Multiway Range Trees: Scalable IP Lookup with Fast Updates

Subhash Suri George Varghese Priyank Ramesh Warkhede

Department of Computer Science
Washington University
St. Louis, MO 63130.

Abstract— **In this paper, we introduce a new IP lookup scheme with worst-case search** *and update* **time of $O(\log n)$, where n is the number of prefixes in the forwarding table. Our scheme is based on a new data structure, a** *multiway range tree***. While existing lookup schemes are good for IPv4, they do not scale well in both lookup speed and update costs when addresses grow longer as in the IPv6 proposal. Thus our lookup scheme is the first lookup scheme to offer fast lookups and updates for IPv6 while remaining competitive for IPv4.**

The possibility of a global Internet with addressable appliances has motivated a proposal to move from the older Internet Ipv4 routing protocol with 32 bit addresses to a new protocol, IPv6 with 128 bit addresses. While there is uncertainty about IPv6, any future Internet protocol will require at least 64 bit, and possibly even 128 bit addresses as in IPv6. Thus for this paper we simply use IPv6 and 128 bits to refer to the next generation IP protocol and its address length.

Our paper deals with *address lookup*, a key bottleneck for Internet routers. Our paper describes the first IP lookup scheme that scales well as address lengths increase as in IPv6, while providing fast search times *and* fast update times.

There are four requirements for a good lookup scheme: search time, memory, scalability, and update time. Search time is important for lookup to not be a bottleneck. Schemes that are memory-efficient lead to faster search because compact data structures fit in fast but expensive Static RAM. Scalability in *both* number of prefixes and address length is important because prefix databases are growing, and a switch to IPv6 will increase address prefix lengths. Finally, schemes with fast update time are desirable because i) instabilities in backbone protocols [4] can cause rapid insertion and deletion of prefixes. ii) multicast forwarding support requires route entries to be added as part of the forwarding process and not by a separate routing protocol iii) Any lookup scheme that does not support fast incremental updates will require two copies of the lookup structure in fast memory, one for updates and one for lookup.

The main drawback of existing lookup schemes is their lack of *both* scalability and fast update. Current IP lookup schemes come in two categories: those that use precomputation, and those that do not. Schemes like "binary search on prefix lengths" [9], "Lulea compressed tries" [1], or "binary search on intervals" [5] perform precomputation to speed up search; however, adding a prefix can cause the data structure to be rebuilt. Thus, precomputation-based schemes have a worst-case update time of $\Omega(n)$. On the other hand, trie-based schemes (e.g., [8], [7]) do not use precomputation; however, their search time grows linearly with the prefix length.

A. Our Contribution

We introduce a new IP lookup scheme whose worst-case search *and update* time is $O(\log n)$, where n is the number of prefixes in the forwarding table. Our scheme is based on a new data structure, a *multiway range tree*, which achieves the optimal lookup time of binary search, but can also be updated fast when a prefix is added or deleted. (By contrast, ordinary binary search [5] is a precomputation-based scheme whose worst-case update time is $\Theta(n)$.)

Our main contribution is to modify binary search for prefixes to provide fast update times. In standard binary search [5], each prefix maps to an address range. A set of n prefixes partition the address line into at most $2n$ intervals. We build a tree, whose leaves correspond to interval endpoints. All packet headers mapped to an interval have the same longest prefix match. Search time is $O(\log n)$.

The problem is update: a prefix range can be split into $\Omega(n)$ intervals, which need to be updated when the prefix is added or deleted. Thus our first idea is associating an *address span* with every tree node v; the span of v is the maximal range of addresses into which the final search can fall after reaching v. Thus the root span is the entire address range. We then associate with every node v the set of prefixes that contain the span of v. However, this results in storing redundant prefixes.

To remove this redundancy, we store a prefix with a node v if and only if the same prefix is not stored at the parent w of the node. Since the span of v is a subrange of the span of w, any prefix that contains the span of w, will contain the span of v. Using this compression rule we can prove that only a small number of prefixes need be precomputed and stored; thus updates change information only at $O(\log n)$ nodes. By increasing the arity of the tree, we can reduce the height of the tree to $O(\log_d n)$, for any integer d. We pick d so that the entire node can fit in one cache line. For instance, a 256 bit cache line and arity-14 yields a tree height of about 5 or 6 for the largest IPv4 databases.

