Report Number: WUCS-99-28

1999-01-01

Multiway Range Trees: Scalable IP Lookup with Fast Updates

Subhash Suri, George Varghese, and Piryank Ramesh Warkhede

Internet routers forward packets based on the destination address of a packet. A packet's address is matched against the destination prefixes stored in the router's forwarding table, and the packet is sent to the output interface determined by the longest matching prefix. While some existing schemes work well for IPv4 addresses, we believe that none of the current schemes scales well to IPv6, especially when fast updates are required. As the Internet evolves into a global communication medium, requiring multiple addresses per user, the switch to longer addresses (e.g. IPv6) seems inevitable despite temporary measures such as network addres... **Read complete abstract on page 2.**

Multiway Range Trees: Scalable IP Lookup with Fast Updates

Subhash Suri, George Varghese, and Piryank Ramesh Warkhede

Complete Abstract:

Internet routers forward packets based on the destination address of a packet. A packet's address is matched against the destination prefixes stored in the router's forwarding table, and the packet is sent to the output interface determined by the longest matching prefix. While some existing schemes work well for IPv4 addresses, we believe that none of the current schemes scales well to IPv6, especially when fast updates are required. As the Internet evolves into a global communication medium, requiring multiple addresses per user, the switch to longer addresses (e.g. IPv6) seems inevitable despite temporary measures such as network addres translation (NAT) boxes. Since IPv6 uses 128 bit addresses, schemes whose lookup time groqws with address length (such as patricia or multi-bit tries) become less attractive. Because of backbone protocol instabilities, it is also important that lookup schemes be able to accomodate fast updates. 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, which achieves the optimal lookup time of binary search, but can also be updated in logarithmic time when a prefix is added or deleted; by contrast, plain binary search relies on precomputation, and a single update can require O(n) time. Our performance analysis shows that, even for IPv4, multiway range trees are competitive with the best lookup schemes currently known. In fact, among existing schemes, only multibit tries have update performance comparable to our scheme. However, when considering IPv6 or any future routing protocol that uses longer addresses, our scheme outperforms all existing schemes, including multibit tries.

Multiway Range Trees: Scalable IP Lookup with Fast Updates

Subhash Suri, George Varghese and
Priyank Ramesh Warkhede

WUCS-99-28

November 1999

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

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

Internet routers forward packets based on the destination address of a packet. A packet's address is matched against the destination prefixes stored in the router's forwarding table, and the packet is sent to the output interface determined by the longest matching prefix. While some existing schemes work well for IPv4 addresses, we believe that none of the current schemes scales well to IPv6, especially when fast updates are required. As the Internet evolves into a global communication medium, requiring multiple addresses per user, the switch to longer addresses (e.g. IPv6) seems inevitable despite temporary measures such as network address translation (NAT) boxes. Since IPv6 uses 128 bit addresses, schemes whose lookup time grows with address length (such as patricia or multi-bit tries) become less attractive. Because of backbone protocol instabilities, it is also important that lookup schemes be able to accommodate fast updates.

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*, which achieves the optimal lookup time of binary search, but can also be updated in logarithmic time when a prefix is added or deleted; by contrast, plain binary search relies on precomputation, and a single update can require $O(n)$ time. Our performance analysis shows that, even for IPv4, multiway range trees are competitive with the best lookup schemes currently known. In fact, among existing schemes, only multibit tries have update performance comparable to our scheme. However, when considering IPv6 or any future routing protocol that uses longer addresses, our scheme outperforms all existing schemes, including multibit tries.

1 Introduction

The number of hosts as well as the amount of traffic on the Internet is exponentially increasing. The possibility of a global Internet with addressable appliances has motivated a proposal for a transition from the older Internet routing protocol, IPv4 with 32 bit addresses, to a next generation protocol, IPv6 with 128 bit addresses. Because of the difficulties associated with such a transition, a number of temporary measures, such as NAT boxes (where several hosts share a common IP address) are being deployed to use addresses more frugally. In the long term, however, there is a clear need for longer addresses. While there is some uncertainty about whether IPv6 will survive in its present form, any future Internet protocol will certainly require longer addresses, requiring at least 64 bit and possibly even 128 bit addresses as in IPv6. We do not dwell on this uncertainty in the rest of this paper, but simply use IPv6 and 128 bits to refer to the next generation IP protocol and its address length.

Communication links in the Internet backbone are also being upgraded to handle the increased traffic: OC-192 (10 Gbps) links are scheduled for the near future. Internet routers operating at Gigabit rates need to forward several million packets per second to keep up with such links. Our paper deals with one component of high speed forwarding: *address lookup*, which is one of the bottlenecks faced by Internet routers.

When an Internet router gets a packet from an input link interface, it uses the destination address in the packet' header to lookup a routing database. The result of the lookup provides an output link interface, to which packet is forwarded. Routers aggregate forwarding information by storing address *prefixes* that represent a group of addresses reachable through the same interface. IPv4 prefixes are arbitrary bit strings up to 32 bits in length; in IPv6, the addresses will be strings of length up to 128 bits. If a packet matches multiple prefixes, then it is forwarded using the *longest* matching prefix.

