Line Speed IP Lookup in Software Using Improved Functional Units*

Hossein Mohammadi, Behnam Robatmili, Hamid Reza Ghasemi, Nasser Yazdani, Mehrdad Nourani

Router Laboratory, ECE Department, University of Tehran, Iran
Emails: {hosm, beroy}@ece.ut.ac.ir, ghasemi@cad.ece.ut.ac.ir, yazdani@ut.ac.ir, nourani@utdallas.edu

Abstract

Due to fast increase in line speeds and number of networks, backbone routers need fast and scalable IP lookup schemes. Hardware solutions are fast but, generally, less scalable than software-based solutions. In this paper, we present generic hardware units to accelerate the IP lookup in software. Experimental results show that using DMP-Tree data structure for IP lookup, more than 30% improvement can be obtained by adding simple instructions to the running processor without any special customization. Our hardware units embedded in software-driven environment (i.e a generic processor) can accelerate other packet processing operations such as parsing, quality of service (QoS), filtering and classification.

Keywords

DMP-Tree, Hardware Acceleration, IP Lookup, Packet Parsing, Packet Processing

1. Introduction

1.1 Motivation and Contribution

Routers forward IP packets to next hops based on the incoming packets' destination addresses. Classless Inter-domain Routing (CIDR) has made this job very challenging since a packet destination address must be matched with the existing networks' addresses. Network addresses are in the form of IP prefixes. Consequently, to forward a packet, the destination address has to be matched with prefixes in the routing or forwarding tables of routers. Performing fast IP lookup is a serious challenge due to fast increase in line speeds and routing table sizes. For instance, in a line with 10Gbps rate and assuming a minimum packet of 64 bytes length, we have only 51 nano seconds to do the match and determine the output link. Meanwhile, any solution to this problem can enable us to provide a fast solution to the more challenging problems of packet filtering and classification. Packet forwarding which is based on IP lookup, filtering and classification constitute the most challenging tasks facing designer of today network processors.

In most of the previously proposed methods, the major bottleneck is the memory access time. In hardware-based solutions, this problem can be avoided by increasing memory bus bandwidth. Hardware-based solutions are, usually, less flexible and scalable. Software-based methods are performing better regarding these aspects. However, memory access can be a serious bottleneck in the most of software-based methods.

In this paper, we modify functional units of a typical RISC processor to add some simple and wisely selected instructions to take advantage of reducing time needed to perform computationally time consuming packet processing jobs. We show that adding these new instructions help every IP lookup method in general and DMP-Tree (Dynamic M-way Prefix Tree) software-based IP lookup [1],[2] in particular. Implementing these instructions is easy and feasible in high speed processors with a minimum overhead.

A brief review of proposed IP lookup methods follows. In section 2, we briefly

* This research work is partially supported by Iran Telecommunication Research Center (ITRC).

discuss DMP-Tree software-based IP lookup. In section 3, we propose the new instructions and do IP lookup using them. Simulation results of IP lookup with new instructions come in section 4. In section 5, we explain implementation method and results of synthesizing new instructions and functional units in hardware. Finally, section 6 concludes the paper and discusses the future work.

1.2 Related Works

Proposed methods for IP Lookup can be categorized based on the platforms and data structures. Hardware solutions like [10], [18], [12] are fast but less scalable and sometimes require a long update time [10], [11]. Flexibility and scalability to large routing tables are essential to lookup approach. Proposed software lookups, which uses trie and its variations like LC-Trie [9], Patricia [6], Multibit-Trie [8], are slow due to multiple memory accesses. Lulea method [5] compresses trie efficiently regarding to common prefix lengths. This method is fast but it is restricted. Therefore, it cannot scale to large routing tables while not supporting incremental updates, since each update requires building the whole trie structure. Some works like [4] combine hash with trie-based data structures. However, since no perfect hash function exists to do IP lookup efficiently for all possible routing tables, these methods are dependent on distribution of the prefixes. Tree-based methods like [7] seem to be better but it is difficult to accelerate them efficiently with hardware assistance. The reason is that their computational tasks are too complex to be added as simple instruction.

