Fast Update Algorithm for IP Forwarding Table Using Independent Sets

Xuehong Sun[1], Sartaj K. Sahni[2], and Yiqiang Q. Zhao[1]

[1] School of Mathematics and Statistics,
Carleton University, 1125 Colonel By Drive,
Ottawa, Ontario CANADA K1S 5B6
{xsun, zhao}@math.carleton.ca
http://mathstat.carleton.ca/~zhao/

[2] Department of Computer and Information Science and Engineering,
University of Florida,
Gainesville, FL 32611, USA
sahni@cise.ufl.edu
http://www.cise.ufl.edu/~sahni/

Abstract. This paper describes a new update algorithm for Internet Protocol (IP) forwarding tables. The update algorithm is so general that it can be combined with many IP address lookup algorithms for fast update. This is done by partitioning the IP forwarding table into independent sets. Each independent set can be looked as a set of disjoint ranges, therefore any data structure for disjoint ranges search can be combined with this algorithm to support fast update. This algorithm achieves fast update by avoiding the worst-case update scenario. The number of independent sets is small, hence, the number of parallel data structures is small.

1 Introduction

An Internet system consists of Internet nodes and transmission media which connect Internet nodes to form networks. Transmission media are responsible for transferring data and Internet nodes for processing data. In today's networks, optical fibers are used as transmission media. Optical transmission systems provide high bandwidth. It can transmit data in several gigabits (OC48=2.4Gb/s and OC192=10Gb/s are common, and OC768=40Gb/s is in the near future) per second per fiber channel. Dense Wavelength Division Multiplexing (DWDM) [6] technology can accommodate more than 100 channels (2004) and more in the future in one strand of fiber. This amounts to a terabits per second transmission speed on optical fiber. In order to keep pace with this speed, the Internet nodes need to achieve this same speed of processing packets.

Internet nodes implement functions incurred by Internet system. IP address lookup is one of the main tasks performed by Internet nodes. Solutions exist to tackle the IP address lookup problem. In general, they can be divided into two groups: One is the Ternary Content Addressable Memory (TCAM)-based solution [4]; the other is algorithm-based solution. TCAM is notorious for high power dissipation and expensive. Its speed is limited by about 10ns TCAM access latency. Potentially, the on-chip SRAM solution can achieve a 5-10 times faster lookup speed than that using TCAM.

In the literature, there are a number of algorithms proposed for the IP address lookup. Surveys on address lookup algorithms were given in [5][11][9]. However, developing an algorithm with both fast lookup and update speed is still a very active research activity.

In this paper, we propose a fast update algorithm for IP forwarding table by using the concept of independent sets which is similar to that in [13]. Combined with algorithm such as in [12], a high performance dynamic algorithm can be developed. The idea is so general that it can be used to convert other static algorithms to dynamic ones.

The rest of the paper is organized as follows. In Sect. 2, the IP address lookup problem is defined. Section 3 points out the worst-case scenario for update. The concept of independent sets and the details of how to partition the forwarding table into independent sets are described in Sect. 4. In Sect. 5, the update algorithm is presented. In Sect. 6, results from previous work are highlighted. Concluding remarks are made in Sect. 7.

2 IP Address Lookup Problem

Internet Protocol defines a mechanism to forward Internet packets. Each packet has an IP destination address. In an Internet node (Internet router), there is an IP address lookup table (forwarding table) which associates any IP destination address with an output port number (or next-hop address). When a packet comes in, the router extracts the IP destination field and uses the IP destination address to lookup the table to get the output port number for this packet. The IP address lookup problem is to study how to construct a data structure to accommodate the forwarding table so that we can find the output port number quickly.

Since the IP addresses in a lookup table have special structures, the IP address lookup problem can use techniques that are different from that used in solving general table lookup problems by exploiting the special structures of the IP addresses. Nowadays, IPv4 address are used and in the future, IPv6 addresses could be adopted. We next introduce these two address architectures.