Our third contribution is to extend multiway range trees to IPv6. On a 32-bit machine, a single 128-bit address will require 4 memory accesses. Thus, an IP lookup scheme designed for IPv4 could slow down by a factor of 4 when used for IPv6. In practice, since we are assuming wide memories of at least 128 bits, we can handle such wide prefixes in one memory access. Unfortunately, the use of 128 bit addresses result in the use of a small branching factor and hence large tree heights. To avoid this problem, we generalize our scheme

to the case of k-word prefixes, giving a worst-case search and updates times of $O(k + \log_d n)$, where n is the total number of prefixes present in the database. Notice that the factor of k is additive.

The rest of this paper is organized as follows. Section I describes basic binary search. Section II describes our main data structure, the multiway range tree. Section III briefly describes how to handle updates efficiently. Section IV describes our extension to long addresses. Section V describes our experiments and measurements. Section VI states our conclusions.

I. IP Lookup by Binary or Multiway Search

We first review binary search [5] prefix matching. Suppose the database consists of m prefixes $r_1, r_2, \ldots, r_m$. Each prefix r matches a contiguous interval $[b, e]$ of addresses. For instance, the interval of prefix $r = 128*$ begins at point $128.0.0.0$ and ends at $128.255.255.255$. View the IP address space as a line in which each prefix maps to a contiguous interval, and each destination address is a point. Prefix intervals are disjoint but one prefix can completely contain another. The *longest matching prefix* problem reduces to determining the *shortest* interval containing a query point.

Let $p_1, p_2, \ldots, p_n$, where $n \leq 2m$, denote the distinct endpoints of all the prefix intervals, sorted in ascending order. With each key p_i, we store two prefixes, $match_=(p_i)$ and $match_<(p_i)$. The first, $match_=(p_i)$, is the longest prefix that begins or ends at p_i. The second, $match_<(p_i)$, is the longest prefix whose interval includes the range $(p_{i-1},\ p_i)$, where p_{i-1} is the predecessor key of p_i.

Given a destination address q, we perform binary search in $\log_2 n$ time to determine the *successor* of q, which is the *smallest key* greater than or equal to q. If $q = succ(q)$, we return $match_=(succ(q))$ as best matching prefix of q; otherwise, we return $match_<(succ(q))$. This search can implemented as a binary tree. To reduce tree height and memory accesses (assuming processing cost in registers is small compared to memory access cost), the binary search tree can be generalized to a multiway search tree; we use a B-tree.

In a B-tree, each node other than the root has at least $t-1$ and at most $2t-1$ keys, where t is known as the *arity* of the tree. The root has at least one key. A node with k keys, where $t-1 \leq k \leq 2t-1$, has exactly $k+1$ subtrees. Specifically, suppose a node v of the tree has keys $x_1, x_2, \ldots, x_k$. Then, there are $k+1$ subtree $T_1, T_2, \ldots, T_{k+1}$, where T_1 is the B-tree on keys less than or equal to x_1; T_2 is the B-tree on keys in the range $(x_1, x_2]$, and so on. To search for a query q at a node v, we find the smallest key x_i that is greater than or equal to q, and continue the search in the subtree T_i. For a tree of arity (degree) worst-case search takes $O(\log_t n)$ (wide) memory accesses.

II. Multiway Range Trees

The main difficulty with the binary search scheme [5] is that the addition of a short prefix can affect the $match_<()$ value of a large number of keys that it covers. For example, consider a database with a lone 8-bit prefix $128*$ together with 32K additional 32-bit prefixes described by the sequence $128.255.1.1, 128.255.1.3, 128.255.1.5 \ldots 128.255.255.255$.

Thus all IP addresses of the form $128.255.x.y$ where the LSB of $y = 0$ will have $128*$ as their best matching prefix. The binary search method [5] handles this by setting $match_<(p) = 128*$ for all the 32K prefixes of length 32. Now consider adding the prefix $128.255.*$ Since this new prefix is now a better match for prefixes of the form $128.255.x.y$, we have to update $match_<(p) = 128.255*$ for 32K values of p. Thus update can take linear time. We avoid this problem using the following ideas.

A. Address Spans

Our key idea is to associate *address spans* with nodes and keys of the tree; intuitively, the address span of a node is the range of addresses that can be reached through the node. For the multiway tree in Figure 1, first consider a leaf node, such as z. For each key x in it, we define the *span* of x to be the range from its predecessor key to the key itself. (When the predecessor key doesn't exist, we use the artificial guard key $-\infty$.) To ensure that spans are disjoint, each span includes the *right endpoint* of its range, but *not* the *left endpoint.* Thus, the spans of a, b and c are $span(a) = (-\infty, a]$, $span(b) = (a, b]$, $span(c) = (b, c]$.