Accelerating software-based IP lookup methods with hardware support can be effective if we have enough computational work comparing to memory access time. In such cases, two different approaches can be used. First, lookup can be accelerated by adding complex memory-driven instructions to take advantage of overlapping memory access time with computation time. Second, computational tasks are speeded up by adding simple and general computational instructions. We took the first strategy in the HASIL method [3]. The result is a scalable software-based IP lookup, which was accelerated by adding two instructions that modified memory unit. This approach leads to overlapping two computational tasks with a memory access. In this paper we take the second strategy and add some simple instructions by modifying functional units of the processor and its scalability is the same because both of them use DMP-Tree software-based IP lookup.

2. Review of IP Lookup Using DMP-Tree

DMP-Tree, proposed in [1],[2], is a super set of the famous B-Tree data structure [13], which brings scalability of B-Tree to the string-matching problem in general and to LPM (Longest Prefix Match) in particular. Like B-Tree, the height of this data structure reduces by increasing number of branching in each node. This is called *branching factor* of the tree and it is an important parameter to determine the height of the tree and lookup time. Our implementation of IP lookup with DMP-Tree shows that the height of this tree data structure is proportional to logarithm of number of entries in base of branching factor. Figure 1 shows the height of DMP-Tree in different branching factors. Therefore, the number of memory access, which is a major bottleneck in IP lookup, decreases sharply using this scheme. Another important point with DMP-Tree IP lookup is that this method is fairly scalable such that by increasing routing table size from 100K entries to 1M, height of the tree just increases by one. Therefore, lookup time increases very slowly.

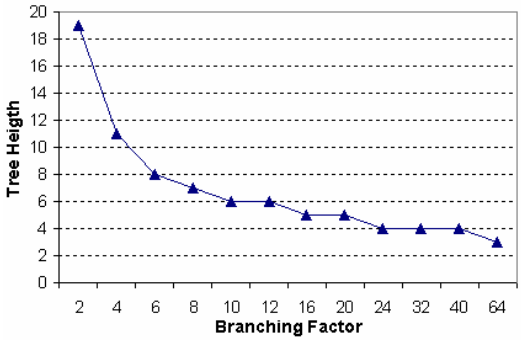

Figure 1: Maximum Height of DMP-Tree for a 100K routing table

To build a B-Tree like data structure, first, we need a method to compare and sort items, here prefixes. To do this the following definition is proposed in [1],[2] which is the basis of the DMP-Tree data structure.

Definition 1: Suppose $A = a_1 \ldots a_n$ and $B = b_1 \ldots b_m$ are two prefixes of {1, 0}. Comparing A and B (i.e. A=B, A>B and A<B) is defined

as follows. If $m=n$ then numerical values of prefixes are compared to determine which prefix is bigger. Otherwise, suppose $m<n$; numerical values of $a_1...a_m$ and $b_1...b_m$ are compared. Prefix with larger value is considered to be larger. If $a_1...a_m$ and $b_1...b_m$ are identical, then, a_{m+1} is checked. If it is 1 then A is considered to be bigger otherwise B is considered to be bigger. ■

To perform IP Lookup using DMP-Tree, we have to build DMP-Tree of the prefixes in the routing table according to Definition 1. Building method is similar to B-Tree but with a special construction rule. The rule says that '*no prefix can stay in a higher level than its prefix'*.

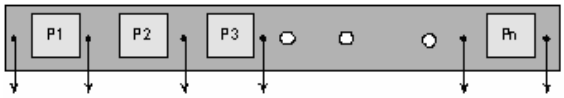

Figure 2: General view of a bucket in DMP-Tree

LPM-Search method of DMP-Tree is expected to find LPM of the incoming packets destination IP address in order to find its next-hop. This method is similar to B-Tree search with just one more step. Fig.1 depicts a general view of a bucket in B-Tree-like data structures. A bucket is a node in the tree and contains sorted elements and pointers to the next level. There are n prefixes and $n+1$ pointers in a bucket. To do LPM-Search, we begin from the *root* bucket and compare incoming IP address with prefixes in the bucket to find P_i, such that $P_i < IP < P_{i+1}$. Then, we try to find LPM of IP in the current bucket. Finally, the pointer between P_i and P_j is followed to the next level bucket. This process continues until following pointer becomes null (e.g. in a leaf).

