are and the rest (which will be masked off in a lookup comparison operation) are replaced by an asterisk.

■ **Definition 1:** Two prefixes are *disjoint* if none of them is a prefix of another.

■ **Definition 2:** A prefix  $p_i \in P$  is called *enclosure* if there is at least one prefix  $p_j \in P$  ( $j \neq i$  and  $len_i < len_j$ ) such that  $p_i$  is a prefix for  $p_j$ .

■ **Definition 3:** All prefixes in  $P$  are *disjoint* if there is no enclosure in  $P$ .

TABLE I  
SAMPLE SET OF LOOKUP ITEMS (IPV4).

$i$	Prefix ( $p_i$ )	$IPN_i/len_i$	Port ( $q_i$ )
1	11000110100110*	198.152.0.0/14	1
2	11000110100*	198.128.0.0/11	2
3	1100011010011010*	198.154.0.0/16	3
4	1100011001*	198.64.0.0/10	2
5	1000101011001*	138.200.0.0/13	3
6	011110000011*	120.48.0.0/12	2
7	11000110011*	198.96.0.0/11	1
8	10001010110011*	138.204.0.0/14	2
9	100010101011*	138.176.0.0/12	2
10	1100011001110*	198.112.0.0/13	3
11	01111000101*	120.160.0.0/11	2
12	11000110010*	198.64.0.0/11	1
13	0111100011100*	120.224.0.0/13	2
14	1110111101*	239.64.0.0/10	2
15	1101011100001011*	215.11.0.0/16	1
16	01111000001110*	120.56.0.0/14	3

Note carefully that each prefix stands for a range of IP numbers and being disjoint means that the two ranges do not overlap. Being enclosure, on the other hand, means that the longer prefix correspond to a range that is a subset of the range represented by the enclosure prefix.

An assumption throughout this paper is that none of the router terminals are connected to a shared link, which means no two next hop routers are reached through the same egress port. With this assumption, there will be a one to one correspondence between our router's egress ports and next hops [2].

### C. Problem Definition

IP address lookup (the longest prefix matching problem) is a major bottleneck in the IP routing process [18]. The number of web servers doubles almost every 6 months [13]. This combined with constant growth of the speed of communication lines leaves an extremely small time budget for network routers to process and forward a packet. Devising fast, scalable and storage efficient lookup techniques have been subject of intensive research during the past 5 years.

Using CIDR routing, the longest prefix matching problem is the problem of finding an entry in  $L$  containing the longest prefix that matches the incoming packet's destination IP address. According to this definition, it is to find the narrowest range represented by a prefix in  $L$  in which the destination address falls.

## II. PARTITIONING THE LOOKUP TABLE

In this section, we first discuss our basic idea in partitioning the LUT followed by a small example. Note that from now on a superscript and a subscript show partition index and row index, respectively. For example,  $p_i^k$  shows the prefix in the  $i$ th row of the  $k$ th partitioned subset of  $L$  ( $L^k$ ).

■ **Partitioning Rule:** We partition set  $L$  into  $N$  subsets  $L^1$  to  $L^N$  such that for partition  $L^k$  we have:  $q_i^k = k \quad \forall (p_i^k, q_i^k) \in L^k$ .

The partitioning rule is very straightforward. It partitions the prefixes based on their corresponding output port. The following Lemma is a direct result of the above definition.

■ **Lemma 1:** Having partitioning rule applied to set  $L$ , for every  $(p_i^k, q_i^k)$  and  $(p_j^k, q_j^k)$  in  $L^k$ ,  $p_i^k$  and  $p_j^k$  are disjoint.

**Proof:** The lemma is true by construction. The prefixes are disjoint in the original  $L$  with respect to each next hop because

TABLE II  
PARTIAL LOOKUP TABLES (PLUT).