The span of a *leaf node* is defined to be the union of the spans of all the keys in it. Thus, we have $span(z) = (-\infty, c]$. The span of a *non-leaf node* is defined as the union of the spans of all the descendants of the node. Thus, for instance, $span(v) = (-\infty, i]$, and $span(w) = (i, o]$, and $span(u) = (-\infty, o]$. (In fact, it is easy to check that nodes at the same level in the tree form a partition of the total address span defined by all the input prefixes.) We now describe how to associate prefixes with nodes of the tree.

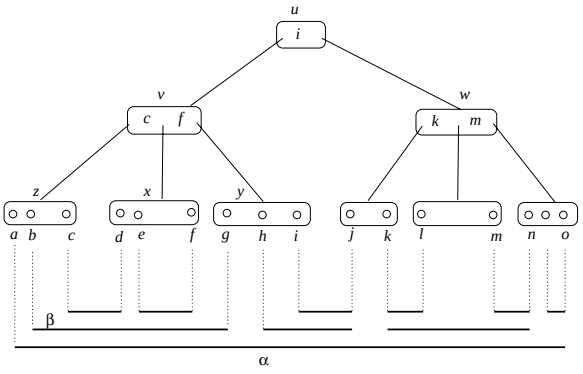

Fig. 1. Address spans.

B. Associating Prefixes with Nodes

Suppose we have a prefix r with range $[b_r, e_r]$. Clearly if for some node v of the tree, $span(v)$ is contained in the range $[b_r, e_r]$, then r is a matching prefix for any address in $span(v)$. We could store r at *every* node v whose span is contained in range $[b_r, e_r]$, but this leads to a potential mem-

ory blowup: we can have n prefixes, each stored at about n nodes. Observe, however, that the span of a parent is the union of the spans of each of its children. Thus if a prefix contains the span of a parent node, then it must contain the span of the child. Thus we do not need to explicitly store the prefix at the child if it is already stored at the parent. Thus we use the following compression rule:

[**Prefix Storing Rule:**] *Store a prefix r at a node or leaf key v if and only if the span of v is contained in $[b_r, e_r]$ but the span of the parent of v is not contained in $[b_r, e_r]$. (The parent of a key in a leaf node L is considered to be L.) In addition, r is stored with the start key of its range, namely, b_r.*

The first rule should be intuitive; the second rule (storing prefixes with the start key) is slightly more technical and is necessary to handle the special case when a search encounters the start point of a range. We can show that the update memory, for all practical purposes, is essentially linear except for a logarithmic factor of $2tw$ (proof omitted for lack of space). With $t = 16$ and $w = 32$, this is an additive factor of around $33N$.

C. Search Algorithm

We summarize the search algorithm. The arity t of the range tree is a tunable parameter. Each node has at least $t-1$ and at most $2t-1$ keys. Let M_v be the (narrowest) prefix stored at each internal node v in the search structure, where M_v is the root of the corresponding heap at node v in the update structure. Let E_k be the prefix stored with each leaf key in the search structure corresponding to the root of the *equal list heap* in the update structure; let G_k be the prefix stored with each leaf key in the search structure corresponding to the root of the *span list heap*.

We initialize the best matching prefix to be null. Suppose the search key is q, and the root of the range tree is u. If M_u is non-empty and is longer than the current best matching prefix, then we update the best matching prefix to M_u. Let $x_1, x_2, \ldots, x_d$ be the keys stored at u, and let $T_1, T_2, \ldots, T_{d+1}$ denote the subtrees associated with these keys. We determine the successor x_i of q as the smallest key in $x_1, x_2, \ldots, x_d$ that is greater than or equal to q. We recursively continue the search in the subtree T_i.

As in the original static search, we initialize the successor to be ∞, and when we descend into the subtree T_i, we update the successor to be x_i. Suppose k is the successor key of q when the search terminates. We check to see if $k = q$. If the equality holds, we return E_k; otherwise, we return G_k.

III. Updates to the Range Tree

We show how to handle prefix additions and deletions. We describe at a high level the changes that occur in the range tree when a prefix $r = [b_r, e_r]$ is inserted or deleted. Conceptually, update can be divided into three phases (which can be combined), each of which takes at most $t \log_t n$ time.