2.1. DMP-Based Lookup Algorithm

The following pseudo-code shows LPM-Search in DMP-Tree. *MaxMatch* holds the latest longest matched prefix. *CurrentBucket* is a pointer to the current bucket (node) of DMP-Tree.

```
LPM-Search (Input: IP Address)
/* Root is a pointer to the root of DMP-Tree – MaxMatch
contains longest matching prefix found so far*/
    CurrentBucket = Root;
    MaxMatch = *;     /* Default Route */
    While CurrentBucket ≠ Null do

        Prf  = first element in Bucket;
        While  PrfixCMP(IP, Prf)==Bigger And Prf ≠
        NULL do
            Prf = Next Prefix;
        Ptr = Left pointer of Prf;
        For  each Prf in bucket do
            If Prf matches IP And Prf is
              Longer than MaxMatch Then
                MaxMatch = Prf;

        CurrentBucket = Child pointed by Ptr;
    End While
    Return MaxMatch;
End LPM-Search
```

Figure 3: DMP-Trees Lookup-search procedure called LPM-Search

Updating DMP-Tree is similar to B-Tree but it requires extra routines like *Space Division,* in order to satisfy DMP-Trees special construction rule. The update process is fast enough and it supports incremental updates. Details of DMP-Tree are beyond the scope of this paper and can be found [1],[2].

2.2. Bottleneck in IP Lookup

In traditional trie-based methods, memory access time is the bottleneck since these methods require many memory accesses and their computational tasks are few. A few methods like [7] and [1],[2] exist in which memory time is less than computation time. In this paper we use DMP-Tree software-based IP lookup since we believe this method has simpler and less computational tasks than method used in [7].

To become sure about memory effect, we have implemented and simulated running time of DMP-Tree software-based lookup and trie lookup with different memory delays. Figure 4 shows the results of the simulation. In this simulation we use a routing table of 100K entries for both trie and DMP-Tree and the branching factor of DMP-Tree is set to 8 in order to fit each bucket in a cache line of our simulation platform.

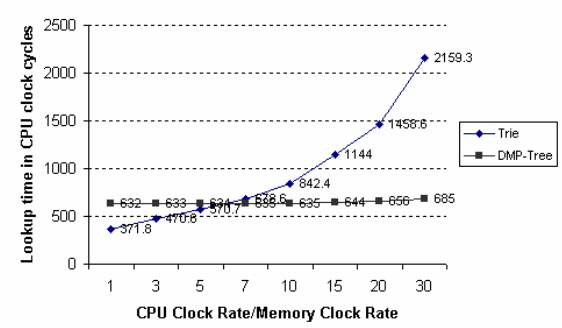

Figure 4: Lookup time in clock cycles for trie and DMP-Tree

The simulation starts from memory delay of 1 meaning that memory can be accessed in one cycle of CPU's clock. This is important because in the case of very fast memory, almost the whole lookup time is computation and software control overhead time. As the figure shows, this value is 372 for trie and 632 for DMP-Tree. When memory delay increases trie lookup time increases exponentially but DMP-Tree software lookup time remains almost unchanged. This result implies that a large part of time needed to perform a lookup in trie is memory access time. In DMP-Tree, computations take more time than memory access time and can internally overlap with memory access. Therefore, increasing memory delay doesn't affect DMP-Tree lookup time and for implementation we need to improve the computational time.

3. Packet-Driven Instructions

In this section, we reduce computation time of lookup methods by asking some simple supports from hardware to accelerate frequently executed operations. The first step is to find out these operations. In the second step we use the new instructions and perform IP lookup.

3.1. Processing Tasks and Prefix Format

Generally, for most of IP packet processing methods and especially for DMP-Tree software-based IP lookup, we can highlight these simple and frequent functions in the lookup-search procedure (see code of Figure 3):

1. Compare prefixes.
2. Checking matching of an IP address with a prefix.
3. Extracting some bits from a word.
4. Extracting prefix length from prefix format.
5. Converting (Value, Length) to the prefix format.
6. Converting prefix format to (Value, Length).
7. Bit checking instructions.

Since adding an instruction requires complex hardware design process, it is meaningful to do our best to solve some of the above needs in software.