(a) Subset $L^1$			
$i$	Prefix ( $p_i$ )	$IPN_i/len_i$	Port ( $q_i$ )
1	11000110100110*	198.152.0.0/14	1
2	11000110011*	198.96.0.0/11	1
3	11000110010*	198.64.0.0/11	1
4	1101011100001011*	215.11.00.00/16	1
(b) Subset $L^2$			
$i$	Prefix ( $p_i$ )	$IPN_i/len_i$	Port ( $q_i$ )
1	11000110100*	198.128.0.0/11	2
2	1100011001*	198.64.0.0/10	2
3	011110000011*	120.48.0.0/12	2
4	10001010110011*	138.204.0.0/14	2
5	100010101011*	138.176.0.0/12	2
6	01111000101*	120.160.0.0/11	2
7	0111100011100*	120.224.0.0/13	2
8	1110111101*	239.64.0.0/10	2
(c) Subset $L^3$			
$i$	Prefix ( $p_i$ )	$IPN_i/len_i$	Port ( $q_i$ )
1	1100011010011010*	198.154.0.0/16	3
2	1000101011001*	138.200.0.0/13	3
3	01111000001110*	120.56.0.0/14	3
4	1100011001110*	198.112.0.0/13	3

if  $p_i$  is an enclosure for  $p_j$  then the latter's range is a subset of the former's. Now if both correspond to the same next hop then  $p_i$  range covers the  $p_j$  range. As explained in Section I-B, we assumed a one to one relationship between next hops and our egress ports, hence  $p_i$  and  $p_j$  will be associated with the same egress port and  $p_i$  already covers  $p_j$ . So  $p_j$  is redundant and can be dropped. Therefore, within  $L$  there is no enclosure and after partitioning  $L$  to multiple subsets based on their egress ports, they remain disjoint. More formally, for  $(p_i^k, k), (p_j^k, k) \in L^k$ , the prefixes  $p_i^k$  and  $p_j^k$  are disjoint. If not, it means that they were not disjoint in  $L$ . An obvious contradiction.  $\square$

If we apply the aforementioned partitioning rule to the prefix set shown in Table I we will get three disjoint subsets  $L^1, L^2$  and  $L^3$ . These partitions are shown in Table II each corresponding to one of the output ports. It can be observed clearly that within each subset the prefixes are disjoint.

Set  $L$  can form a lookup table when organized in an appropriate data structure (referred to as LUT). The same way, each subset  $L^k$  can form a partial lookup table such that there will be  $N$  partial lookup tables to which we will refer as  $PLUT^1$  through  $PLUT^N$ .

■ **Lemma 2:** Given an IP address to be looked up, searching a PLUT will result in zero or one match.

**Proof:** This is an immediate conclusion of Lemma 1. All prefixes in each PLUT are disjoint which means if the given IP address matches for example  $p_i^k$  in  $PLUT^k$ , then it can not match another prefix, e.g.  $p_j^k$  at the same time. If it happens, it means one of  $p_i^k$  or  $p_j^k$  is a prefix of the other one. An obvious contradiction.  $\square$

Based on Lemma 2, having a partitioned lookup tables, a longest match can be obtained as  $N$  parallel single match search operations, as shown in Figure 1. Now we explain this algorithm using the example of Table I. Assume that a packet has arrived to our router with a destination IP address as 198.88.191.1 (binary value 11000110 01011000 11000001 00000001). The

```

IFPLUT_Algorithm ()
Input: destination_ip (dest_ip)
Output: destination_port (m)
01: For (k := 1 to N) Do in Parallel
02:    $e^k = \text{first\_match}(\text{dest\_ip}, L^k)$ 
03: longest_match =  $e^m \ni \text{len}^m = \max_{1 \leq k \leq N} \{\text{len}^k\}$ 
04: Return (m)

```

Fig. 1. IFPLUT algorithm.

TABLE III

$L^3$  WITH SORTED PREFIXES.

<i>i</i>	Ordered Prefix ( $p_i$ )	$IPN_i/\text{len}_i$
1	01111000001110*	120.56.0.0/14
2	1000101011001*	138.200.0.0/13
3	1100011001110*	198.112.0.0/13
4	1100011010011010*	198.154.0.0/16

first step of the algorithm (lines 1 and 2) looks for possible matches in  $L^1$ ,  $L^2$  and  $L^3$  separately. As the result of this step we will have our partial matches:  $e^1 = (198.64.0.0/11, 1)$  or  $11000110\ 010^*$ ,  $e^2 = (198.64.0.0/10, 2)$  or  $11000110\ 01^*$  and  $e^3 = \emptyset$  (i.e.  $\text{len}^3 = 0$ ). This takes us to the second step (line 3) where the longest match is figured out among the partially unique matches based on their prefix length value, i.e.  $\text{longest\_match} = e^m$  such that  $\exists \text{len}^m = \max_{1 \leq k \leq N} \{\text{len}^k\}$ . So,  $\text{longest\_match} = e^1$ . Thus, 198.64.0.0/11 is the longest matching prefix for 198.88.191.1 and the arrived packet should be forwarded to a next hop connected to the egress port 1.