There are four key requirements for a good lookup scheme: search time, memory, scalability, and update time. We discuss each of these requirements in turn. Search time is clearly important for lookup to not be a bottleneck. Schemes that are memory-efficient can also lead to good search times because compact data structures can fit in fast but expensive Static RAM memory. The fastest lookup implementations may require on-chip SRAM (2-4 nsec access time) which is currently limited to around 16 Mbits. Thus reducing memory needs makes it feasible to support larger prefix databases on-chip. Alternately, an implementation may use external SRAM, in which case reduced memory will translate to lower implementation cost.

Scalability in terms of *both* number of prefixes and address length is important because prefix databases are growing, and a switch to IPv6 will significantly increase the address prefix lengths. Several existing schemes work well for IPv4, but do not scale well to longer addresses as their search speed is $O(W)$, where W is the maximum address length.

Besides the longer addresses in IPv6, multicasting in IPv4 also potentially uses longer prefixes. For instance, forwarding a multicast packet using DVMRP requires the insertion of a (G, S) forwarding entry when the first packet to group G from source S is sent. While G is almost always[1] a full 32 bit address, S is a prefix. Thus a simple way to handle multicast is to concatenate G and S to form a 64 bit prefix, which then requires a 64 bit prefix lookup. Many router vendors use this technique to handle multicast forwarding. Alternative schemes do exist that first do a hash on G which then points to a trie on the S field, but they require additional mechanisms just for multicast forwarding. Many hardware designers also prefer not to deal with non-deterministic (due to hash collisions) lookup times.

Finally, schemes with fast update time are desirable because instabilities in the backbone protocols [6] can lead to rapid insertion and deletion of prefixes. While some instability has been traced to flawed implementations and some problems have been ameliorated using route

[1]There are proposals to make the G field a prefix, which would then make the (G, S) lookup a more complicated two dimensional filter matching problem. However, this is still not common.

dampening timers, it seems important to allow routing protocols to add or prefixes rapidly to cope with failures and changes in traffic patterns. Another reason for fast update is to support multicast forwarding, where the route entry is added as part of the forwarding process and not by a separate routing protocol. A third and subtler reason is to reduce memory needs. Any lookup scheme that does not support fast incremental updates will require two copies of the lookup structure in fast memory; one that is being updated and one that is being used for lookup. *By contrast, schemes like ours that support fast incremental updates, however, can allow updates to atomically update the database while lookups continue on the same copy.*

The main drawback of the existing lookup schemes is their lack of scalability and fast update. Current IP lookup schemes can be broadly classified in two categories: those that use precomputation, and those that do not use precomputation. Schemes like "binary search on prefix lengths" [17], "Lulea compressed tries" [1], or "binary search on intervals" [7] perform a lot of precomputation to speed up the search for the best matching prefix. The down side of this precomputation (see detailed analysis later) is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. Thus, broadly speaking, precomputation-based schemes have an update time of $\Omega(n)$ in the worst-case. On the other hand, schemes based on a Patricia trie or a multibit trie [15] do not use precomputation; however, their search time grows linearly with the prefix length, and thus these schemes do not scale well to longer IP addresses. (Please note that reducing the height of the multibit trie *exponentially* increases the memory requirement.)

1.1 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, *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. (We note that ordinary binary search [7] is a precomputation-based scheme whose worst-case update time is $\Theta(n)$.)

The $O(\log n)$ bound ensures that our scheme scales well to large databases as well as long addresses. The main idea behind our scheme is to *dynamize* static binary search for prefixes. Each prefix maps to a range in the address domain $[0,\ 2^W - 1]$. A set of n prefixes partition the address line into at most $2n$ intervals. We build a tree, whose leaves correspond to the endpoints of these intervals. The key observation is that all packet headers mapped to an interval have the same longest matching prefix. The height of the tree is $O(\log n)$, which bounds the search time. So far, this is a fairly standard application of a binary search tree.

The problem is the update: a prefix range can be split into $\Omega(n)$ intervals, which need to be updated when the prefix is added or deleted. So, we introduce our first major idea: associate an *address span* with every node in the search tree. Intuitively, the address span of a node v is the maximal range of addresses into which the final search must fall after reaching v in the search; clearly the address span of the root is the entire range of addresses. Once this is done, we associate with every node the set of prefixes that contain the span of node v. Unfortunately, this can result in storing redundant prefixes with each node.

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. This makes sense because the span of v is a subrange of the span of w. Thus if any prefix contains the span of w, it will contain the span of v; thus the storage at v is redundant. 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. Further, the prefix storage rule and the definition of a node span clearly guarantees that all prefixes that match an address D will be encountered in some node on the search path for D. In practice, since we are only interested in the longest match, the search structure needs only store the longest such prefix at each node. The update structure, however, needs to store a heap of all such prefixes at each node to accommodate prefix deletions. Fortunately, even the size of the update structure can be shown to have minimal worst case storage.

By increasing the arity of the tree, we can reduce the height of the tree to $O(\log_d n)$, for any integer d. By picking an appropriate d so that the entire node can fit in one cache line, we can minimize the number of memory accesses. For instance, assuming a 256 bit cache line, we can make the arity of our tree to be 14, which yields a tree height of about 5 or 6 for the largest IPv4 databases.