2.1 IPv4 Address Architecture

IPv4 addresses are 32 bits long. An address can be represented in dotted-decimal notation: 32 bits are divided into four groups of 8 bits with each group represented as decimal and separated by a dot. For example, 134.117.87.15 is a computer IP address at Carleton University. Sometimes we use the binary or decimal representation of IP addresses for convenience. An IP address is partitioned into two parts: A constituent network prefix (hereafter called prefix) and a host number on that network. The CIDR [7] uses a notation to explicitly mention the bit length for the prefix. Its form is "IP address/prefix length." For example, 134.117.87.15/24 means that 134.117.87 is for the network and 15 is for the host. 134.117.87.15/22 means that 134.117.84 is for the network and 3.15 is for the host. For the later case, some calculations are needed. This address has 22 bits as prefix and 10 bits as host. So, 134.117.87.15 (10000110 01110101 01010111 00001111) is divided into two parts: 10000110 01110101 010101* (134.117.84) and 11 00001111(3.15)). Sometimes, we use a mask to represent the network part. For example, 134.117.87.15/255.255.255 is equivalent to 134.117.87.15/24, since the binary form of 255.255.255 is 24 bits of 1s (note that binary form of 255 is 11111111). 134.117.87.15/255.255.253 is equivalent to 134.117.87.15/22, since the binary form of 255.255.253 is 22 bits of 1s.

We can look at the prefix from other perspective. The IPv4 address space is the set of integers from 0 to $2^{32} - 1$ inclusive. A prefix represents a subset of the IPv4 address space. For example, 10000110 01110101 010101* (134.117.84) represents the integers between 2255836160 and 2255837183 inclusive. We will define the conversion in a later section. The longer the prefix is, the smaller the subset is. For example, 10000110 01110101 010101* (length 22) has $2^{10} = 1024$ IP addresses in it; while 10000110 01110101 01010111* (length 24) has only $2^8 = 256$ IP addresses in it. We can also see that if an address is in 10000110 01110101 01010111*, it is also in 10000110 01110101 010101*. We say 10000110 01110101 01010111* is more specific than 10000110 01110101 010101*. IP address lookup is to find the most specific prefix that matches an IP address. It is also called the *longest prefix match* (because the longer the prefix is, the more specific it is).

2.2 IPv6 Address Architecture

The research on the next-generation Internet protocol IPv6 [1] was triggered by solving the IPv4 address space exhaustion problem among other things. In IPv6, the IPv6 addressing architecture [2] is used.

An IPv6 address is 128 bits long. A form similar to the CIDR notation for IPv4 addresses is used to indicate the network part of an IPv6 address. For example, "1200: 0: 0: CD30: 1A3: 4567: 8AAB: CDEF/60" means that the first 60 bits are the network part and the other 68 bits are the host part. It also represents a 60 bits length prefix. Refer to [2] for the details.

In the following sections, we use IPv4 addresses as an example to explain the concept for the purpose of simplicity.

3 The Worst-Case Scenario for Update

The longest prefix match problem is more difficult than the general table lookup problem, especially for update. There is the worst-case scenario for update in the longest prefix match. Figure 1 gives an example of the worst-case scenario. It shows that the IP addresses of prefixes P_1, P_2, $\cdots$, P_n are covered by prefix P_{n+1}. According to the longest prefix match rule, the port number for each address interval is shown in the lower part. If there is an update for P_{n+1}, e.g., the port number d is changed to e, then all, here $n+1$, the intervals with port d will be changed to e. The complexity is $O(n)$, where n is the total number of prefixes. If n is large, a lot of memory accesses will be involved. If n is small, the memory accesses incurred can be tolerant. In order to solve this problem, intuitively, we need to separate the prefix P_{n+1} from other prefixes. This leads to the concept of *independent sets* which is defined in the next section.