#### A. Advantages of IFPLUT

Our methodology offers many advantages to the layer 3 packet switching in terms of performance and cost. The most outstanding achievements are:

- 1) In each partition, all prefixes are disjoint and there can be at most one match for every prefix search. This makes the data structure of these partial lookup tables much simpler and the search operation fast and easy. This will be discussed in more details in the next subsection.
- 2) According to Lemma 2, having a partitioned lookup table at hand, a longest match can be found as  $N$  parallel single match search operations. This implies that the IFPLUT algorithm can run on parallel architecture for higher speed.
- 3) It is an implied feature that partition  $L^k$  corresponds to output port  $k$ . Therefore, we won't need to store the destination port number anymore since all the entries of a partition have the same port number.
- 4) The simplification of data structure results in the reduction of the overall lookup table size relatively by reducing storage complexity order. See Table IV and the related discussion in the next section.

#### B. The Choice of Data Structure

In each particular PLUT, the first match, if exists, is the only match. This feature effectively changes the need to find the longest match to a much simpler classic problem of finding a single unique match. On the other hand, prefixes can be sorted based on the  $IPN$  part of the  $IPN/\text{len}$  pair, a feature that does not exist in a regular LUT. Having such a sorted

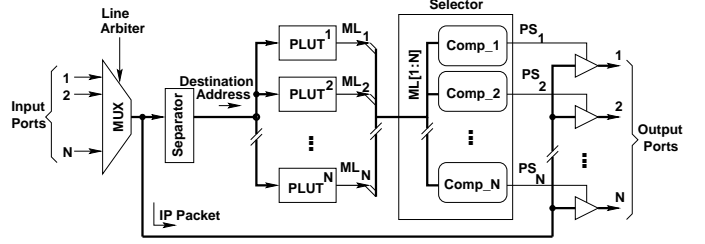


Fig. 2. IFPLUT routing architecture.

database at hand, a given IP can be looked up only by regular binary search. Table III illustrates the  $L^3$  partial lookup table sorted as described. This will bring us to the binary tree implementation of the partial lookup table. A binary search tree (or one of its more efficient variants such as heap or balanced tree) can be constructed so that any left subtree contains the prefixes with smaller absolute values of the IPN part [11]. The right subtree, contains the prefixes with bigger IPN parts. The search job is done in the conventional way. The only point is that the given destination address must be masked before being compared to the prefix. For every node of the tree, a 32 bits mask ( $\text{mask}_{31} \dots \text{mask}_0$ ) is generated out of the  $\text{len}$  field so that:  $\text{mask}_{31-i} = 1$  when  $31 - \text{len} < i < 31$ , and 0 otherwise. For example, the mask for prefix 0111100011100\* (120.224.0.0/13) will be: 11111111 11111000 00000000 00000000.

A multiway search tree is another alternative to improve the search time efficiency [10]. It is worth emphasizing that we are not restricted to any specific data structure or search algorithm. Almost any search method can be adopted by IFPLUT architecture for PLUT lookup. In Section III-D we will explain one such application using content addressable memory.

### III. THE ROUTING ARCHITECTURE

Figure 2 is a straight forward illustration of the IFPLUT Algorithm. A line arbiter is used (e.g. a multiplexor) to choose one of the  $N$  input ports for processing. The arbiter delivers one received packet to all PLUT units at the same time. The entire set of prefixes is partitioned into  $N$  units according to the partitioning rule explained in Section II. The partitions are labeled  $PLUT^1$  through  $PLUT^N$ . When an IP packet is received on input port  $i$  its destination IP address field is separated and broadcasted to all the PLUTs.  $N$  partial lookup table units process the destination IP address field of every arriving packet in parallel. Each PLUT outputs  $ML$  that is the length of its first (and only) prefix match which is the 5 bits (7 bits for IPv6) value of  $\text{len}$  of the matched entry.  $ML = 0$  means that no match is found in that particular PLUT. Theoretically,  $1 \leq \text{len} \leq 32$  and to cover this range we need at least 6 bits for  $ML$ . However, in practice the prefixes are 8 or longer, that is  $\text{len} > 7$  [2]. Therefore, we can use only 5 bits for  $ML$  such that  $ML = 0$  means no match is found in that particular PLUT and for  $8 \leq \text{len} \leq 32$  we have  $ML = \text{len} - 1$ .