Our last contribution is to extend the multiway range trees to IPv6. 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. 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 will only result in the use of a small branching factor and hence large tree heights (even using 1 key and 1 pointer will fill a 256 bit cache line, necessitating a branching factor of only 2). Thus the real reason we wish to use small word sizes so that we can have a large branching factor and hence a small tree height.

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 present in the database. Notice that the factor of k is additive which is an attractive scaling property.

1.2 Previous Work

Ternary CAMs (with bitwise maskable entries) have been suggested as a competitive solution for IP lookup. Current CAMs are limited to 8000 prefixes, which is too small for the largest databases. Larger CAMs able to fit 32,000 prefixes are likely to be available in near future. Many backbone router vendors, however, do not consider CAMs to be viable. First, projecting for growth, multicast addresses and host routes, many vendors envision databases with 150,000 or more prefixes, for which CAMs are too small. Second, unlike algorithmic solutions, the remainder of the forwarding tasks (updating hop counts etc.) cannot be integrated in a single forwarding chip. Third, CAM technology does not scale in density as well as SRAM technology; thus as SRAMs get denser and faster, algorithmic solutions based on SRAMs should beat solutions based on CAMs.

Caching [3] is a standard solution for improving average performance, however, experimental studies have shown poor cache hit rates for backbone routers [12]. Schemes like Tag and Flow Switching suggest protocol changes to avoid the lookup problem altogether. These proposals depend on widespread acceptance, and still require lookups at network boundaries.

In the last few years, several algorithmic techniques for best matching prefix have been proposed. The method of [1] is based on compressing trie nodes so that they will fit into the cache. The approach in [17] is based on doing binary search on the *possible prefix lengths*. Another form of compressed tries, *LC tries* is presented in [11]. All these schemes have fairly good search times for IPv4, but they all suffer from $\Omega(n)$ update time in the worst-case. The search time of the trie-based schemes also grows linearly with the prefix lengths, so they are not likely to scale well to IPv6.

In [7], a scheme based on binary search is presented that finds the best matching prefix in $O(\log n)$ time. This scales well to longer addresses, but requires precomputation that makes the update cost $\Omega(n)$ in the worst-case. Among the existing schemes, the best update cost is achieved by multibit tries [15]. Unfortunately, the search times of multibit tries do not scale well to IPv6. We do an exhaustive and quantitative comparison of our new scheme with all existing schemes in Section Section 6.

1.3 Paper Organization:

The rest of this paper is organized as follows. Section 2 describes the basic binary search. Section 3 describes our main data structure, the multiway range tree. Section 4 briefly describes how to handle updates efficiently. Section 5 describes our extension to long addresses. Section 6 describes our experiments and measurements. Section 7 states our conclusions.

2 IP Lookup by Binary or Multiway Search

We first review the use of binary or multiway search for computing best matching prefix in a *static* (non-changing) prefix database. Suppose the prefix 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 point 128.255.255.255. One can view the problem of computing longest matching prefix geometrically as follows: the IP address space is a line in which each prefix maps to a contiguous interval. A fully specified address (namely, the query) maps to a point. The prefix property guarantees that two intervals are either disjoint or one completely contains the other. The fewer the number of bits specified in a prefix, the longer is the corresponding interval. Thus, the *longest matching prefix* problem corresponds to the problem of 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. (For convenience, we will assume that there is a default prefix $*$, covering the whole interval $[0, 2^{32} - 1]$.) Assume that the sequence $(p_1, p_2, \ldots, p_n)$ is 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 query 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 be modeled by a binary tree, whose internal nodes correspond to the key comparisons. Figure 1 shows an example.

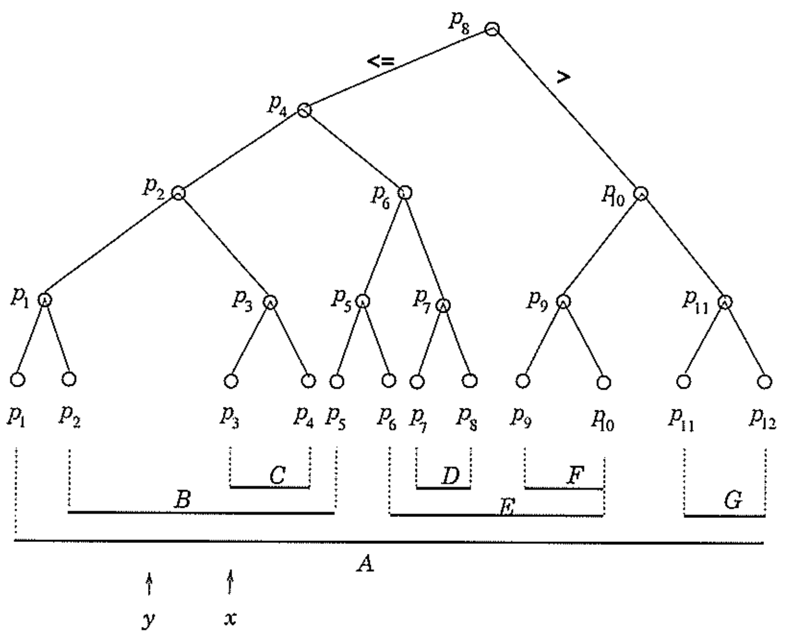
p_8
<=
>
p_4
p_2
p_6
p_{10}
p_1
p_3
p_5
p_7
p_9
p_{11}
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12}
C
B
D
E
F
G
A
y
x

Figure 1: The binary search tree. The search for x terminates at p_3, which is the smallest key satisfying $x \leq p_3$. In this case, since $x = p_3$, the returned prefix is C. The search for y also terminates at p_3, and in that case the returned prefix is B.

