# Scalable High Speed IP Routing Lookups

Marcel Waldvogel†, George Varghese‡, Jon Turner‡, Bernhard Plattner†

†Computer Engineering and Networks Laboratory
ETH Zürich, Switzerland
{waldvogel,plattner}@tik.ee.ethz.ch

‡Computer and Communications Research Center
Washington University in St. Louis, USA
{varghese,jst}@ccrc.wustl.edu

## Abstract

Internet address lookup is a challenging problem because of increasing routing table sizes, increased traffic, higher speed links, and the migration to 128 bit IPv6 addresses. IP routing lookup requires computing the best matching prefix, for which standard solutions like hashing were believed to be inapplicable. The best existing solution we know of, BSD radix tries, scales badly as IP moves to 128 bit addresses. Our paper describes a new algorithm for best matching prefix using binary search on hash tables organized by prefix lengths. Our scheme scales very well as address and routing table sizes increase: independent of the table size, it requires a worst case time of $\log_2(address\,bits)$ hash lookups. Thus only 5 hash lookups are needed for IPv4 and 7 for IPv6. We also introduce Mutating Binary Search and other optimizations that, for a typical IPv4 backbone router with over 33,000 entries, considerably reduce the average number of hashes to less than 2, of which one hash can be simplified to an indexed array access. We expect similar average case behavior for IPv6.

## 1 Introduction

The Internet is becoming ubiquitous: everyone wants to join in. Since the advent of the World Wide Web, the number of users, hosts, domains, and networks connected to the Internet seems to be exploding. Not surprisingly, network traffic is doubling every few months. The proliferation of multimedia networking applications and devices is expected to give traffic another major boost.

The increasing traffic demand requires three key factors to keep pace if the Internet is to continue to provide good service: link speeds, router data throughput, and packet forwarding rates.[1] Readily available solutions exist for the first two factors: for example, fiber-optic cables can provide faster links,[2] and switching technology can be used to move packets from the input interface of a router to the corresponding output interface at gigabit speeds. Our paper deals with the third factor, packet forwarding, for which current techniques perform poorly as network speeds increase.

The major step in packet forwarding is to lookup the destination address (of an incoming packet) in the routing database. While there are other chores, such as updating TTL fields, these are computationally inexpensive compared to the major task of address lookup. Data link Bridges have been doing address lookups at 100 Mbps [Dig95] for many years. However, bridges only do exact matching on the destination (MAC) address, while Internet routers have to search their database for the *longest prefix* matching a destination IP address. Thus standard techniques for exact matching, such as perfect hashing, binary search, and standard Content Adressable Memories (CAMs) cannot directly be used for Internet address lookups.

Prefix matching was introduced in the early 1990s, when it was foreseen that the number of endpoints and the amount of routing information would grow enormously. The address classes A, B, and C (allowing sites to have 24, 16, and 8 bits respectively for addressing) proved too inflexible and wasteful of the address space. To make better use of this scarce resource, especially the class B addresses, bundles of class C networks were given out instead of class B addresses. This resulted in massive growth of routing table entries. So, in turn, Classless Inter-Domain Routing (CIDR) [F+93] was deployed, to allow for arbitrary aggregation of networks to reduce routing table entries.

To reduce routing table space, aggregation is done aggressively. Suppose all the subnets in a big network have identical routing information except for a single, small subnet that has different information. Instead of having multiple routing entries for each subnet in the large network, just two entries are needed: one for the big network, and a more specific one for the small subnet (which has preference, if both should match). This results in better usage of the available IP address space and decreases the amount of routing table entries. On the other hand, the processing power needed for forwarding lookup is increased.

Thus today an IP router's database consists of a number of *address prefixes*. When an IP router receives a packet, it must compute which of the prefixes in its database has the longest match when compared to the destination address in the packet. The packet is then forwarded to the output link associated with that prefix. For example, a forwarding database may have the prefixes $P1 = 0101$, $P2 = 0101101$ and $P3 = 010110101011$. An address whose first 12 bits are 010101101011 has longest matching prefix $P1$. On the other hand, an address whose first 12 bits are 010110101101 has longest matching prefix $P3$.

The use of best matching prefix in forwarding has allowed IP routers to accomodate various levels of address hierarchies, and has allowed different parts of the network to have different views of the

[1] In our paper, we distinguish between *routing* (a process that computes a database mapping destination networks to output links) and *forwarding* (a process by which a routing database is consulted to decide which output link a single packet should be forwarded on.) Route computation is less time critical than forwarding because forwarding is done for each packet, while route computation needs to be done only when the topology changes.

[2] For example, MCI is currently upgrading its lines from 45 Mbits/s to 155 Mbits/s; they plan to switch to 622 Mbits/s within a year.

address hierarchy. Given that best matching prefix forwarding is necessary for hierarchies, and hashing is a natural solution for exact matching, a natural question is: "Why can't we modify hashing to do best matching prefix." However, for several years now, it was considered not to be "apparent how to accommodate hierarchies while using hashing, other than rehashing for each level of hierarchy possible" [Skl93].

Our paper describes a novel algorithmic solution to longest prefix match, using binary search over hash tables organized by the length of the prefix. Our solution requires a worst case complexity[3] of $O(\log_2 W)$, with $W$ being the length of the address in bits. Thus, for the current Internet protocol suite (IPv4) with 32 bit addresses, we need at most 5 hash lookups. For the upcoming IP version 6 (IPv6) with 128 bit addresses, we can do lookup in at most 7 steps, as opposed to 128 in current algorithms (see Section 2), giving an *order of magnitude performance improvement*. Using perfect hashing, we can lookup 128 bit IP addresses in at most 7 memory accesses. This is significant because on current RISC processors, hash functions can be found whose computation is cheaper than a memory access.