As for the second step of the algorithm, these  $N$  values go to the selector block to select the longest match of all. This block generates  $N$  boolean control signals ( $PS_1, \dots, PS_N$ ) one for each egress which, if true, will put the packet on the corresponding egress. In case of unicast routing, only one of this signals are true at each time.

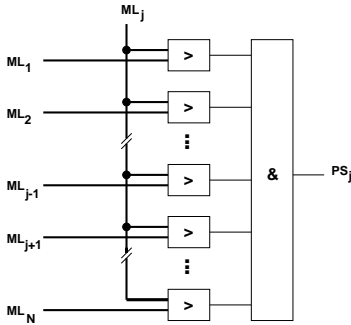


Fig. 3. One slice of selector (*Comp-j*)

The selector block must be as efficient as possible for if it has poor timing behavior, then it will kill the performance gained by the first stage. One approach is to implement it as hardwired logic. Each comparator slice of Figure 2 as shown in Figure 3.  $PS_j$  can be seen as the *AND* of the outputs of  $N - 1$  GT (*Greater Than*) blocks, labeled as  $>$  in the figure. Each GT block compares the  $j$ th match length with one of the other  $N - 1$  match lengths and generates a logic “1” only if the  $j$ th match length is greater. Therefore,

$$PS_j = \begin{cases} 1 & \text{if } ML_j > ML_i \quad \forall i \ni 1 \leq i \leq N \quad i \neq j \\ 0 & \text{otherwise} \end{cases}$$

The complexity of the GT cell depends on the bit length of its inputs which is constant 5 for IPv4 (7 for IPv6). The  $N - 1$  input *AND* block has a delay order of  $O(\log_2 N)$  assuming a 2-input *AND* gate implementation. In Section IV we present evidence showing that the cost and delay of this architecture is quite reasonable.

Another approach for the selector can be a software implementation using an embedded processor. All the PLUT blocks that have generated a match length other than 0 will submit it to the embedded microprocessor or a RISC core unit and this unit will switch on the port that corresponds to the maximum match length. Assuming that microprocessor has a comparison instruction, total number of comparisons needed for a packet received from each line is  $O(N^2)$ . For practical  $N$  (e.g. 16, 32) and a normal speed processor, the computation time is quite reasonable.

#### A. The Size of PLUT Memory Modules

If a lookup table with  $S$  entries is going to be partitioned into  $N$  PLUTs according to partitioning rule, then assuming a uniform distribution of prefixes over egress ports, each  $PLUT^k$  will have about  $S_k = S/N$  entries. In a sorted list or binary tree implementation the storage complexity for  $PLUT^k$  will be  $O(S_k)$  leading to a cumulative storage complexity of  $O(S)$  for the router.

#### B. Lookup Time

Since for every given destination address there is only zero or one match in each PLUT, the conventional binary search can be employed for finding the possible match. Let’s assume that PLUTs are implemented as binary search tree. The longest match is done in two steps according to IFPLUT algorithm (Figure 1). In the first step, *first\_match()* is a binary tree search operation that runs in parallel with a time complexity of

TABLE IV

TIME/MEMORY COMPARISON FOR DIFFERENT ALGORITHMS.

Algorithm Name	Lookup Time	Storage Requirement	Update Time
Binary Trie	$W$	$S * W$	$W$
Patricia	$W^2$	$S$	$W$
LC-Trie	$W/k$	$2^k * S * W/k$	–
Binary Search on Trie-Level	$\log(W)$	$S * \log(W)$	–
Binary Search on Interval	$\log(2S)$	$S$	–
IFPLUT+Binary Search	$\log(S)$	$S$	$\log(S/N)$
IFPLUT+TCAM	$\log(N)$	$S$	1

$O(\log_2(S_k))$  for every  $PLUT^k$ , where  $S_k$  is the size of that partition. Following the discussion in Section III-A, if each  $PLUT^k$  has a uniform  $S/N$  number of entries, the search time complexity will be  $O(\log_2(S/N))$ . We have already seen that the selector runs with a complexity of  $O(\log_2(N))$ . It’s easy to show that the overall time complexity of finding the longest matching is the sum of this two factors:  $O(\log_2(S/N) + \log_2(N))$  or simply  $O(\log_2(S))$ .