To reduce the height, the binary search tree described above can be readily generalized to a multiway search tree; we choose to 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. 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. Figure 2 shows an example.

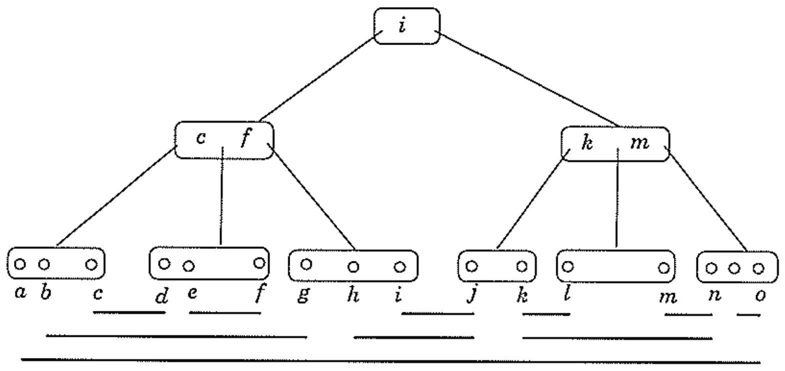

Figure 2: A multiway tree.

Clearly, multiway search is attractive when an entire tree node can be accessed in one wide memory access. While finding the specific key within a node can take $\log_2 t$ processing time, this can be ignored compared to the time for memory accesses which dominates search times. Since a tree of arity (degree) t has depth at most $1 + \log_t n$, the worst-case search takes $O(\log_t n)$ memory accesses.

2.1 Difficulty of Updating Search Trees

While the standard binary and multiway trees allow fast lookup times, the worst case cost of updates can be large. The problem (described in [7]) arises in maintaining the precomputed prefixes $match_=(p)$ and $match_<(p)$ as new prefixes are added or old ones deleted. Figure 3 shows a worst-case example where each update can require $\Theta(n)$ work. Initially the database contains the prefixes $A, B, \ldots, H$. Now we add prefix Y. This has the effect of changing $match_<()$ for the left endpoints of prefixes $A, B, \ldots, H$. Next, if we add prefix X, then the $match_<()$ field for the left endpoints of prefixes $B, \ldots, H$ changes. In this manner, we can create a series of prefixes each of which requires $\Theta(n)$ updates of $match_<()$.

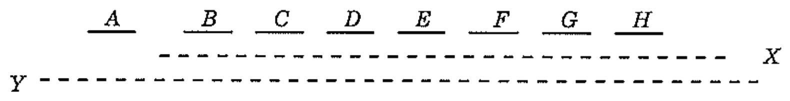

Figure 3: Binary search tree can require $\Theta(n)$ updates for each addition or deletion.

3 Multiway Range Trees

The main difficulty with the binary search scheme of the preceding section [7] is that a prefix can affect the $match_<()$ value of a large number of keys. Intuitively, a short prefix (with long interval) can cover many keys, setting their $match_<()$ values. In the search trees, prefixes are stored only at the leaf nodes; the internal nodes only store keys used for branching. Our main idea is to store prefixes at all nodes of the search tree, such that the following two conditions are met: (1) every prefix matching a key is stored at some nodes along the root to leaf path for that key, and (2) no prefix is stored at more than $O(\log n)$ nodes. The first condition permits logarithmic time lookup, while the second condition makes the update cost $O(\log n)$.

3.1 Address Spans

A key idea is to associate *address spans* with both the nodes and keys of the search tree. Intuitively, the address span of a node is the range of addresses that can be reached through the node, which in turn defines the set of prefixes that can be matched by addresses whose search path encounters this node.

Consider the multiway tree in Figure 4. We first consider a leaf node, such as the one labeled 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.) Note that this corresponds to the intuitive notion of a span of node w defining the range of addresses that can be reached after the multiway search reaches node w. We now describe how to associate prefixes with nodes of the tree.