In addition, we use several optimizations to *significantly* reduce the average number of hashes needed. For example, our analysis of an IPv4 forwarding table from an Internet backbone router at the Mae-East network access point (NAP) [Mer96] show an average case performance of less than two hashes, where the first hash can be replaced by a simple index table lookup.

The rest of the paper is organized as follows. Section 2 describes drawbacks with existing approaches to IP lookups. Section 3 describes our basic scheme in a series of refinements that culminate in the basic binary search scheme. Section 4 describes a series of important optimizations to the basic scheme that improve average performance. Section 5 describes our implementation, including algorithms to build the data structure and perform insertions and deletions. Section 6 describes performance measurements using our scheme for IPv4 addresses, and performance projections for IPv6 addresses. We conclude in Section 7 by assessing the theoretical and practical contributions of this paper.

## 2 Existing Approaches to IP Lookup

We survey existing approaches to IP lookups and their problems. We discuss approaches based on modifying exact matching schemes, trie based schemes, hardware solutions based on parallelism, proposals for protocol changes to simplify IP lookup, and caching solutions. For the rest of this paper, we use BMP as a shorthand for Best Matching Prefix.

**Modifications of Exact Matching Schemes** Classical fast lookup techniques such hashing and binary search do not directly apply to the best matching prefix (BMP) problem since they only do exact matches. A modified binary search technique, originally due to Butler Lampson, is described in [Per92]. However, this method requires $\log_2 2N$ steps, with $N$ being the number of routing table entries. With current routing table sizes, the worst case would be 17 data lookups, each requiring at least one costly memory access. As with any binary search scheme, the average number of accesses is $\log_2(2N) - 1$. A second classical solution would be to reapply any exact match scheme for each possible prefix length [Skl93]. This is even more expensive, requiring $W$ iterations of the exact match scheme used (e.g. $W = 128$ for IPv6).

[3] This assumes assuming $O(1)$ for hashing, which can be achieved using perfect hashing, although limited collisions do not affect performance significantly.

**Trie Based Schemes** The most commonly available IP lookup implementation is found in the BSD kernel, and is a radix trie implementation [Skl93]. If $W$ is the length of an address, the worst-case time in the basic implementation can be shown to be $O(W^2)$. Current implementations have made a number of improvements on Sklower's original implementation. The worst case was improved to $O(W)$ by requiring that the prefix be contiguous (previously non-contiguous masks were allowed, a feature which was never used). Despite this, the implementation requires up to 32 or 128 costly memory accesses (for IPv4 or IPv6, respectively). Tries also can have large storage requirements.

**Hardware Solutions** Hardware solutions can potentially use parallelism to gain lookup speed. For exact matches, this is done using Content Addressable Memories (CAMs) in which every memory location, in parallel, compares the input key value to the content of that memory location.

Some CAMs allow a mask of bits that must be matched. Although there are expensive so-called ternary CAMs available allowing a mask to be specified per word, the mask must typically be specified in advance. It has been shown that these CAMs can be used to do BMP lookups [MF93, MTW95], but the solutions are usually expensive.

Large CAMs are usually slower and much more expensive than ordinary memory. Typical CAMs are small, both in the number of bits per entry and the number of entries. Thus the CAM memory for large address/mask pairs (256 bits needed for IPv6) and a huge amount of prefixes appears (currently) to be very expensive. Another possibility is to use a number of CAMs doing parallel lookups for each prefix length. Again, this seems expensive. Probably the most fundamental problem with CAMs is that CAM designs have not historically kept pace with improvements in RAM memory. Thus a CAM based solution (or indeed *any* hardware solution) runs the risk of being made obselete, in a few years, by software technology running on faster processors and memory.

**Protocol Based Solutions** One way to get around the problems of IP lookup is to have extra information sent along with the packet to simplify or even totally get rid of IP lookups at routers. Two major proposals along these lines are IP Switching [NMH97] and Tag Switching [CV95, CV96, $R^+96$]. Both schemes require large, contiguous parts of the network to adopt their protocol changes before they will show a major improvement. The speedup is achieved by adding information on the destination to every IP packet.

In IP Switching, this is done by associating a flow of packets with an ATM Virtual Circuit; in Tag Switching, this is done by adding a "tag" to each packet, where a "tag" is a small integer that allows direct lookup in the router's forwarding table. Tag switching is based on a concept originally described by Chandranmenon and Varghese ([CV95, CV96]) using the name "threaded indices". The current tag switching proposal[$R^+96$] goes further than threaded indices by adding a stack of indices to deal with hierarchies.

Neither scheme can completely avoid ordinary IP lookups. Both schemes require the ingress router (to the portions of the network implementing their protoocol) to perform a full routing decision. In their basic form, both systems potentially require the boundary routers between autonomous systems (e.g., between a company and its ISP or between ISPs) to perform the full forwarding decision again, because of trust issues, scarce resources, or different views of the network. Scarce resources can be ATM VCs or tags, of which only a small amount exists. Thus towards the backbone, they need to be aggregated; away from the backbone, they need to be separated again.

Different views of the network can arise because systems often know more details about their own and adjacent networks, than about networks further away. Although Tag Switching addresses that problem by allowing hierarchical stacking of tags, this affects routing scalability. Tag Switching assigns and distributes tags based on routing information; thus every originating network now has to know tags in the destination networks. Thus while both tag switching and IP switching can provide good performance within a level of hierarchy, neither solution currently does well at hierarchy boundaries without scaling problems.