#### C. Updating PLUTs

No special hardware or procedure is needed for adding or deleting entries to/from PLUTs. Depending on the type of data structure used updating is done in the conventional way. When a new entry is going to be added to the lookup table, it is delivered to the appropriate PLUT based on its port number. Assuming that PLUTs are implemented as binary search trees, the new entry is attached to proper branch of the tree as a new leaf. Also to delete an entry, the corresponding node is deleted from the tree and the tree will be reformed. With a proper binary tree implementation (such as heap) both addition and deletion of a single entry can be fulfilled in  $O(\log_2(S_k))$  time complexity for  $PLUT^k$  where  $S_k$  is the number of entries (tree nodes) in  $PLUT^k$ . For a balanced partitioning, this is  $O(\log_2(S/N))$ .

Table IV shows a comparison between our approach and some of the well-known IP lookup search algorithms in terms of search time, storage complexity and updating time [18]. A “–” in the fourth column means that incremental update is too expensive.  $W$  is the maximum prefix length (32 for IPv4),  $N$  is the number of egress ports and  $S$  is the total number of prefixes. As seen in this table, some of the algorithms that offer good lookup time suffer from lack of efficient update time.

#### D. Implementing TCAM

A popular device for implementing fast lookup tables is Ternary Content Addressable Memory (TCAM), a form of CAM customized to allow comparisons of the variable length elements for routing tables [19]. Recent increases in lookup table size have made the use of fast memory technologies like TCAM expensive [18]. In our method each memory module that allocates one of the partitions is much smaller (e.g. by a factor of almost  $N$  in a balanced distribution) compared to the whole lookup table size. This reduces the complexity of the logic that is needed to take care of multiple concurrent outputs of TCAM words. Thus, the use of TCAM becomes more feasible. Another important advantage of IFPLUT is that we won’t need a priority encoder at the output of TCAM (see [19] for a conventional TCAM design). The reason is that each PLUT

TABLE V  
IFPLUT IMPLEMENTATION STATISTICS.

$N$	IPv4		IPv6	
	Delay [ns]	Cost	Delay [ns]	Cost
16	7.96	310	9.70	460
32	8.78	642	10.52	952
64	9.16	1303	10.90	1933
128	9.62	2625	11.41	3920

generates no more than one hit, eliminating the need for encoding based on priority. For the same reason, we won't need to sort the entries in each PLUT TCAM. This significantly reduces the update complexity of TCAM to  $O(1)$  (see Table IV) because the new entries can be added to any place in TCAM. Since conventional lookup using TCAM suffers from slow updates [19], this is probably the best advantage of using TCAM in IFPLUT architecture.

#### IV. EXPERIMENTAL RESULTS

The two main timing overhead in our architecture are: 1) memory lookup, and 2) selector delay. We implemented the PLUT memory modules as TCAM. Therefore, the delay of the first step (memory access) will be constant. As mentioned in Section III the selector part causes the main time overhead. We modeled a comparator block (Figure 3) in VHDL and let SYNOPSYS tools [20] synthesize it using a  $\lambda = 0.35\mu\text{m}$  library. Table V summarizes the timing and the gate counts for various  $N$  (i.e. 16, 32, 64 and 128 egress ports) and both IP versions (i.e. IPv4 and IPv6). The table shows the scalability advantage of our method. For example, doubling (quadrupling)  $N$  from 16 to 32 (from 32 to 128) does not increase the worst case delay with the same rate. Instead, it increases by a factor of 10.3% (9.6%) for IPv4 and a factor of 8.5% (8.5%) for IPv6. However, as reflected in Table V, the hardware cost in terms of 2-input NAND gate reported by SYNOPSYS, doubles (quadruples) for the above two scenarios. It is also worth emphasizing that the delay growth is negligible from IPv4 to IPv6 and this makes our architecture suitable for the next generation of packet processing equipments. When PLUTs are implemented using TCAM, the delay increment factor based on Table V is very small, i.e. less than 10%. This makes the delay of IFPLUT architecture independent of  $N$  to a large extent which is very important for the scalability feature.