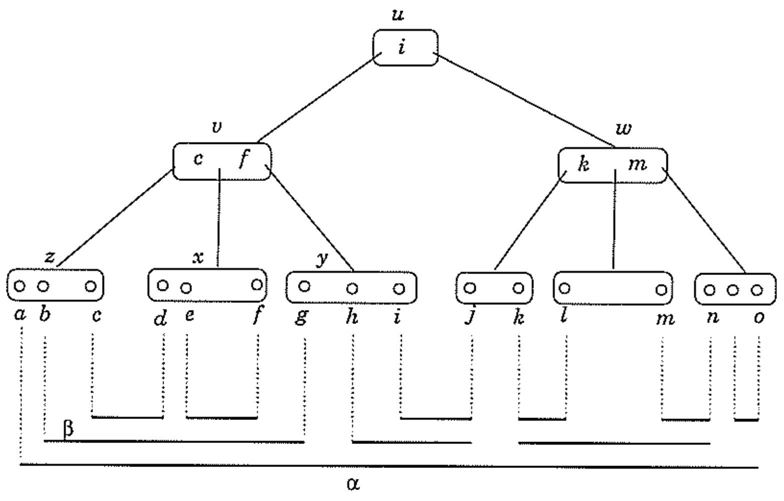

Figure 4: Address spans.

3.2 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 memory blowup: we can have n prefixes, each stored at about n nodes. Thus, we need to be more conservative about storing prefixes, yet need to ensure that every prefix matching an address is stored along the search path for that address.

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 because we must encounter the parent in any search that reaches the child. This motivates the following rule for storing prefixes that solves the prefix storage problem:

> [**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 intuitively clear; 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. (Note, however, that under the second rule a prefix is stored only at leaf nodes, and never at internal nodes.) Consider the example shown in Figure 4. By the first part of Prefix Storing Rule, the prefix α will be stored at keys b, c and nodes x, y, w. The span of w is contained in the range of α, and so we need not store α with the children of w. Similarly, the spans of x and y are contained in α, and therefore α is not stored with any of the leaves under nodes x or y. However, the spans of nodes z, u, or v are not contained in the range of α, and to α is not stored with those nodes. Finally, by the second part of the rule, α is also stored at the leaf a, because a is the left endpoint of α's range.

For the purpose of updates, we partition the prefixes stored at each key into two lists, one for the prefixes starting at that key, and the other for the remaining prefixes. For ease of reference, let us call them the *equal list* and the *span list*. However, for purposes of search, we only need to store the head of each list, a direct correspondence with the two best matching prefixes precomputed in the original binary search, namely, $match_{=}(k)$ and $match_{<}(k)$. Suppose key k is the successor of a query address q. If $k = q$, then we return the longest prefix in the equal list of k; otherwise we return the longest prefix in the span list of k.

Except for the keys, all other nodes in the range tree have a single span list of prefixes, which are maintained in a heap, ordered by prefix length. In a later optimization, we will show how we can avoid maintaining two lists per key by modifying the search keys slightly. It is crucial to observe, however, that the heaps are required only for update.

The following lemma follows easily from the definition of node spans and the compression rule. Intuitively, if a search key k has a successor q and prefix r matches q, then r contains the span of k; thus by the compression rule, either r is stored at k or some ancestor of k.

Lemma 3.1 *Suppose a prefix r matches an address key q. Let k be the successor key of k, namely, the smallest key satisfying $q \leq k$. Then, r is stored at some node along the range tree path from the root to the leaf containing k.*

Clearly, a prefix can be stored at multiple nodes in the update structure, but at how many nodes? *Note that the search memory requirement is the standard memory required for a tree together with a single prefix per node.* We now analyze this problem to show that the update memory, for all practical purposes, is essentially linear except for a logarithmic factor of $2tw$. With $t = 16$ and $w = 32$, this is additional factor of around $33N$. Fortunately, we can almost completely remove the effect of prefix storage even for updates by using bit maps to replace prefix lists.

Lemma 3.2 *In a range tree of arity t, the total number of prefixes stored with all the nodes is at most $4nt \log_t 2tW + n$.*

PROOF. In our range tree, all leaves are at the same depth. Let us call a depth d *cutoff depth* if every node at that depth has a subtree of size at least W and at most $2tW$. Call a node at depth d a *cutoff node*. Thus, there are at most n/W cutoff nodes. First, if v is a cutoff node, then all the ancestors of v (including v itself) have at most W prefixes stored with them because each of these must be a prefix of any prefixes stored at the leaf nodes, and there can be at most W prefixes of a given prefix. Thus, the total number of prefixes stored at and above cutoff nodes is $W \times \frac{n}{W} = n$.

We now bound the number of prefixes stored below the cutoff nodes. Consider a cutoff node v, and let T_v denote the subtree rooted at v. By the definition of a cutoff node, this subtree has size at most $2tW$. If a prefix $r = [b_r, e_r]$ is stored at a node in T_v, then either b_r or e_r (or both) must belong to a leaf in T_v—if both endpoints of r lie outside T_v, then $span(v) \subseteq [b_r, e_r]$, and so r will be stored at v or higher. Thus, at most $2tW$ prefixes qualify to be stored at the nodes of T_v. Since each of these prefixes is stored at no more than $\log_t(2tW)$ nodes, the total number of prefixes stored below v is $2tW \log_t 2tW$. Since the total number of cutoff nodes is n/W, the total memory bound is $\frac{n}{W} \times 2tW \log_t W + n = 4nt \log_t W + n$. □

3.3 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.