**Caching** For years, designers of fast routers have resorted to caching to claim high speed IP lookups. This is problematic for several reasons. First, information is typically cached on the entire address, potentially diluting the cache with hundreds of addresses that map to the same prefix. Second, a typical backbone router of the future may have hundreds of thousands of prefixes and be expected to forward packets at Gigabit rates. Although studies have shown that caching in the backbone can result in hit ratios up to and exceeding 90 percent [Par96, NMH97], the simulations of cache behavior were done on large, fully associative caches which commonly are implemented using CAMs. CAMs, as already mentioned, are usually expensive. It is not clear how set associative caches will perform and whether caching will be able keep up with the growth of the Internet. So caching does help, but does not avoid the need for fast BMP lookups, especially in view of current network speedups.

**Summary** In summary, all existing schemes have problems of either performance, scalability, generality, or cost. Lookup schemes based on tries and binary search are (currently) too slow and do not scale well; CAM solutions are expensive and carry the risk of being quickly outdated; tag and IP switching solutions require widespread agreement on protocol changes, and still require BMP lookups in portions of the network; finally, locality patterns at backbone routers make it infeasible to depend entirely on caching.

We now describe a scheme that has good performance, excellent scalability, and does not require protocol changes. Our scheme also allows a cheap, fast software implementation, and also a more expensive (but faster) hardware implementation.

## 3 Basic Binary Search Scheme

Our basic algorithm is based on three significant ideas, of which only the first has been reported before. First, we use hashing to check whether an address $D$ matches any prefix of a particular length; second, we use binary search to reduce number of searches from linear to logarithmic; third, we use precomputation to prevent backtracking in case of failures in the binary search of a range. Rather than present the final solution directly, we will gradually refine these ideas in Section 3.1, Section 3.2, and Section 3.5 to arrive at a working basic scheme. We describe further optimizations to the basic scheme in the next section.

### 3.1 Linear Search of Hash Tables

Our point of departure is a simple scheme that does linear search of hash tables organized by prefix lengths. We will improve this scheme shortly to do binary search on the hash tables.

The idea is to look for all prefixes of a certain length $L$ using hashing and use multiple hashes to find the best matching prefix, starting with the largest value of $L$ and working backwards. Thus we start by dividing the database of prefixes according to lengths.

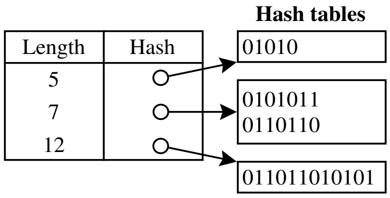



Figure 1: Hash Tables for each possible prefix length

Assuming a particularly tiny routing table with four prefixes of length 5, 7, 7, and 12, respectively, each of them would be stored in the hash table for its length (Figure 1). So each set of prefixes of distinct length is organized as a hash table. If we have a sorted array $L$ corresponding to the distinct lengths, we only have 3 entries in the array, with a pointer to the longest length hash table in the last entry of the array.

To search for address $D$, we simply start with the longest length hash table $l$ (i.e. 12 in the example), and extract the first $l$ bits of $D$ and do a search in the hash table for length $l$ entries. If we succeed, we have found a BMP[4]; if not, we look at the first length smaller than $l$, say $l'$ (this is easy to find if we have the array $L$ by simply indexing one position less than the position of $l$), and continuing the search.

More concretely, let $L$ be an array of records. $L[i].length$ is the length of prefixes found at position $i$, and $L[i].hash$ is a pointer to a hash table containing all prefixes of length $L[i].length$. The resulting code is shown in Figure 2.

**Function** LinearSearch($D$) (* search for address $D$ *)
Initialize $BMP$ to the empty string;
$i$ := Highest index in array $L$;
**While** ($BMP = nil$) and ($i \geq 0$) do
  Extract the first $L[i].length$ bits of $D$ into $D'$;
  $BMP$ := Search($D'$, $L[i].hash$); (* search hash for $D'$ *)
  $i := i - 1$;
**Endwhile**

Figure 2: Linear Search

### 3.2 Binary Search of Hash Tables

The previous scheme essentially does (in the worst case) linear search among all distinct string lengths. Linear search requires $O(W)$ expected time (more precisely, $O(W_{dist})$, where $W_{dist} \leq W$ is the number of distinct lengths in the database.)

A better search strategy is to use binary search on the array $L$ to cut down the number of hashes to $O(\log_2 W_{dist})$. However, for binary search to work, we need *markers* in tables corresponding to shorter lengths to point to prefixes of greater lengths. Markers are needed to direct binary search to look for matching prefixes of greater length. Here is an example to illustrate the need for markers.