1. [**Updating the Keys and Handling Splits and Merges.**] The keys representing the endpoints of the prefix range, namely, b_r and e_r, may need to be inserted or deleted. A key may be the endpoint of multiple prefix ranges, so we maintain a count of the number of prefixes terminating at each key. When a key is newly inserted, its count is initialized to one. Whenever a prefix is inserted or deleted, the counts of its endpoint keys are updated accordingly. When the count of a key decrements to zero, it is deleted from the range tree. Insertion and deletion can result in nodes being split and merged; it is easy to update the spans of the affected nodes using the spans of the parent nodes.

2. [**Updating the Address Spans and Prefix Heaps.**] When a new key is inserted, it may change the address span of some nodes in the range tree. Specifically, suppose we add a new key q. Then, any node u whose *rightmost key* is q has its span enlarged—previously, the span extended to the predecessor of q; now it extends to q. In addition, if s is the successor of q, then the span of every node that is an *ancestor of s but not an ancestor of q* shrinks—previously, the span terminated at the predecessor of q; now it terminates at q. Thus, altogether there can be $O(\log_t n)$ nodes whose address span need updating. Similarly, when a key is deleted, spans of a logarithmic number of nodes are affected.
Modification of address spans can cause changes in the prefix heaps stored at these nodes—when the address span increases, some of the prefixes stored at the node may need to be removed, and when the address span of a node shrinks, some of the prefixes stored at a node's children may move up to the node itself.

3. [**Inserting or Deleting the New Prefix.**] Finally, the prefix r is stored in the heaps of some nodes, as dictated by the Prefix Storing Rule. This phase of the update process adds or removes r from those heaps.

Lemma 1: Given an IP address, we can find its longest matching prefix in worst-case time $O(t \log_t n)$ using the range tree data structure. The number of memory accesses is $O(\log_t n)$ if each node of the tree fits in one cache line. The data structure requires $O(nt \log_t W)$ memory, and a prefix can be inserted or deleted in the worst-case time $O(t \log_t n)$.

IV. Multiword Addresses: IPv6 or Multicast

In this section, we describe an extension of our scheme to multiword prefixes. On a 32-bit machine, accessing a single 128-bit address will require 4 memory accesses. Thus, an IP lookup scheme designed for IPv4 addresses could suffer a slowdown by factor of 4 when used for IPv6.

We consider the general problem of handling k-word address prefixes, where each word is some fixed W bits long. We show that our multiway range tree scheme generalizes to the case of k-word prefixes, giving a worst-case search and updates times of $O(k + \log_d n)$, where n is the total number of prefixes. We describe the general construction. The top level tree T_0 is the range tree on the first words of the input set of prefixes, namely, S. Consider a key x in the tree T_0, and let $S(x)$ denote the set of prefixes whose first word equals x. We build a second range tree on the set $S(x)$, and have the $=$ branch of x in T_0 point to it. We will call this second tree $T(x)$. We also store with the key x the longest (best) prefix in T_0 matching x.

In general, the tree $T(x_1 \cdot x_2 \cdots x_{i-1})$ is a range tree on the

ith words of the prefixes whose first $i-1$ words are exactly $x_1, x_2, \cdots, x_{i-1}$. If x_i is a key in this subtree, then its left subtree contains keys smaller than x_i, the right subtree contains keys bigger than x_i, and the $=$ pointer points a range tree on the $(i+1)$st words of the set $S(x_1 \cdot x_2 \cdots x_{i-1} \cdot x_i)$. (This is the set of prefixes whose first i words are exactly $x_1, x_2, \cdots, x_i$.) The key x_i also stores the best prefix in the set $S(x_1 \cdot x_2 \cdots x_{i-1})$ that matches x_i.

For instance, suppose $W = 4$, and consider three prefixes $r_1 = 10*$, $r_2 = 1010.01*$, and $r_3 = 1010.110*$. Then, the search tree on the first word includes the prefix range $10*$, and points to 1010 and 1010. This example also illustrates that the keys in the search tree form a *multiset*, since many different prefixes may have a common first word. Thus, each of our range trees will allow duplicate keys. In addition to allowing duplicate keys, the multiword multiway search tree has one other significant difference from the single word tree. Consider a key x at some node in the range tree. The key x partitions the key space in three parts: $<, >$ and $=$. Perhaps surprisingly, we can prove that the obvious trick of removing duplicate keys in each range tree, results in a correct algorithm but with poor performance.