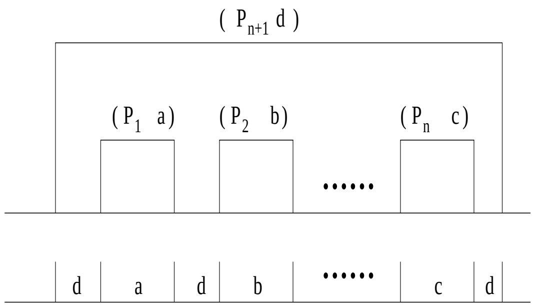

Fig. 1. The worst-case scenario for update.

4 Partitioning the Prefixes into Independent Sets

We first give the definition of independent sets and then provide an algorithm of partitioning the prefixes into independent sets. Before doing that, we need the following definition which is consistent with the concept in [12].

4.1 Definitions

Definition 1. *A prefix P represents address(es) in a range. When an address is expressed as an integer, the prefix P can be represented as a set of consecutive integer(s), expressed as $[b, e)$, where b and e are integers and $[b, e) = \{x : b \leq x < e, x \text{ is integer}\}$. $[b, e)$ is defined as the* range *of the prefix P. b and e are referred to as the* left endpoint *and the* right endpoint *of the prefix P respectively, or* endpoints *of the prefix P.*

For example, for the 6-bit-length addresses, the prefix 001* represents the addresses between 001000 and 001111 inclusive (in decimal form, between 8 and 15 inclusive). $[8, 16)$ is the *range* of the prefix 001*. 8 and 16 are the *left endpoint* and the *right endpoint* of the prefix 001*, respectively.

Definition 2. *If the ranges of two prefixes overlap with each other, then the two prefixes are* dependent*, otherwise they are* independent*. For a set of prefixes, if any pair of prefixes in it are independent, then the set is called an* independent set*.*

For example, in Fig. 1, any pair of prefixes P_1, P_2, $\cdots$, P_n are independent. Therefore, they form an independent set. Prefix P_{n+1} is independent of none of them.

Note that for any two prefixes, their ranges either contain one another or are disjoint. They cannot overlap with each other.

4.2 Partitioning

Given a set of prefixes, we try to partition out a set of independent sets from the prefix set. An algorithm to do this is provided in Fig. 2. It possesses the following inclusion property that for any two prefixes which are covered by one another, the longer prefix is chosen first.

Assume that all prefixes $P[1]$, $P[2]$, $\ldots$, $P[n]$ in the prefix set are sorted according to the left endpoint of prefixes as integers in increasing order. If two left endpoints are equal, the less specific prefix (i.e. with a range of a longer length) is put before the more specific one. For convenience, we introduce $P[0]$ as a prefix that contains all of $P[1]$, $\cdots$, $P[n]$, and $P[n+1]$ as a prefix that is not contained in $P[0]$. Let $P[i].set$ be the index for $P[i]$. All prefixes having the same index value are partitioned into the same independent set. Initially, $P[i].set = 1$ for all i. Assume we have a stack.

```
/*Begin Pseudocode*/
push $P[0]$;
for ($i = 1; i <= n + 1; i++$) {
        $j = 1$;
        while ($P[i]$ not contained in top){
                pop temp;
                temp.set = max{temp.set, $j$};
                $j++$;
        }
        top.set = max{top.set, $j$ };
        push $P[i]$;
}
/*End Pseudocode*/
```

Fig. 2. An algorithm for partitioning a prefix set.

The algorithm can be explained as follows. The prefixes P_1, P_2, $\cdots$, P_q define a nested sequence if P_i is contained in $P_{(i-1)}$, $q \geq i > 1$. The length of this nested sequence is q. It is easy to see that the $P[i]$s with the same $P[i].set$ value are independent and that I_j is the set of $P[i]$s with $P[i].set = j$.

Each $P[i]$ (other than $P[n+1]$) is pushed once and popped once. $P[n+1]$ is pushed but not popped. Not accounting for the sorting part, The total complexity is linear in the number of pops and pushes, i.e. $O(n)$.

Theoretically, an IPv4 forwarding table would be partitioned into at most 32 independent sets and an IPv6 forwarding table into at most 128 independent sets. However, in reality, the number of independent sets is much smaller. The following subsection provides experimental results on independent sets from using real forwarding tables.