Suppose we have the prefixes $P1 = 0$, $P2 = 00$, $P3 = 111$ (Figure 3 (b)). Assume that the zeroth entry of $L$ points to $P1$'s hash table, the first to $P2$'s hash table, and the second points to $P3$'s hash table. Suppose we search for 111. Binary search (a) would start at the middle hash table and search for 11 in the hash table containing $P2$ (the triangles denote a pointer to the hash table

[4] Recall that BMP stands for Best Matching Prefix. We use this abbreviation through the rest of the paper

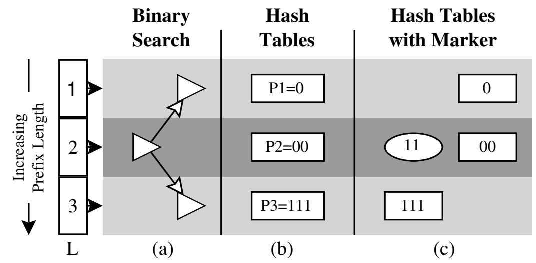



Figure 3: Binary Search on Hash Tables

| | | |
|---|---|---|
| Total entries | 33199 | 100% |
| Entries needing no markers | 4743 | 14% |
| Entries needing 1 marker | 22505 | 68% |
| Entries needing 2 markers | 3562 | 11% |
| Entries needing 3 markers | 2389 | 7% |
| Total markers requested (before sharing) | 36796 | 111% |
| Total markers | 8699 | 26% |
| Pure markers | 8336 | 25% |

Table 1: Marker Overhead for Backbone Forwarding Table

to search). It would fail and have no indication that it should search among the longer prefix tables for a better matching prefix. To fix this problem, we simply add a marker entry 11 to the middle table. Now when binary search is done for 111, we will lookup 11 in the middle hash table and find the marker node. This can be used to direct binary search to the lower half of the table.

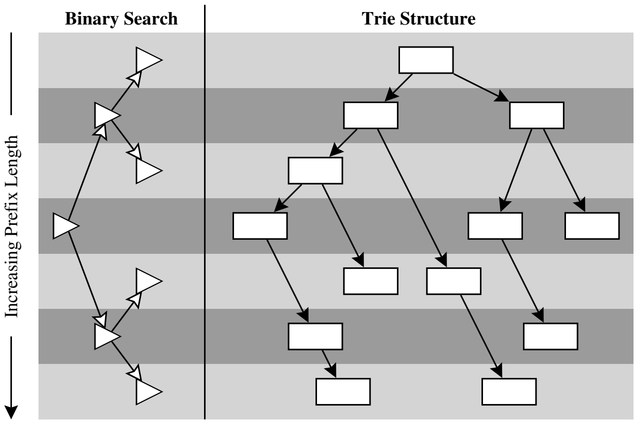



Figure 4: Binary Search on Trie Levels

Each hash table (markers plus real prefixes) can be thought of as a horizontal layer of a trie corresponding to some length $L$ (except that the hash table contains the complete path to that layer of each entry in that layer). Our basic schemes is essentially doing binary search on the levels of a trie (Figure 4). We start by doing a hash on prefixes corresponding to the median length of the trie. If we match, we search the upper half of the trie; if we fail we search the lower half of the trie.

Figure 4 and other figures describing search order contain several elements: (1) Horizontal stripes grouping all the elements of a specified prefix length, (2) a trie containing the prefixes, shown on the right of the figure and rooted on the top of the figure, and (3) a binary tree, shown on the left of the figure and rooted at the left, which depicts all possible paths that binary search can follow. We will use *upper half* to mean the half of the trie with prefix lengths strictly less than the median length. We also use *lower half* for the portion of the trie with prefix lengths strictly greater than the median length. It is important to understand the conventions in Figure 4 to understand the later figures and text.

## 3.3 Reducing Marker Storage

The following definitions are useful before proceeding. For a prefix P in the table, define $Level(P)$ to be the integer $i$ for which $L[i].length = length(P)$ (i.e., the index of the entry in $L$ that points to $P$'s hash table). Also, "up" to refers to shorter, "down" to longer prefixes.

How many markers do we need? A naive view would indicate placing a marker for prefix $P$ at all levels in $L$ higher than the level of $P$. However, it suffices to place markers at all levels in $L$ *that could be visited by binary search when looking for an entry whose BMP is* $P$. This reduces the number of markers to at most $\log_2 W$ per real prefix, which keeps the storage expansion modest. More precisely, if the $Level(P)$ is written down in binary as $a_1, a_2, \ldots, a_n$, then we need a marker at each level $a_1 a_2, \ldots, a_k, 0, 0, \ldots, 0$ such that $a_k = 1$. (We assume that $L$ is padded so that its size is a power of 2). In fact, the number of marker nodes is limited by the number of 1 bits in $Level(P)$. Clearly this results in a logarithmic number of markers.

In the typical case, many prefixes will share markers (Table 1), reducing the marker storage further. In our sample routing database [Mer96], the storage required will increase by 25%. However, it is easy to give a worst case example where the storage needs require $O(\log_2 W)$ markers per prefix. (Consider $N$ prefixes whose first $\log_2 N$ bits are all distinct and whose remaining bits are all 1's).

## 3.4 Problems with Backtracking

**Function** NaiveBinarySearch($D$) (* search for address $D$ *)
Initialize search range $R$ to cover the whole array $L$;
**While** $R$ is not a single entry do
  Let $i$ correspond to the middle level in range $R$;
  Extract the first $L[i].length$ bits of $D$ into $D'$;
  Search($D'$, $L[i].hash$); (* search hash table for $D'$ *)
  **If** found then set $R$ := lower half of $R$ (*longer prefixes*)
    **Else** set R := upper half of $R$; (*shorter prefixes*)
  **Endif**
**Endwhile**

Figure 5: Naive Binary Search

Binary search of hash tables can be expressed as shown in Figure 5. Unfortunately, this algorithm is not correct as it stands and *does not* take logarithmic time if implemented naively. The problem is that while markers are good things (they lead to potentially better prefixes lower in the table), they can also cause the search to follow false leads which may fail. In case of failure, we would have to modify the binary search (for correctness) to backtrack and search the upper half of $R$ again. Such a naive modification can lead us back to linear time search. An example will clarify this.