Given a packet destination address $(p_1 \cdot p_2 \cdots p_k)$, the search algorithm is as follows. Observe that each p_i is fully specified, so it maps to a point. We begin with the first word p_1, and find its successor key s in the top level tree T_0. During the search for s, we maintain the best matching prefix found along the path.

As soon as we find a key q such that $q = p_1$, we stop searching the tree T_0. If the best matching prefix stored at q is longer than the one found so far, we update the answer. We now extract the second word of the packet header, namely, p_2, and start the search in the range tree $T(p_1)$ pointed to by q. Search terminates either when the associated B-tree is empty, or when we don't get an equal match with the search key. In the former case, we simply output the best matching prefix found so far. In the latter case, suppose the search terminates at a successor key s such that $s > p_1$. In this case, we compare the current best matching prefix with the prefix stored at the root of the span list heap of s, and choose the longer prefix.

V. Experimental Results

This section describes the experimental setup and measurements for our scheme. We consider software platforms using modern processors such as the Pentium and the Alpha. For the Pentium processor, size of a cache line is 32 bytes (256 bits). For hardware platforms, we assume a chip model that can read data from internal or external SRAM. We assume that the chip can make wide accesses to the SRAM (using wide buses), retrieving upto 512 and even 1024 consecutive bits. We count memory accesses in terms of distinct cache-line accesses, as opposed to the number of memory words retrieved. Since lookup memory needs to be in fast SRAM while update memory need not, we separately state our lookup-relevant memory and update memory. We implemented our range tree scheme in C on a UNIX machine.

A. Results for IPv4 and IPv6

Experiments were conducted using (somewhat dated) routing table snapshots from IPMA[6]; only a subset of results are described here. Experiments involved inserting prefixes in the same order as the input database. Measurements of lookup time were obtained for randomly generated IP addresses.

Given an input set of n prefixes, the height of the range tree of maximum arity $2t$ is $h \leq \lceil \log_t 2n \rceil$. The number of memory accesses per lookup is h in the worst case. For insertion of a new prefix, the worst case number of memory accesses is $5ht+4(h+t+1)$, while for deletion of a prefix the worst case number of memory accesses is $5ht + 6h + 5t + 5$.

Table I summarizes observed tree height, memory requirement for the search structure and the overall memory requirement (*i.e.,* including memory relevant only for updates) for the largest available dataset, Mae-East.

Arity	Height	Search Memory	Total Memory
4	15	0.82MB	1.76MB
8	8	0.51MB	1.15MB
14	6	0.44MB	0.99MB
18	5	0.42MB	0.95MB

TABLE I
Mae-East database: 41,456 prefixes, 62273 distinct keys

Similar observations hold for the smaller PacBell dataset (Table II) for which the number of distinct keys is about a third of Mae-East (though it has half the number of prefixes of Mae-East). This is reflected in the memory requirement and height of the range tree, which are a function of number of keys, not prefixes.

Arity	Height	Search Memory	Total Memory
4	12	0.26MB	0.53MB
8	6	0.17MB	0.36MB
14	5	0.15MB	0.32MB
18	4	0.15MB	0.30MB

TABLE II
PacBell database: 24740 prefixes, 24380 distinct keys

An important purpose of the experiments was to understand variation of performance parameters (speed and memory requirement) with change in arity of the range tree. The results show that when the arity is small, say 4, the total memory requirement is quite high. This occurs because the range tree structure guarantees a minimum utilization of $\frac{t-1}{2t-1}$ for maximum arity $2t$. Thus, when the maximum arity is 4, many nodes have only one key, which leads to an increased number of allocated nodes and large total memory. Increasing arity of the tree from very small values achieves significant reduction in the number of nodes.

The worst-case update time for our scheme is calculated assuming node split/merge at each level of the tree. To evaluate the average update performance, we first built the prefix database corresponding to the Mae-East database. We then

generated 500 random prefixes, inserted them into the data structure, and then deleted them in random order. Table III below summarizes the results of this experiment.

Arity	Height	Average	Worst Case
4	12	107	337
8	6	58	321
14	5	45	455
18	4	38	479

TABLE III
Average update performance of multiway range trees.

For small arity values, say 4, the average update time is fairly close to the worst case. But with increasing arity, average update time becomes much smaller than worst case update time. This is expected, because the probability of a node merge/split is very high for lower arity values, and decreases linearly with increasing arity. For arity 14, which is used later on for comparison with other schemes, the average update time is less than 10% of the worst case.