According to the LPM-Search method in Figure 3; *Prefix Compare* and *Prefix Matching* are two main functions running many times for each lookup. For instance, in a non-optimized implementation of DMP-Tree of height 5 and branching factor of 16, each of these operations will be executed around 80 times (height * branching factor). Therefore, our first step is to optimize these two frequently used functions. Using ordinary prefix representation (zero-filled prefix, Length), straight implementation of Definition 1 to compare an IP address with a prefix, requires 13 instructions. Therefore, in our example, total prefix compares for each lookup will take approximately 1040 clock cycles using the typical MIPS [19] processor, because the IP address must be compared and matched with all prefixes in each bucket. This is too costly and we need new ideas to do it faster.

We have solved comparison problem by defining a new prefix representation format called Hosm-Format. We have proposed a new format to represent prefixes. This format significantly reduces instructions and time needed for *PrefixCompare* and *PrefixMatch* functions sharply.

Hosm-Format: Prefix can be represented by adding a zero to its tail, and then filling it with ones to make its length equal to the biggest possible prefix length plus one, here 32. (e.g.: 101* represents as 101*0*11….1.)

Theorem 1: Numerical comparison between prefixes in *Hosm-Format* is equivalent to prefix-comparison using Definition 1.

Proof: Suppose $A = a_1 \ldots a_n$ and $B = b_1 \ldots b_m$.

1. If $m=n$ then as Definition 1 offers, numerical comparison will determine the result.
2. If $m<n$, two cases are possible:
   2.1. $a_1 \ldots a_m \neq b_1 \ldots b_m$: Definition 1 says that prefix with larger substring is bigger. So, Hosm-Format acts correctly.

2.2. $a_1...a_m = b_1...b_m$: In this case, A is represented as $A=a_1...a_m011..1$ and $B=a_1...a_mb_{m+1}...b_n011..1$ in Hosm-Format. Of course in numerical comparison between A and B, b_{m+1} will determine the result and that is what Definition 1 offers.■

Since we use Hosm-Format to compare a prefix with an IP address, there are two concerns. First, what if prefix in Hosm-Format equals to IP. Second, what we can do with prefixes of length 32. The solution to the first concern is that, by assuming maximum prefix length to be 31, no prefix can be equal to IP. In the case of equal Hosm-Format representations, IP should be considered to be smaller (proof is trivial.). For the second concern, to keep prefix lengths smaller than 32, we can keep 32 bit prefixes in a simple jump table and check it first.

Prefix comparison is the basis of all LPM search methods. Using Hosm-Format, we can reduce the prefix compare time to just one clock cycle since it requires one integer-to-integer comparison, which is a common instruction in almost any general-purpose processors. Therefore, we do not need any special hardware implementation for prefix compare. Now we focus on prefix matching which is also a frequent job in the LPM-Search process.

Lemma 1: Assume *prf* is a prefix in Hosm-Format and *IP* is an IP address. If $prf \oplus IP < 2^{32-Length(prf)}$ holds then, *prf* matches *IP*.

Example: Consider prefix 1101* whose length is 4. Thus, in Hosm-Format the prefix becomes *prf=1101011..1*. Assuming *IP=1101xx..x* then we have $IP \oplus prf = 0000yy...y < 000100...0$.

Proof: $2^{32-Length(prf)}$ is a mask for prefix in which the bit before Hosm-Formats zero is set. Therefore, if IP matched the prefix, the result of XOR will be smaller than mask and if it does not match, it will not be smaller.■

Now, we can compare prefixes in one clock cycle and match them with an IP address using a straight implementation of Lemma 1, in up to 5 clock cycles. This may be good for a software implementation but because of time-consuming match function, it is slower than hardware implementations with an order of magnitude. Therefore, we would like to accelerate this software-based lookup with some easy-to-implement hardware supports. Of course, *Prefix Matching* is the first candidate to be implemented in hardware. While using Hosm-Format, implementation of *Prefix Matching* with Lemma 2 becomes straight forward. We discuss the implementation overhead in section 5.