First consider the prefixes $P1 = 1$, $P2 = 00$, $P3 = 111$. As discussed above, we add a marker to the middle table so that the middle hash table contains 00 (a real prefix) and 11 (a marker pointing down to $P3$). Now consider a search for 110. We start at the middle hash table and get a hit; thus we search the third hash table for 110 and fail. But the correct best matching prefix is at the first level hash table — i.e., $P1$. The marker indicating that there will be longer prefixes, indispensible to find $P3$, was misleading in this case; so apparently, we have to go back and search the upper half of the range.

The fact that each entry contributes at most $\log_2 W$ markers may cause some readers to suspect that the worst case with backtracking is limited to $O(\log^2 W)$. This is incorrect. The worst case is $O(W)$. The worst-case example for say $W$ bits is as follows: we have a prefix $P_i$ of length $i$, for $1 \leq i < W$ that contains all 0s. In addition we have the prefix $Q$ whose first $W - 1$ bits are all zeroes, but whose last bit is a 1. If we search for the $W$ bit address containing all zeroes then we can show that binary search with backtracking will take $O(W)$ time and visit every level in the table. (The problem is that every level contains a false marker that indicates the presence of something better below.)

### 3.5 Precomputation to Avoid Backtracking

We use precomputation to avoid backtracking when we shrink the current range $R$ to the lower half of $R$ (which happens when we find a marker at the mid point of $R$). Suppose every marker node $M$ is a record that contains a variable $M.bmp$, which is the value of the best matching prefix of the marker $M$. $M.bmp$ can be precomputed when the marker $M$ is inserted into its hash table. Now, when we find $M$ at the mid point of $R$, we indeed search the lower half, but we also *remember* the value of $M.bmp$ as the current best matching prefix. Now if the lower half of $R$ fails to produce anything interesting, we need not backtrack, because the results of the backtracking are already summarized in the value of $M.bmp$. The new code is shown in Figure 6.

**Function** BinarySearch($D$) (* search for address $D$ *)
Initialize search range $R$ to cover the whole array $L$;
Initialize $BMP$ found so far to null string;
**While** $R$ is not empty do
    Let $i$ correspond to the middle level in range $R$;
    Extract the first $L[i].length$ bits of $D$ into $D'$;
    $M$ := Search($D'$, $L[i].hash$); (* search hash for $D'$ *)
    **If** $M$ is nil **Then** set $R$ := upper half of $R$; (* not found *)
    **Elseif** $M$ is a prefix and not a marker
    **Then** $BMP := M.bmp$; break; (* exit loop *)
    **Else** (* $M$ is a pure marker, or marker and prefix *)
        $BMP := M.bmp$; (* update best matching prefix so far *)
        $R$ := lower half of $R$;
    **Endif**
**Endwhile**

Figure 6: Binary Search

The standard invariant for binary search when searching for key $K$ is: "$K$ is in range $R$". We then shrink $R$ while preserving this invariant. The invariant for this algorithm, when searching for key $K$ is: "EITHER (The Best Matching Prefix of $K$ is BMP) OR (There is a longer matching prefix in $R$)".

It is easy to see that initialization preserves this invariant, and each of the search cases preserves this invariant (this can be established using an inductive proof.) Finally, the invariant implies the correct result when the range shrinks to 1. Thus the algorithm works correctly; also since it has no backtracking, it takes $O(\log_2 W_{dist})$ time.

## 4 Refinements to Basic Scheme

The basic scheme described in Section 3 takes just 7 hash computations, in the worst case, for 128 bit IPv6 addresses. However, each hash computation takes at least one access to memory; at gigabit speeds each memory access is significant. Thus, in this section, we explore a series of optimizations that exploit the deeper structure inherent in the problem to reduce the average number of hash computations.

### 4.1 Asymmetric Binary Search

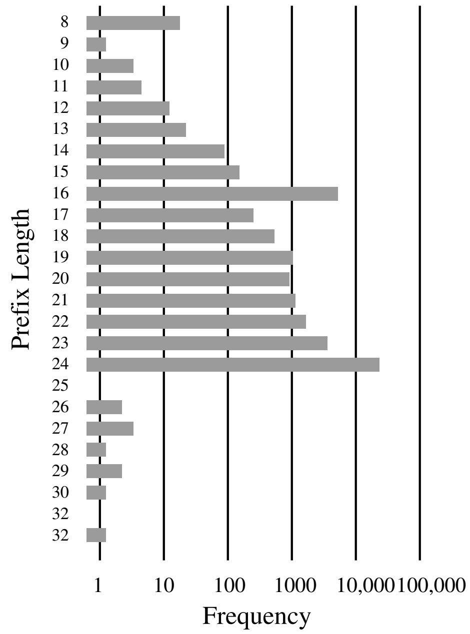



Figure 7: Histogram of the Prefix Length Distribution

We first describe a series of simple-minded optimizations. Our main optimization, mutating binary search, is described in the next section. A reader can safely skip to Section 4.2 on a first reading.

The current algorithm is a fast, yet very general, BMP search engine. Usually, the performance of general algorithms can be improved by tailoring them to the particular datasets they will be applied to. As can be seen in Figure 7, the distribution of a typical backbone router's forwarding table as obtained from [Mer96], the entries are not equally distributed over the different prefix lengths. All the concepts we described below apply to any set of addresses; however, we will quantify the potential improvements using the existing table.