4.3 Experimental Results

We downloaded IPv4 routing tables from [15] for the experiment. We have five forwarding tables from five different NAPs. We also combine these five tables and remove the duplicate to form a larger table. The results are shown in Tab. 1. The first row is the name of the NAPs. The second row is the sizes of the forwarding tables. The following rows show the sizes of the independent sets. We can see all the original tables have five independent sets. The combined table has six independent sets. For all the tables, the first independent set is much larger than the following independent sets.

Table 1. The size of the Independent Sets.

name	aads	mae-east	mae-west	pacbell	paix	combined
total	17486	18661	29609	24193	15332	50449
1st	16191	17635	27635	22728	14237	46597
2nd	1161	919	1759	1296	960	3305
3rd	124	94	195	148	118	469
4th	9	12	18	20	15	72
5th	1	1	1	1	2	5
6th	0	0	0	0	0	1

We expect the similar results for IPv6, i.e. the number of independent sets is much smaller than that in the worst case scenario (128). This indicates that our fast update algorithm is scalable to IPv6 and it is feasible for a hardware implementation.

5 Update Algorithm

In order to achieve fast update, the first step is to partition the prefix set into independent sets as described in the previous section. Then, parallel searches are performed on all the independent sets. When there are multiple matches, the first (largest) independent set has the highest priority, then the second and so on. Specifically, all the searches return a result, indicating either a successful search or an unsuccessful search. Among the successful searches, the result with the largest independent set is selected.

Let us take the aads forwarding table in Tab. 1 (the first column) as an example. The total of 17486 prefixes are partitioned into 5 independent sets. We could use 5 parallel searches. However, we can see that the first independent set is very large. The following independent sets become smaller and smaller. When the size of an independent set is very small, the worst-case update will not produce any significant effect on the update, because even recreating the data structure will not take many operations. For example, we can combine the 4th and the 5th independent sets into one set of 10 prefixes. Even in the worst case, only about tens operations are needed. According to the application requirement, the last three independent sets or the last four independent sets could be combined. More independent sets are combined, fewer parallel searches are needed, but updates become slower.

Let $I_1, I_2, \cdots, I_t$ be the partitioned sets obtained by the procedure described before. $I_1, I_2, \cdots, I_{t-1}$ are independent sets. I_t may not be an independent set. However, the number of prefixes in I_t is so small that even recreating a data structure of I_t will not take many steps in the case that I_t is not an independent set.

We provide an update algorithm, which involves both adding a prefix and deleting a prefix. If only deleting or adding a prefix is involved, the update algorithm will be much simpler than the one described in the following subsections.

5.1 Deleting a Prefix

Let P_1 be the prefix needs to be deleted. A simple-mind solution would be such that: Find the prefix first, and then delete it. This could cause problems when we add a prefix. Consider the case in Fig. 3. Assume P_1 has been deleted and P_4 needs to be added. I_k will still be independent if P_4 is added into I_k. However, this will result in a structure, which does not preserve the inclusion property on independent sets obtained from the algorithm given above since P_4 covers P_2 (as a prefix, P_2 is longer than P_4). To preserve the inclusion property, P_2 should have been moved to I_k after deleting and P_4 to I_{k+1}.

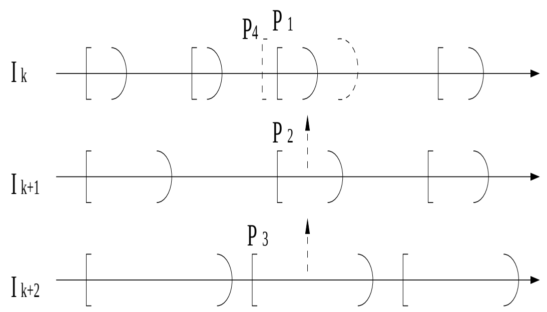

Fig. 3. A scenario for the need of moving prefixes after deleting.