Note that this requires storing two prefixes with every leaf key in the search structure; a simple optimization is to increase the range to place range endpoints slightly before the actual prefix endpoints; in this case, the equality test never occurs and we only need to store the equivalent of G_k (a single prefix) with each key, and a single prefix with each internal node. The size of the memory access to read a leaf node can be reduced further by having the associated prefixes of a leaf node be retrieved by one extra memory access. The extra prefix stored at an internal node does not increase the size substantially.

4 Updates to the Range Tree

We now discuss the process of updating the search tree when a prefix is inserted or deleted. We describe at a high level the type of changes that occur in the range tree when a prefix

$r = [b_r, e_r]$ is inserted or deleted; we defer the (fairly complicated) details for the final paper. Conceptually, the update can be divided into three phases (which can be combined in an implementation), 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 4.1 *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 $O(t \log_t n)$.*

5 Multiword Addresses: IPv6 or Multicast

In this section, we describe an extension of our scheme to multiword prefixes. While IPv4 uses 32 bit IP addresses, the next generation, IPv6, will use 128 bit addresses. 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. 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 will only result in the use of a small branching factor and hence high tree heights (even using 1 key and 1 pointer will fill a 256 bit cache line, necessitating a branching factor of only 2). Thus the real reason we wish to use small word sizes so that we can have a large branching factor and hence a small tree height.

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 present in the database.

Let $S = \{r_1, r_2, \ldots, r_m\}$ be a set of m prefixes, each upto k-words long. We say that the first word includes bits 1 through W; the second word include bits $W + 1$ through $2W$, and so on. (We ignore the zero-length prefix, namely, $*$, from our discussion—this is the default prefix and need not be stored explicitly in the data structure.)

We first build a multiway range tree T_0 on the *first words* of the prefixes in S. We observe that the prefixes of S fall in two categories: (1) those with fewer than W bits—these behave just like normal prefixes, they correspond to finite length ranges, and they are treated like the single-word prefixes of the preceding section; (2) those with W or more bits—they behave like *point ranges*, and treated like individual keys in the range tree.

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 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 $=$. (In the single word scheme, we had combined the $<$ and $=$, resulting in just two subsets, $\leq$, and $>$.) The branch $<$ leads to the set of keys that are strictly smaller than x, while $>$ leads to keys strictly bigger than x. *The third branch, $=$, is fundamentally different: it points to a range tree on the set of prefixes whose first word is x.* Figure 5 shows an example of our general construction of multiword search tree.

Let us now describe the general construction a bit more precisely. 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.

Remarkably, it is possible to prove that if we do the obvious trick of removing duplicate keys in each range tree, the result is a correct algorithm but with poor performance. Intuitively, this is because of the following. To avoid paying $\log_t n$ time in each word search, it must be the case that if we pay a large search time in the first word search, then we should pay less

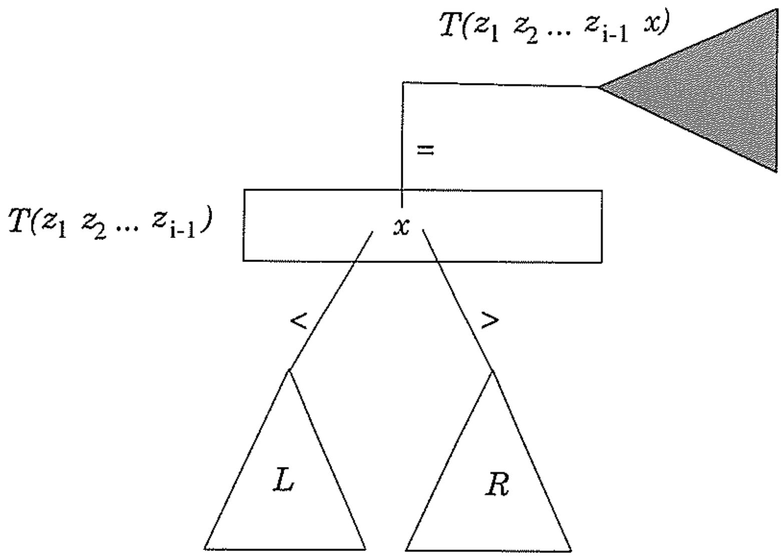

Figure 5: Range tree for multiword prefixes.

in the remaining word searches. Suppose there are a large number of prefixes that start with first word X. Then we may have to pay a large search time in searching for the second and subsequent words after matching X. If we do not remove duplicates, the search in the first column for X will terminate very early, compensating for the slower later word searches. For example, if there are $n/2$ copies of X, it is easy to see that the first word search will terminate after the second level. On the other hand, if we replace the $N/2$ copies of X with a single copy, search for X can take $log_t(n) - 1$ time. In general, removing duplicates can lead to a search time of $\log_t n - 1 + \log_t n - 2 + \ldots$, which leads back to the multiplicative factor we were trying to avoid.

Search Algorithm: Given a packet header $(p_1 \cdot p_2 \cdots p_k)$, the search for its best matching prefix is performed 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.

The 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.