3.2. New Instructions

By carefully examining results of section 3.1 and considering general requirements of packet processing applications (here, IP packet parsing) we can distinguish instructions presented in Table 1 to be added to accelerate the lookup process. These instructions can generally accelerate any IP lookup process since they are quite general. Most lookup methods such as DMP-Tree lookup will be accelerated since their computational tasks can fit into the requirements of these instructions.

Table 1: New instructions proposed to be added

	Inst.	Function	Example	Result
PARSE	**ebis**	extracting bits	ebis Ra #s #l Rr	Rr <= Ra & MASK[s,l]
	ebia	extracting and adjusting	ebia Ra #s #l Rr	Rr<=SHIFT_R ((Ra &MASK[s,l]),s)
	cbit	Check bit	cbit R1 #b R2	If b-th bit of R1 then R2 = 1 Else R2 = 0
PREFIX	**cpr**	create prefix	cpr Rp Rl RPx	RPx <=CreatePref (Rp, Rl)
	vpr	prefix value	vpr RPx Rv	Rv <= Value of RPx prefix
	lpr	prefix length	lpr RPx Rl	Rl <= Length of RPx prefix
	mpr	Match prefix	mpr RPx Ra ADDR	If RPx is a prefix of Ra value then jump to ADDR

cpr, vpr, lpr and *mpr* instructions take Hosm-Format as their prefix representation format and do operations like prefix matching, creation, etc. *ebis, ebia* and *cbit* are bitwise operations which do bit extraction and manipulation.

The philosophy behind selecting these instructions for hardware implementation is that, in most of packet processing applications (lookup, parsing, classification, etc.) prefix operations are executed many times. For example, in packet parsing, all we have to do is to extract some static or dynamic aligned fields from an IP packet. These tasks can be accelerated using our bit-wise operations.

Other examples are: 1) Lookup process in which prefix-matching and prefix length-extraction are frequently executed. 2) Prefix-creation and decoding (converting from natural numbers to a specific prefix representation format and vice versa) is frequent in table-update routines. Therefore, our prefix unit's instructions can help generic IP lookup methods.

4. Simulation Analysis of the New Instructions

4.1. Prefix Instructions

Using the new instructions we can simplify lookup code and gain higher speeds. We have implemented DMP-Tree lookup method in MIPS assembly language [20] and we have simulated its running time with different memory speeds. Using new instructions we can reduce code size by 13%. Since this 13% part of code (matching instruction and length extraction from a prefix) is frequently executed during a lookup, lookup time is reduced by 27.44%. Figure 5 compares running results of DMP-Tree lookup method using new instructions and without them. It is worth noting that for DMP-Tree lookup method we have used *mpr* and *lpr* and *ebia* instructions in matching phase of the LPM-Search code. Other instructions may be used in other lookup methods or in packet processing applications like parsing.

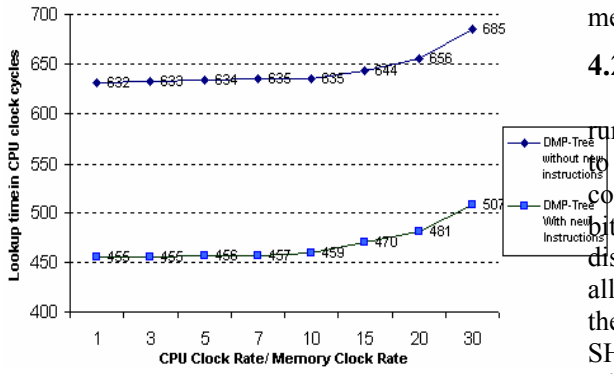

Figure 5: DMP-Tree Lookup speed with and without new instructions

We use *mpr* instruction to check matching of an IP address with a prefix and *lpr* to extract length of a prefix in Hosm-Format and *ebia* instruction to extract next-hop information from a tree element. As discussed this leads to 27.44% improvement in overall lookup speed.

As an example we consider a 2.4GHZ processor with a 333MHZ DDR-RAM. In this case, memory is 7 times slower than the processor and according to Figure 5, each lookup requires 457 clock cycles. It means that each lookup requires 190ns and consequently our method can forward 5.26 million packets per second. Assuming average packet size to be 2000 bits, this means supporting 10Gbps (OC-192) lines.