An algorithm for deleting a prefix and preserving the inclusion property is given in Fig. 4. The algorithm stops when there is no prefix in the next independent set that covers the previous prefix and is independent of all other prefixes in the set to which the previous prefix belongs. The property given after the algorithm guarantees that the algorithm produces independent sets with the same structure property as that possessed by the original independent sets.

```
Step one: Let $P_1$ be the prefix to be deleted. Find $P_1$ from $\{I_1, I_2, \cdots, I_t\}$. Assume $P_1$
is in $I_k$. Delete $P_1$ from $I_k$.
Step two: Move prefixes between independent sets. Let $P_s$ and $P_b$ be variables for pre-
fixes.
$P_s$=$P_1$;
i=1;
$P_b$= a prefix in $I_{k+i}$ that covers $P_s$ and is independent of all other prefixes in $I_{k+i-1}$;
while($P_b$ is not empty) {
        move $P_b$ to $I_{k+i-1}$;
        i++;
        $P_s$=$P_b$;
        $P_b$= a prefix in $I_{k+i}$ that covers $P_s$ and is
        independent of all other prefixes in $I_{k+i-1}$;
}
```

Fig. 4. Algorithm for deleting a prefix.

Property 1. Let $P_1 = [b_1, e_1)$ be in I_k. Assume that there is no prefix P in I_{k+1} such that P covers P_1 and is independent of all other prefixes in I_k. Then, there is no prefix P in any of $I_{k+2}, I_{k+3}, \cdots, I_t$ such that P covers P_1 and is independent of all other prefixes in I_k.

Proof. We prove it by contradiction.

Assume there is a prefix $P_2 = [b_2, e_2)$ in I_{k+2} such that P_2 covers P_1 and is independent of all other prefixes in I_k. With the help of Fig. 5, we can see that there must be a prefix P in the interval of $[b_2, e_2)$ in I_{k+1}, otherwise, P_2 should have been moved to I_{k+1}. The prefix P cannot be in the interval of $[b_1, e_1)$, otherwise, P should have been moved to I_k since P_1 covers P. Thus, P should be in either $[b_2, b_1)$ or $[e_1, e_2)$. However, in either case, P should have been moved to I_k since P_2 is independent of all other prefixes in I_k and P should be independent of all other prefixes in I_k too. This indicates a contradiction. □

5.2 Adding a Prefix

For adding a prefix P, we first check I_1 to see if P is independent of I_1. If yes, then prefix P will be inserted in I_1 and the adding process ends; otherwise, there must exist a prefix in I_1, say P', which is dependent on P. There are only two cases: either P contains P' or vice versa. If P contains P', then proceed to check the independence of P and I_2; otherwise remove P' from I_1, insert P in I_1 in the

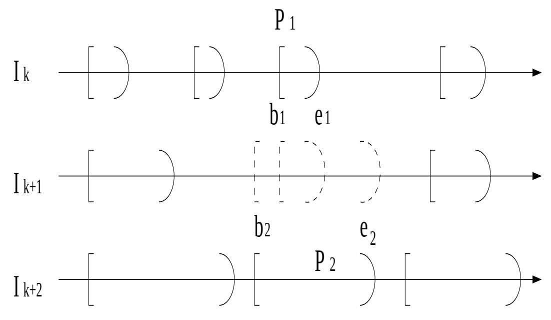

Fig. 5. Proof of the property.

place where $P^{'}$ was (we do not need to search from scratch), and then proceed to check the independence of the prefix $P^{'}$ and I_2. This procedure continues until the prefix is inserted in a set or all the sets are exhausted. In the latter case, the single prefix forms a new set.

5.3 Example for Adapting a Static Algorithm

We use the algorithm in [12] as an example to demonstrate how to adapt a static algorithm into a dynamic algorithm by combining the algorithm in this paper.

Given an IP address lookup table, we partition it into a set of independent sets. For each independent set, we adapt the algorithm in [12] to create a data structure for searching and updating. We first provide a brief review on the algorithm in [12].