IPv6 Performance: Since IPv6 prefix databases are unavailable, we only provide analytical bounds. Using 32 bit words, each address is $l = 4$ words long. Worst case lookup time is $l + \lceil \log_t 2n \rceil$ for trees of arity $2t$. Let h be the height of any tree. Then, $h \leq \lceil \log_t 2n \rceil$. Since we already know worst case update time for a single tree of height h, total update time for the data structure is $l * (5ht + 6h + 5t + 5)$ in the worst case. For 512 bit cache-line, we can choose an arity of 14. For a dataset of the same size as Mae-East ($41,456$ prefixes), we get a projected worst case lookup time of 9 memory accesses and an update time four times the IPv4 update time, 1800 memory accesses. The total memory requirement also changes for IPv6. We measured the increased overhead (due to $=$ tree pointers associated with each key) by prepending the string $0_0 \ldots 0_{95}$ to each prefix. The constructed data structure required almost double the amount of memory required for the corresponding IPv4 data structure.

B. Comparison with other schemes

Several schemes have been proposed in the literature for performing IP address lookups. We evaluate these schemes on four important metrics: lookup time, update time, memory requirement and scalability to longer prefixes. Table IV compares the complexity of search time, update time and memory requirement of the proposed scheme against some of the prominent schemes proposed in literature.

Scheme	Lookup	Update	Memory
Patricia trie	$O(W)$	$O(W)$	$O(nW)$
Multibit tries	$O(W/k)$	$O(\frac{W}{k}2^k)$	$O(nW2^k)$
Binary search	$O(\log_2 n)$	$O(n)$	$O(n)$
Multiway search	$O(\log_k n)$	$O(n)$	$O(n)$
Prefix length binary search	$O(\log W)$	$O(n)$	$O(n \log W)$
Multiway Range Trees	$O(\log_k n)$	$O(k \log_k n)$	$O(kn \log_k n)$

TABLE IV
Comparison of search, update and space complexities

As seen from the table, all existing schemes, except trie-based schemes, require $O(n)$ update time in the worst-case. Trie based schemes are slow for large address sizes; large strides can increase worst-case memory exponentially (most papers only report memory on typical databases.)

VI. Conclusion

We have described a new IP lookup scheme that scales well to IPv6 and multicast addresses, allowing fast search and update. It is the first scheme we know of that possesses all these properties.

For example, prior schemes such as [9], [1], [5] all have $\Omega(n)$ update times, where n is the number of prefixes, while multibit trie schemes (e.g., [8], [7], [3]) take $O(w)$ search times, where w is the address length. With the size of prefix tables approaching 100,000 and expected to perhaps reach 500,000 in a few years, $\Omega(n)$ update times can be problematic. On the other hand, multibit trie schemes require w/c memory accesses where c is at most 8 in order to bound memory. This can be expensive for 128-bit IPv6 addresses. Thus our scheme, which takes worst-case search *and update* times of $O(\log n)$, may be interesting for larg IPv6 tables, especially by choosing a sufficiently high arity to make the $\log n$ term very small.

Even for IPv4, unlike other schemes such as [9], [1], [8], our scheme is not patented and can be used freely for publically available software. Our scheme is competitive for IPv4 For example, on a Mae-East database using a 512 bit cache line, our scheme takes 5 memory accesses to do a worst-case lookup, requires 0.44 MB storage, and needs 455 memory accesses for update in the worst case. Even if its performance is not as good as multibit tries, its availability and simplicity could make it useful for a software solution used by low-end routers that sit on the edge of the network.

References

[1] Andrej Brodnik, Svante Carlsson, Mikael Degermark, and Stephen Pink. Small Forwarding Table for Fast Routing Lookups. *Computer Communication Review*, October 1997.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. *INFOCOM '98*, April 1998.

[4] C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *ACM SIGCOMM 97*.

[5] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multiway and Multicolumn Search. *IEEE INFOCOM '98*.

[6] Merit Inc. *IPMA Statistics*. http://nic.merit.edu/ipma.

[7] S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. *Proceedings of IEEE Broadband Communications 98*, April 1998.

[8] V. Srinivasan and George Varghese. Faster IP Lookups using Controlled Prefix Expansion. *ACM Sigmetrics'98*, June 1998.

[9] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. *ACM SIGCOMM 97*.