The improvement that these instructions can achieve is, of course, limited. The first reason is that when we develop a software program, control instructions like loop control branches and result checking instructions push a large overhead on system's behavior. Another reason is that in the case of DMP-Tree, the new instructions are only used to accelerate matching part. However, the *while* loop in *finding place* part of the code (see Figure 3) which finds the place of the IP address in the bucket, remains unchanged. We have not added special instructions to help DMP-Tree IP lookup in all of its tasks because we would like our newly added instructions to remain general enough and useful for a variety of lookup methods just like DMP-Tree lookup. Of course adding more complex and specialized instructions can help DMP-Tree lookup. For example, instructions added in HASIL [3] can accelerate DMP-Tree lookup process with an order of magnitude. However, those instructions cannot help other lookup methods.

4.2. Parse Instructions

Almost all of packet processing applications run on a network processor need instructions to create prefixes in a specified format, comparing and matching them and doing some bit-wise operations to extract fields. As discussed in 3.2, in packet parsing application, all we have to do is to extract some fields from the IP packet's header. This requires some SHIFT, AND, XOR and other instructions which can be substituted with our bit extraction instructions (ebis, ebia). Our implementation showed that using new instructions reduces code of the IP packet parsing process about *60%* and its running time about *50%*.

As another example, these instructions can generally help an IP packet classification

program. Because it also needs instructions to match prefixes, compare them and doing bit-wise operations.

Almost all of Quality of Service (QoS) methods require bit-wise operations to extract labels (e.g. Diffserv labels [RFC 3260]) or to implement efficient queuing mechanisms. Therefore, our instructions can generally help QoS methods.

5. Hardware Implementation

In this section, we briefly describe the overhead due to adding the new instructions in terms of area and delay. We augmented a simple 32-bit ALU (Arithmetic/ Logical unit) with our extra operations to support prefix and parse instructions. This is illustrated in Figure 6. In this figure, Parse unit and Prefix unit perform the parse and prefix instructions, respectively. As can be seen, we have considered some new signals from the ALU controller to control operation of these two additional units. However, to minimize the overhead and ease of implementation, ALU and these two units work independently.

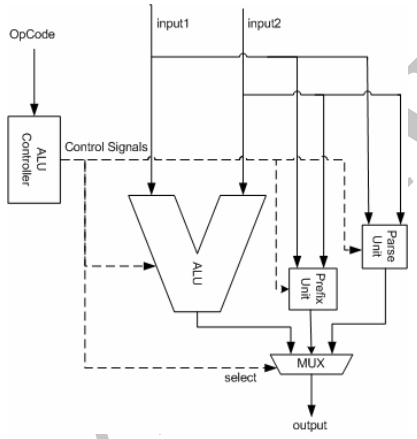

Figure 6: An ALU augmented with Prefix and Parse Units

Our simple ALU provides primary logical operations (including AND, OR, NOT, XOR, etc), integer operations (including ADD, SUB, MULT, etc) and relational operations (including equality, greater than, etc). It has two 32-bit inputs (*input1* and *input2*) and an output (*output*) of the same size. Our prefix unit includes five operators which perform the Prefix operations.

We have implemented the ALU before and after being augmented with our new units in a 0.25u ASIC (TSMC25) technology. The behaviors of Parse and Prefix units are written in VHDL and synthesized using *LEONARDO SPECTRUM 2002* [20]. These units are not complicated as evident from the list of instructions in Table 1. Details of these units are beyond the scope of this paper. Table 2 shows the result of this comparison in terms of area and critical path delay.

Table 2: The results of the hardware implementation of Prefix and Parse unit

32 bit units	Area (Gates)	Area overhead (%)	Delay (ns)
ALU	90105	-	13.32
Prefix	13203	14.65	2.73
Parse	18742	20.8	4.65
Overall	31945	35.45	-

In Table 2, the third column shows the area overhead of the additional units with respect to the ALU area. The second and fourth column clearly shows that these units are not too costly or slow. Specifically, since ALU and these two units work in mutually exclusive fashion, adding such acceleration units can only benefit the performance, for example during IP lookup.