Given a set of points, each of which is associated with a port number, we use these points to create a tree structure. Nodes in the tree have various children. The length of the paths from the root to all the leaf nodes are the same.

The following adaptations are applied for a dynamic algorithm. Each independent set is mapped to a set of endpoints of prefixes. Each left endpoint is associated with the port number of its corresponding prefix. Each right endpoint is associated with a number representing an empty port. The set of endpoints is used to create a tree as in [12]. (The variant two algorithm in [12] is recommended to use, because the endpoints in the high levels are less affected by an update.) When the tree is put in memory, it looks like the left part in Fig. 6.

We have at least two options to adapt the tree structure to facilitate update. First, we may give a headroom to each node for adding a prefix (at most two endpoints). The second option is to split the nodes from a leaf to the root when adding a prefix. In fact, a better choice is to combine these two option: We give headroom to root node and the level of nodes which are close to the root; we split leaf nodes and the level of nodes which are close to leaves. This makes good balance between memory efficiency and update speed. Merging nodes operation may be involved for deleting a prefix. For either adding or deleting a prefix, only the nodes from a leaf to the root are involved. Therefore, the update is very fast.

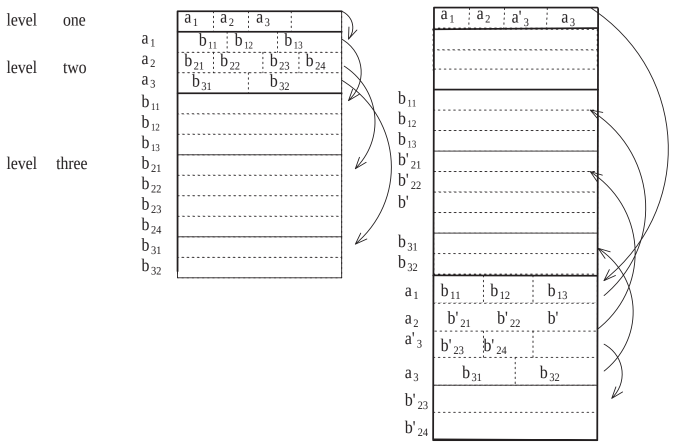

Fig. 6. Memory update.

Figure 6 gives one scenario for memory update. Level one node has headroom for endpoints. Level two and three are tightly stored in memory. A new endpoint is to be added between b_{22} and b_{23}. The endpoints in the node are re-constructed and have to be stored in two nodes: $\{ b_{21}^{'}, b_{22}^{'}, b^{'} \}$ and $\{ b_{23}^{'}, b_{24}^{'} \}$. Because of the spliting of the node, The one block in level three and one block in level two need to be moved to the end of the memory. We notice that a new endpoint is added to the level one node. Since we have given headroom in the node of this level, the new endpoint can be added without causing node splitting. The right part of Fig. 6 is the update result. We can see only two blocks of memory are needed to swap in.

The memory consumption of the headroom is not large. For example, we consider a tree with five levels and with an average degree of ten for each node. Level one to level three are given 100% headroom and there are no headroom in level four or level five. This only increases the memory by about 1%.

As we may notice, the memory may grow and data structure may become non-optimal with update. According to a particular application, after a period of time, we need to recreate the whole data structure and swap it in to achieve memory efficiency. However, this is typical for a dynamic algorithm.

The complexity is estimated in the following.

5.4 Estimation on Complexity

Assume that a search takes about s memory accesses. For deleting a prefix, about ts memory accesses are needed for step one. Fewer than $2ts$ memory accesses are needed for step two. Thus, a maximum of $3ts$ memory accesses is needed for deleting a prefix. If $s = 5, t = 4$, fewer than 60 memory accesses are needed. Only two or three blocks of memory need to be swapped in or updated in deleting a prefix. For adding a prefix, a similar result can be obtained. In comparison, the algorithm in [12] needs to recreate the data structure for each update. For example, if we need to store a forwarding table with 1M entries. Using the algorithm in [12], the whole bank of memory needs to be dumped with the updated data structure. It may take about one second for an update. Using the update algorithm in this paper, 1M updates per second update rate can be achieved. The time used for updating is less than 5% of the lookup time.