6 Experimental Results

This section describes the experimental setup and measurements we use to compare the performance of our schemes with other prominent IP lookup schemes. In order to carry out a platform independent and fair comparison of different schemes, we first determine appropriate performance metrics, and discuss the model used in evaluating these algorithms.

6.1 The Model

We will consider software platforms using modern processors such as the Pentium and the Alpha. When a READ request for a single word in memory is made, an entire *cache line* is fetched into the cache. This is important because access time for remaining words in the cache line is much smaller than a single memory READ. 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); for example, a single memory access retrieving upto 512 and even 1024 consecutive bits can be supported. Given the significant difference between main memory and cache access times, for both hardware and software implementations, the number of memory accesses should be measured in terms of number of distinct cache-line accesses, as opposed to the number of memory words retrieved. This metric is also independent of the exact memory cycle time for various random access memories. We note that memory accesses are the most dominant component of overall processing time for IP lookup schemes, and therefore we can approximate total processing time by the number of memory accesses.

In a hardware implementation, on-chip SRAM is usually very limited, and only that portion of data structure that is relevant to search is kept in the on-chip memory. Other parts, required only during updates, are kept in off-chip memory. IP lookup schemes proposed in the literature (except multibit tries) that have poor update time usually keep two separate copies of the structure: one for lookup and another one for updates. When update is complete, the corresponding copy is atomically copied onto the lookup copy. In order to compare our scheme with these data structures, we separately state our lookup-relevant memory and update memory.

6.2 Implementation

We implemented our range tree scheme in the C programming language on a UNIX machine. The implementation used the following representations for the range tree and heap structures:

- **[Range Tree:]** All the children of a node are allocated in a single array, and the node maintains a pointer to its *leftmost child.* This results in significant memory saving, because otherwise each node will have to store t child pointers, for arity t. This representation is a modification of the standard *child-sibling* pointer representation of variable arity trees using fixed number of pointers per node.

- **[Prefix-Heaps:]** We represent the set of prefixes associated with a node by a *bitmap* of $W+1$ bits. The i-th bit in the bitmap indicates whether or not a prefix of length $i-1$ is associated with that node. This representation not just reduces memory requirement for the heap, but also makes heap operations (insert, delete, merge two heaps) as cheap as a few bit manipulation instructions.

Suitable counters were used in programs to count number of memory accesses (which simply equals number of node accesses).

6.2.1 Empirical Results for IPv4

Experiments were conducted using a number of routing table snapshots obtained from the IPMA database [10], however only a representative 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 1 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 1: Mae-East database: 41,456 prefixes, 62273 distinct keys

Similar observations for the smaller PacBell dataset are presented in Table 2. The PacBell dataset is interesting, as the number of *distinct* keys is less than number of prefixes. This happens because of a large number of adjacent prefixes. Though it has more than half as many prefixes in the Mae-East dataset, the number of distinct keys is about a third of Mae-East. This is reflected in memory requirement and height of the range tree, which are a function of number of keys.

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 2: 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. Both the tables above 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, total memory requirement as well as the height of the tree. These incremental gains decrease with increasing arity, as the increase in guaranteed minimum utilization of $\frac{t-1}{2t-1}$ decreases with increasing arity.

The worst-case update time for our scheme is calculated assuming node split/merge at each level of the tree. We wanted to evaluate the average update performance of the scheme. To do that, 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. This experiment ensured that the size of the database remained comparable to the original prefix snapshot, and therefore should reflect realistic updates to an actual database of this size. Table 3 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 3: 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.

Another interesting experimental observation is regarding the effect of insertion order. If keys are inserted in sorted order, nodes that are split never receive any new keys. So almost all the nodes have the minimum guaranteed arity. However, if keys are inserted in random order, average node arity improves. Thus, as the initial structure undergoes a large number of random updates, the performance of the data structure in fact should improve.

6.2.2 IPv6 Implementation and Results

Practical IPv6 prefix databases are not available. The actual performance of our scheme depends significantly on the hierarchical nature of the prefix data, as it affects sizes of next-word trees. Thus, we cannot provide any typical performance numbers. However, the exact worst case lookup and update times can be determined analytically.

IPv6 addresses are 128 bit long. So 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.

6.3 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 4 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 4: 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. For trie-based schemes, the cost of an update depends on the number of bits in a stride. Note that large strides increase update time and worst-case memory needs *exponentially*. Most papers on multibit tries only report the memory costs on typical databases such as Mae-East; their worst case memory needs are much worse! The Patricia trie scheme has a stride of 1, and hence allows fast updates. But for the same reason, its lookup time is unacceptably bad.

In order to compare these schemes for IPv4 lookup performance and memory requirement, we assume a 512 bit cache-line. See Table 5. For multi-bit tries and multi-column search schemes, optimal parameters are chosen so that each node access is equal to one memory access and lookup performance is optimized. Measurements for other schemes are from Srinivasan [15], and are for a smaller database of 38,816 prefixes. Measurements for our proposed scheme use the more recent version of the Mae-East database consisting of 41,456 prefixes (which should only favor existing schemes over ours.) In the table, the lookup time is measured as the worst-case number of memory accesses required for each scheme. The notes attached to each row give more details of each comparison.