As the first improvement, which has already been mentioned and used in the basic scheme, the search can be limited to those prefix lengths which do contain at least one entry, reducing the worst case number of hashes from $\log_2 W$ (5 with $W = 32$) to $\log_2 W_{dist}$ (4.5 with $W_{dist} = 23$, the number of non-empty buckets in the histogram), as shown in Figure 8. (While this is an improvement for the worst case, in this case, it harms the average performance, as we will see later.)

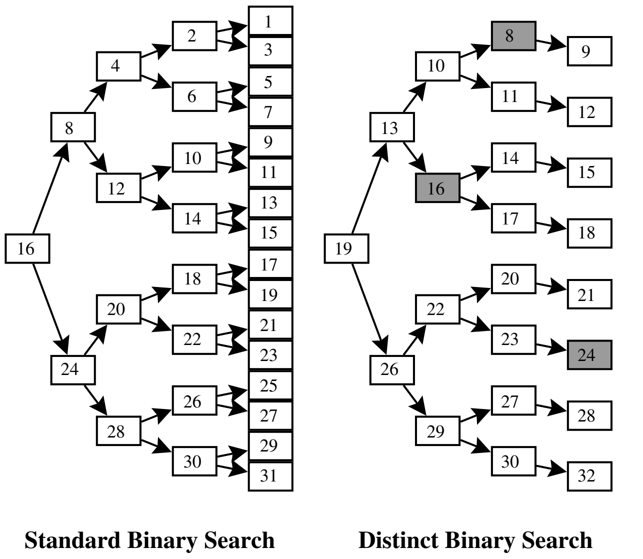



Figure 8: Search Trees for Standard and Distinct Binary Search

A more promising approach is to change the tree structure to search in the most promising prefix length layers first, introducing asymmetry into the binary tree. While this will improve average case performance, introducing asymmetries will not improve the maximum tree height; on the contrary, some searches will make a few more steps, which has a negative impact on the worst case. Given that routers can temporarily buffer packets, worst case time is not as important as the average time. The search for a BMP can only be terminated early if we have a "stop search here" ("terminal") condition stored in the node. This condition is signalled by a node being a prefix but no marker (Figure 6).

But how can we select these "most promising" layers mentioned earlier? Optimally, they would correspond to layers whose addresses are requested most — i.e. where most of the network traffic is destined. As long as only a few entries with even fewer distinct prefix lengths dominate the traffic characteristics, the solution can be found easily. However, with a large number of frequently accessed entries, building an optimal tree is a complex optimization problem, especially, because restructuring the tree also removes the terminal condition on many markers and adds it to others.

To build a useful asymmetrical tree, we can recursively split both the upper and lower part of the binary search tree's current node's search space, at a point selected by a heuristic weighting function. Two different weighting functions with different goals (one strictly picking the level covering most addresses, the other maximizing the entries while keeping the worst case bound) are shown in Figure 9, with coverage and average/worst case analysis for both weighting functions in Table 2. As can be seen, balancing gives faster increases after the second step, resulting in generally better performance than "narrow-minded" algorithms.

Now we can see why our first attempt, while improving the worst case, makes the average case worse: the prefixes with length 8, 16, and 24 are very common and also cover a big part of the address space, so they should be reached in early stages of the binary tree. In the original binary search, they were reached in step 2, 1, and 2, respectively. In the new, "optimized" approach, they were moved to step 4, 3, and 5, respectively (Figure 8, to the bottom of the tree. Besides slowing down the search, this increased the number of pure markers required to exceed the real prefixes, resulting

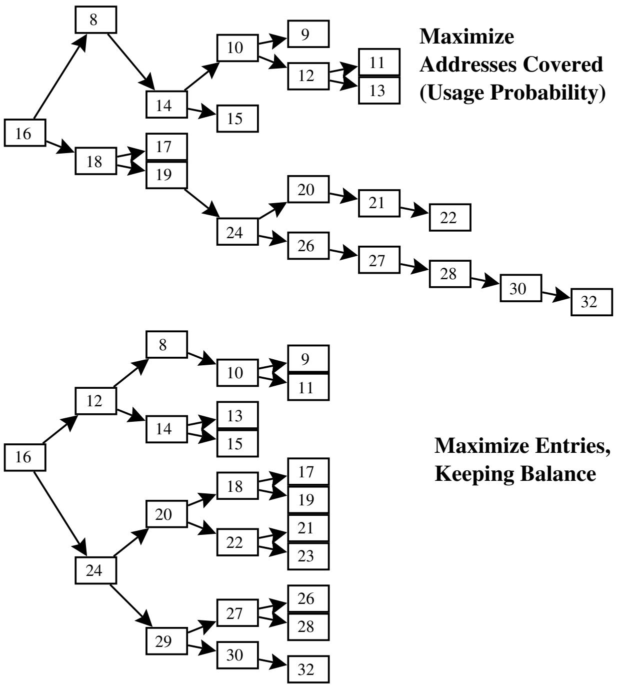



Figure 9: Asymmetric Trees produced by two Weighting Functions

| Steps | Usage | | Balance | |
|---|---|---|---|---|
| | A | E | A% | E% |
| 1 | 43% | 14% | 43% | 14% |
| 2 | 83% | 16% | 46% | 77% |
| 3 | 88% | 19% | 88% | 80% |
| 4 | 93% | 83% | 95% | 87% |
| 5 | 97% | 86% | 100% | 100% |
| Average | 2.1 | 3.9 | 2.3 | 2.4 |
| Worst case | 9 | 9 | 5 | 5 |

Table 2: Address (A) and Entry (E) Coverage for Asymmetric Trees

in a large growth in memory requirements and insertion time.