To see another advantage of our architecture in today's market consider the OC-192 lines (e.g. processed by the optical I/O cards). OC-192 runs in 10 Gbps (gigabit per second). We now illustrate the possibility of handling OC-192 by the IFPLUT+TCAM architecture. Using  $\lambda = 0.35\mu\text{m}$  library [20], we achieved about 10 ns and 15 ns delay for selector and TCAM, respectively. This means overall delay is about 25 ns and IFPLUT can handle 40 million lookups per second. Assuming average packet size of 125 bytes this is equivalent to 4 OC-192 lines, overall throughput, at wire speed. It's a known fact that scaling the transistor dimension by  $1/\alpha$  causes delay to drop by  $1/\alpha^2$  for a fixed voltage [21]. Therefore, using smaller VLSI feature size of  $0.25\mu\text{m}$  and  $0.18\mu\text{m}$  the estimate of throughput that IFPLUT can achieve will be 8 OC-192 and 16 OC-192, respectively.

#### V. CONCLUSION

We proposed a new methodology and architecture for IP routing lookup. Our approach advocates partitioning the big lookup table to smaller tables, each associated with an output port. The IFPLUT shows promising scalability (e.g. by cascading modules) and ease of migration to IPv6 as the IFPLUT search time does not depend on the prefix length. Various QoS-aware solutions for scheduling of incoming packets and queuing at the outputs can be employed by our methodology to handle QoS efficiently. The IFPLUT works in its peak efficiency for the balanced networks in which the prefixes are distributed almost equally among PLUTs.

#### REFERENCES

- [1] F. Baker, "Requirements for IP v4 Routers," RFC 1812, <http://www.ietf.org/rfc/rfc1812.txt>, June 1995.
- [2] A. Leon-Garcia and I. Widjaja, *Communication Networks, Fundamental Concepts and Key Structures*, McGraw Hill, 2000.
- [3] V. Fuller, T. Li, J. Yu and K. Varadhan, "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC 1519, <http://www.ietf.org/rfc/rfc1519.txt>, 1993.
- [4] D. Morrison, "PATRICIA- Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of ACM*, vol. 15, no. 4, pp. 514-534, Oct. 1968.
- [5] R. Stevens and G. Wright, *TCP/IP Illustrated - Vol.1 2: The Implementation*, Addison Wesley, 1995.
- [6] W. Doeringer, G. Karjoth and M. Nassehi, "Routing on Longest Matching Prefixes," *IEEE Trans. on Networking*, vol. 4, no. 1, pp. 86-97, Feb. 1996.
- [7] V. Srinivasan and G. Varghese, "Faster IP Lookups Using Controlled Prefix Expansion," *IEEE Trans. on Computer Systems*, vol. 17, no. 1, pp. 1-40, Feb. 1999.
- [8] V. Srinivasan and G. Varghese, "IP Address Lookups Using LC-Tries," *IEEE Journal of Selected Areas in Communications*, vol. 17, no. 6, pp. 1083-1092, June 1999.
- [9] M. Waldvogel, G. Varghese, J. Turner and B. Platter, "Scalable High Speed IP Routing Lookups," *SigComm 1997*, pp 25-36
- [10] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multieay and Multicolumn Search," *IEEE Trans. on Networking*, vol. 7, no. 3, pp. 324-334, June 1999.
- [11] E. Lee, "Algorithms and Data Structures in Computer Engineering," *Jones and Bartlett Publishers*, Boston, 1991
- [12] T. Pei and C. Zukowski, "Putting Routing Tables in Silicon," *IEEE Network Magazine*, Jan. 1992.
- [13] P. Newman, G. Minshall and L. Huston, "IP Switching and Gigabit Routers," *IEEE Communication Magazine*, vol. 35, pp. 64-69, Jan. 1997.
- [14] A. McAulley, P. Tsuchiya and D. Wilson, "Fast Multi Level Hierarchical Routing Table Using Content-Addressable Memory," *US Patent 034444*, 1995.
- [15] G. Parulkar, D. Schmidt and J. Turner "IP/ATM: A Strategy for Integrating IP with ATM," *SIGCOMM'95*, Cambridge, MA, 1995.
- [16] M. Ruiz-Sanchez, E. Biersack and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network Magazine*, March 2001.
- [17] H. Tzeng and T. Przygienda, "On Fast Address Lookup Algorithms," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, June 1999.
- [18] P. Gupta, "Algorithms for Routing Lookup and Packet Classification," PhD Dissertation, Stanford University, December 2000
- [19] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAMS," *IEEE Micro*, Feb. 2001
- [20] Synopsys Design Analyzer, "User Manuals for SYNOPSYS Toolset Version 2000.05-1," Synopsys, Inc., 2000.
- [21] J. Rabaey, *Digital Integrated Circuits*, Prentice Hall, 1996.