6 Previous Work

Reference [8][3] proposed algorithms with $O(\log n)$ complexity for both update and search. They are the first to achieve this performance for both update and search. Reference [14] also proposesd a fast update for multiway range trees. These algorithms are restricted to particular data structures. Our algorithm is much more general. It does not restrict us to any particular data structure or algorithm.

The algorithm in [10] supports an incremental update. Again, the supporting update is a characteristic of their search algorithm rather than a new algorithm.

For IP address lookup algorithms, readers are referred to the survey papers [5][11][9] and the references wherein. By carefully studying these algorithms, we can convert some of them into a dynamic algorithm with fast update using the scheme proposed in this paper.

7 Concluding Remarks

We have developed a novel update algorithm based on independent sets. In fact, it can be viewed as a general approach to potentially turn a static algorithm into a dynamic one with high speed updates. This can be done by introducing a parallel mechanism in the algorithm.

When a particular algorithm is combined with our update algorithm, some modifications cannot be avoided. As is the case for any dynamic algorithm, memory management needs to be elaborated to achieve memory efficiency. Since our proposed update algorithm does not incur any resource penalty, such as a large memory tradeoff, it provides a good alternative for the practical design.

Especially, we provide a case study for modifying the static algorithm in [12] into a dynamic one. Preliminary analysis shows that the resulting dynamic algorithm can achieve 1M updates per second update rate. The time used for updating is less than 5% of the lookup time. In comparison, the algorithm in [12] may need about one second for an update.

8 Acknowledgment

The authors would like to thank Professor Brett Stevens of Carleton University for making valuable comments in improving the quality of this paper.

References

[1] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *RFC 2460*, December 1998.

[2] R. Hinden and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture," *RFC 3513*, April 2003.

[3] H. Lu and S. Sahni, "O(log n) Dynamic Router-Tables for Ranges," *IEEE Symposium on Computers and Communications*, 2003, pp. 91-96.

[4] A. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," *IEEE INFOCOM 1993*, vol. 3, March 1993, pp. 1382-1391.

[5] M. A. Ruiz-Sanchez, E.W. Biersack and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network* 15, 2 March/April, 2001, pp. 8-23.

[6] R. Ramaswami and K.N. Sivarajan, "*Optical Networks: A Practical Perspective*," Morgan Kaufmann, San Francisco, CA, 1998.

[7] Y. Rekhter and T. Li, "An Architecture for IP Address Allocation with CIDR," *RFC 1518*, September 1993.

[8] S. Sahni and K. Kim, "O(log n) Dynamic Packet Routing," *IEEE Symposium on Computers and Communications*, 2002, pp. 443-448.

[9] S. Sahni, K. Kim and H. Lu, "Data Structures for One-dimensional Packet Classification Using Most-specific-rule Matching," *International Journal on Foundations of Computer Science*, 14, 3, 2003, pp. 337-358.

[10] Kari Seppänen, "Novel IP Address Lookup Algorithm for Inexpensive Hardware Implementation," *WSEAS Transactions on Communications*, 2002, Vol. 1, No. 1, pp. 76-84.

[11] X. Sun, "IP Address Lookups and Packet Classification: A Tutorial and Review," Technical Report #380, LRSP, Carleton University, 2002.

[12] X. Sun and Y. Zhao, "An On-Chip IP Address Lookup Algorithm," submitted to *IEEE Transactions on Computers*, 2004. (patent pending)

[13] X. Sun and Y. Zhao, "Packet Classification Using Independent Sets," *IEEE Symposium on Computers and Communications*, 2003.

[14] Subhash Suri, George Varghese and Priyank Ramesh Warkhede, "Multiway Range Trees: Scalable IP Lookups with Fast Updates," *Globecom*, 2001.

[15] http://www.merit.edu/ipma/routing_table/