## 4.2 Mutating Binary Search

In this subsection, we further refine the basic binary search tree to change or *mutate* to more specialized binary trees each time we encounter a partial match in some hash table. We believe this a far more effective optimization than the use of asymmetrical trees though the two ideas can be combined.

In the last section, we looked at prefix distributions sorted by prefix lengths. This resulting histogram led us to propose asymmetrical binary search, which can improve average speed. Further information about prefix distributions can be extracted by dissecting the histogram: For each possible $n$ bit prefix, we could draw $2^n$ individual histograms with possibly fewer non-empty buckets, thus reducing the depth of the search tree.

When partitioning according to 16 bit prefixes[5], and counting the number of distinct prefix lengths in the partitions, we discover a

[5] There is nothing magic about the 16 bit level, other than it being a good root for a binary search of 32 bit IPv4 addresses.

| Distinct Lengths | Frequency |
|---:|---:|
| 1 | 4977 |
| 2 | 608 |
| 3 | 365 |
| 4 | 249 |
| 5 | 165 |
| 6 | 118 |
| 7 | 78 |
| 8 | 46 |
| 9 | 35 |
| 10 | 15 |
| 11 | 9 |
| 12 | 3 |

Table 3: Number of Distinct Prefix Lengths in the 16 bit Partitions (Histogram)

nice property of the routing data (Table 3). Though the whole histogram (Table 7) shows 23 distinct prefix lengths with many buckets containing a significant number of entries, none of the "sliced" histograms contain more than 12 distinct prefixes; in fact, the vast majority only contain one prefix, which often happens to be in the 16 bit prefix length hash table itself. This suggests that if we start with 16 bits in the binary search and get a match, we need only do binary search on a set of lengths that is much smaller than the 16 possible lengths we would have to search in naive binary search.

In general, every match in the binary search with some marker $X$, means that we need only search among the set of prefixes for which $X$ is a prefix. This is illustrated in Figure 10. On a match we need only search in the subtrie rooted at $X$ (rather than search the entire lower half of the trie, which is what naive binary search would do.) Thus the whole idea in mutating binary search is as follows: *whenever we get a match and move to a new subtrie, we only need to do binary search on the levels of new subtrie.* In other words, the binary search *mutates* or changes the levels on which it searches dynamically (in a way that always reduces the levels to be searched), as it gets more and more match information.

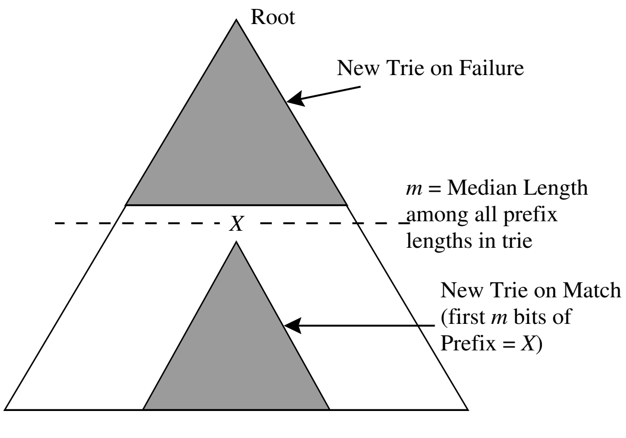



Figure 10: Showing how mutating binary search for prefix $P$ dynamically changes the trie on which it will do binary search of hash tables.

Thus each entry $E$ in the search table could contain a description of a search tree specialized for all prefixes that start with $E$. This simple optimization cuts the average search time to below two

| Steps | Usage | | Balance | |
|---:|---:|---:|---:|---:|
| | A | E | A% | E% |
| 1 | 43.9% | 14.2% | 43.9% | 14.2% |
| 2 | 98.4% | 65.5% | 97.4% | 73.5% |
| 3 | 99.5% | 84.9% | 99.1% | 93.5% |
| 4 | 99.8% | 93.6% | 99.9% | 99.5% |
| 5 | 99.9% | 97.8% | 100.0% | 100.0% |
| Average | 1.6 | 2.4 | 1.6 | 2.2 |
| Worst case | 6 | 6 | 5 | 5 |

Table 4: Address (A) and Entry (E) Coverage for Mutating Binary Search

steps (Table 4), assuming probability proportional to the covered address space. Also with other probability distributions, (i.e., according to actual measurements), we expect the average number of lookups to be around two.

As an example, consider binary search to be operating on a tree of levels starting with a root level, say 16. If we get a match which is a marker, we go "down" to the level pointed to by the down child of the current node; if we get a match which is a prefix and not a marker, we are done; finally, if we get no match, we go "up". In the basic scheme without mutation, we start with root level 16; if we get a marker match we go down to level 24, and go up to Level 8 if we get no match.

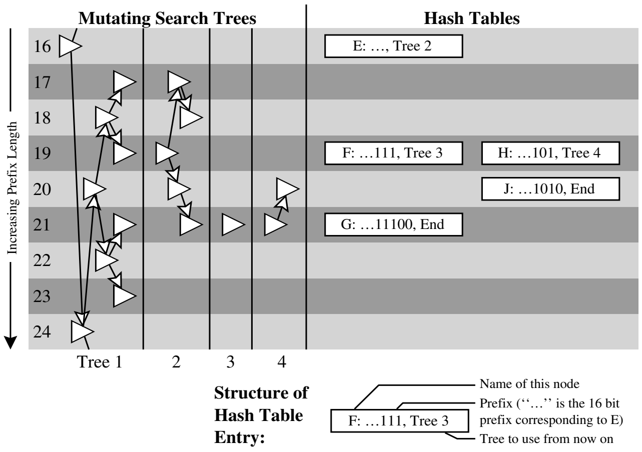



Figure 11: Mutating Binary Search Example

Doing basic binary search for an IPv4 address whose BMP has length 21 requires checking the prefix lengths 16 (hit), 24 (miss), 20 (hit), 22 (miss), and finally 21. On each hit, we go down, and on misses up.