Scheme	Lookup time	Memory	Update time
Patricia trie	32	3.26 MB	32
Multibit tries (Optimal stride)	5	0.39 MB	Rebuild (See #1)
Multibit tries (Fixed stride)	5	$>$ 0.39 MB	224 (See #1)
Lulea compressed trie	9	0.16 MB	2^{16} (See #2)
LC Trie	7	0.70 MB	Rebuild (See #3)
Binary search	15	0.95 MB	Rebuild
Multiway search	12	0.95 MB	Rebuild
Prefix length binary search	5	1.60 MB	Rebuild
Multiway Range Tree	5	0.44 MB	450

Table 5: Update performance for IPv4 prefix data.

For all the existing non-trie schemes, the data structure has to be rebuilt for each update. Rebuild time is usually of the order of seconds, and hence incomparably bad. Our proposed scheme is better than these schemes in terms of memory requirement and search time as well. Our search time is deterministic, as against the binary search on prefix length scheme, for which actual number of memory accesses is dependent on number of collisions.[2]

Among trie-based schemes, the Patricia trie is unacceptable due to its poor lookup time. Our proposed scheme provides better search time than LC Trie and Lulea schemes, while still providing fast updates.

- [#1.] Multibit tries as described in [15] come in two flavors: fixed stride and variable stride. Variable size strides, where optimal stride sizes are computed using dynamic programming, lead to good memory bound, but they require a rebuild per update *if this optimality is to be preserved.* In [15], the authors suggest recomputing optimal strides infrequently, such as once a day. However, in between rebuilds, the optimal behaviour of the trie is no longer guaranteed. For completeness sake, we mention that optimizing the stride size takes about 1.6 seconds [15].

  By contrast, multibit tries using fixed strides have good search and update times, but their memory bound is worse. In terms of our performance metrics (insertion time, memory, and lookup speed), the trie most competitive with our scheme should have uniform strides $6, 6, 5, 5, 5$. However, we have not been able to determine the worst-case memory for the fixed stride sizes, and therefore left that entry blank in the table.

[2]Alternately, if such a scheme is implemented using perfect hashing, the amount of memory used is non-deterministic.

- [#2.] The Lulea scheme performs a large amount of precomputation and has an initial stride of 16 bits. Thus, any update to a prefix of short length will induce an overhead of at least 2^{16} to recompute this bitmap and to adjust array locations.
- [#3.] Similarly, the LC Trie replaces complete subtries of height j by a node of arity 2^j. So deletion of a single prefix may render a previously-complete subtrie incomplete. More importantly, their strategy of laying out trie nodes contiguously leads to the need for complete rebuilding.

In IPv6, the prefix lengths can be $W = 128$ bits long. None of the trie-based schemes are likely to scale well for such long addresses, because longer addresses lead to a proportionately (factor of $128/32 = 4$) larger search time. But an even bigger problem may be the following: Almost all trie-based schemes use a large initial stride to reduce the trie height. For example, a $(16, 8, 8)$ trie [1] requires 3 memory accesses. However, scaling this to IPv6 so that the worst-case lookup takes at most $3 * 4 = 12$ accesses will require many 16-bit trie nodes, which make the memory requirement infeasible. On the other hand, the non-trie schemes have poor update times. Thus our proposed scheme is the only scheme which is good with respect to all four metrics.

7 Conclusion

We have described a new IP lookup scheme that is competitive for IPv4 in terms of memory and lookup speed and yet has fast update times. For example, on Mae-East database using 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. The average update performance is significantly better (less than 10% of the worst-case). However, the major attraction of our scheme is that it scales well to IPv6 and multicast addresses, while preserving the fast lookup and update times. It is the first scheme we know of that possesses all these properties.

References

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

[2] Gene Cheung and Steven McCanne. Optimal Routing Table Design for IP Address Lookups Under Memory Constraints. *Proceedings of INFOCOM '99*, March 21, 1999, New York, NY.

[3] Tzi-cker Chiueh and Prashant Pradhan. High Performance IP Routing Table Lookup using CPU Caching *Proceedings of IEEE INFOCOMM '99.*

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

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

[6] C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *Proceedings of SIGCOMM 97*, October 1997.

[7] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multi-way and Multicolumn Search. *Proceedings of IEEE INFOCOM '98*, April 1998.

[8] Nick McKeown, Martin Izzard, Adisak Mekkittikul, Bill Ellersick, and Mark Horowitz. The Tiny Tera: A Packet Switch Core. *IEEE Micro*, Jan 1997.

[9] Merit. Routing table snapshot on 14 Jan 1999 at the Mae-East NAP. ftp://ftp.merit.edu/statistics/ipma.

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

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

[12] Peter Newman, Greg Minshall, and Larry Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, January 1997.

[13] Radia Perlman. *Interconnections, Bridges and Routers.* Addison-Wesley, 1992.

[14] S. Sikka Fast and Efficient Memory Allocators. *M.S. Thesis, Washington University*, June 1999.

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

[16] J. Turner. Design of a Gigabit ATM Switch. *Proceedings of IEEE INFOCOM '97*, March 1997.

[17] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. *Computer Communication Review*, October 1997.