6. Conclusion and Future works

Augmenting RISC processors is a well known method to build network processors. In this paper we climbed the first stage. We identified and added some parsing and prefix instructions to accelerate packet processing methods, in general, and packet parsing and IP lookup in particular. We also showed that DMP-Tree based software lookup is a good candidate for simultaneous need to speed and scalability in IP lookup algorithms. This method can be accelerated efficiently with our new instructions. Our method can easily support routing tables with millions of entries because tree's height is logarithmic with respect to number of routing entries on basis of branching factor.

Future works of this research includes study of packet classification and quality of service to identify their main computational bottlenecks and complete the new instructions to accelerate these applications.

References

[1] N. Yazdani and P. S. Min, "Fast and Salable Schemes for IP Lookup Problem", Proc. of IEEE Conf. on High Performance Switching and Routing, Heidelberg, Germany, June 2000.

[2] Nasser Yazdani, Hossein Mohammadi, "IP Lookup in Software for Large Routing Tables Using DMP-Tree Data Structure" Proc. of the 9th Asia Pacific Conference on Communications (APCC) 2003

[3] H. Mohammadi, B. Robatmili, N. Yazdani, M. Nourani, "HASIL: Hardware Assisted Software-based IP Lookup for Large Routing Tables", Proceeding of the 11th IEEE International conference on networks (ICON) 2003, pp. 99-105, Sydney - Australia

[4] M.Waldvogel, G.Varghese, et.al. "Scalable High Speed IP Routing Lookups", Proc. of ACM SIGCOM`97, pp. 25-35, Cannes, France, 1997.

[5] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," Proceeding of ACM SIGCOM`97 Conf., pp. 3-14, Cannes, France, 1997.

[6] W. Doeringer, G. Karjoth and M. Nassehi, "Routing On Longest Matching Prefixes,", IEEE/ACM Trans. Net. vol.4, pp. 86-97, Feb, 1996.

[7] B. Lampson, V. Srinivasan and G.Varghese, "IP Lookups Using Multiway and Multicolumn Search", Proc. IEEE Infocom`98, 1998.

[8] S. Sahni and K. S. Kim, "Efficient Construction of Variable-Stride Multibit Tries For IP Lookup", Proc. of IEEE Symposium on Applications and the Internet, SAINT, 2002.

[9] S. Nilsson and G. Karlsson, "IP Address Lookups Using LC-Tries", IEEE JSAC, Vol.17, No. 6, pp. 1083-1092, June 1999

[10] N. Yazdani and N. Salimi, "Performing IP Lookup on Very High Line Speed,", Proc. of ICT 2002, Shiraz, Iran, 2002.

[11] N. McKeown, P. Gupta, and S. Lin, "Routing Lookups in Hardware at Memory Access Speeds", Proc. of IEEE Infocom`98 Conf., pp. 1240-1247, 1998.

[12] W. E. Chen and C. J. Tsai, "A fast and scalable IP lookup scheme for high-speed networks", Proc. of IEEE ICON99, 1999.

[13] T.Cormen, C.Leiserson, R. Rivest and Stein, "Introduction to Algorithms", MIT Univ. Press, 2001.

[14] H. Y. Tzeng, "Longest Prefix Search Using Compressed Trees", Proc. of IEEE GlobCom 98 Conf., Sydney, Australia, 1998.

[15] B. Lampson , V. Srinivasan and G. Varghese , "IP Lookups Using Multiway and Multicolumn Search", Proc. of IEEE Infocom`98 Conf., pp. 1247-1256, San Francisco, CA, 1998.

[16] T. Chiueh, P. Pradhan, "High Performance IP Routing Lookup Using CPU Caching", Proc. of IEEE Infocom`99, 1999.

[17] H. Liu, "Routing Prefix Caching in Network Processor Design" , Proc. of International Conference on Computer Communications and networks (ICCCN), Phoenix, AZ, 2001.

[18] N. McKeown, P. Gupta, and S. Lin, "Routing Lookups in Hardware at Memory Access Speeds", Proc. of IEEE Infocom`98 Conf., pp. 1240-1247, 1998.

[19] D. A. Patterson, J. L. Hennessy, N. Indurkhya, "Computer Organization and Design: The Hardware/Software Interface", 2nd Edition, 1998.

[20] Mentor Graphics' Leonardo Spectrum synthesis tool. http://www.mentor.com/leonardospectrum