Using Mutating Binary Search, looking for an address (see Figure 11) is different. First, we explain some new conventions for reading Figure 11. As in Figure 4, we continue to draw a trie on the right. However, in this figure, we now have multiple binary trees drawn on the left of the figure, labeled as Tree 1, Tree 2, etc. This is because the search process will move from tree to tree. Each binary tree has the root level (i.e., the first length to be searched) at the left; the upper child of each binary tree node is the length to be searched on failure, and whenever there is a match, the search switches to the more specific tree.

Finally, Figure 11 has a number of prefixes and markers that are labeled as $E, F, G, H, J$ for convenience. Every such entry in our example has $E$ as a prefix. Thus rather than describe all the bits in $E$, we denote the bits $E$ as $\ldots$; the bits in say $F$ are denoted as $\ldots 111$, which denotes the concatenation of the bits in $E$ with the suffix 111. Finally, each hash table entry consists of the name of the node, followed by the bits representing the entry, followed by the label of the binary tree to follow if we get a match on this entry. The $bmp$ values are not shown for brevity.

Consider now a search for an address whose BMP is $G$ in the database of Figure 11. The search starts with a generic tree, Tree 1, so length 16 is checked, finding $E$. Among the prefixes starting with $E$, there are known to be only five distinct lengths (say 17, 18, 19, 20, 21, and 22). So $E$ contains a description of the new tree, Tree 2, limiting the search appropriately. Using Tree 2, we find $F$, giving a new tree with only a single length, leading to $G$. The binary tree has *mutated* from the original tree of 32 lengths, to a secondary tree of 5 lengths, to a tertiary "tree" containing just a single length.

Looking for $J$ is similar. Using Tree 1, we find $E$. Switching to Tree 2, we find $H$, but after switching to Tree 4, we miss at length 21. Since a miss (no entry found) can't update a tree, we follow our current tree upwards to length 20, where we find $J$.

In general, whenever we go down in the current tree, we can potentially move to a specialized binary tree because each match in the binary search is longer than any previous matches, and hence may contain more specialized information. Mutating binary trees arise naturally in our application (unlike classical binary search) because each level in the binary search has *multiple entries* stored in a hash table. as opposed to a *single entry* in classical binary search. Each of the multiple entries can point to a more specialized binary tree.

In other words, the search is no longer walking through a single binary search tree, but through a whole network of interconnected trees. Branching decisions are not only based on the current prefix length and whether or not a match is found, but also on what the best match so far is (which in turn is based on the address we're looking for.) Thus at each branching point, you not only select which way to branch, but also change to the most optimal tree. This additional information about optimal tree branches is derived by *precomputation* based on the distribution of prefixes in the current dataset. This gives us a faster search pattern than just searching on either prefix length or address alone.

Two possible disadvantages of mutating binary search immediately present themselves. First, precomputing optimal trees can increase the time to insert a new prefix. Second, the storage required to store an optimal binary tree for each prefix appears to be enormous. We deal with insertion speed in Section 5. For now, we only observe that while routes to prefixes may frequently change in cost, the addition of a new prefix (which is the expensive case) should be much rarer. We proceed to deal with the space issue by compactly encoding the network of trees.

**Rope** A key observation is that *we only need to store the sequence of levels which binary search on a given subtrie will follow on repeated failures to find a match.* This is because when we get a successful match (see Figure 10) we move to a completely new subtrie and can get the new binary search path from the new subtrie. The sequence of levels which binary search would follow on repeated failures is what we call the Rope of a subtrie, and can be encoded efficiently. We call it Rope, because the Rope allows us to swing from tree to tree in our network of interconnected binary trees.

If we consider a trie, we define the *Rope* for the root of the trie node to be the sequence of trie levels we will consider when doing binary search on the trie levels while failing at every point. This is illustrated in Figure 12. In doing binary search we start at Level $m$ which is the median length of the trie. If we fail, we try at the quartile length (say $n$), and if we fail at $n$ we try at the one-eight level (say $o$). The sequence $m, n, o, \ldots$ is the Rope for the trie.

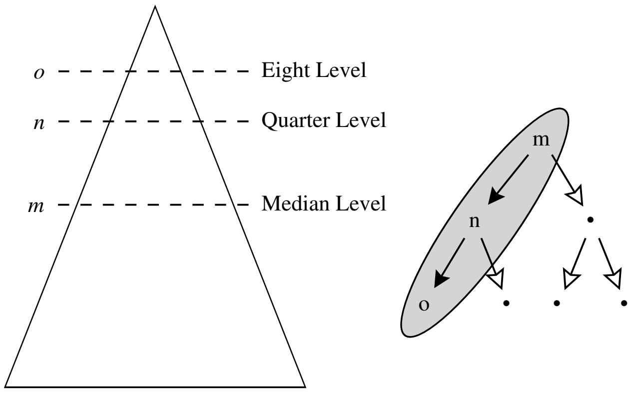
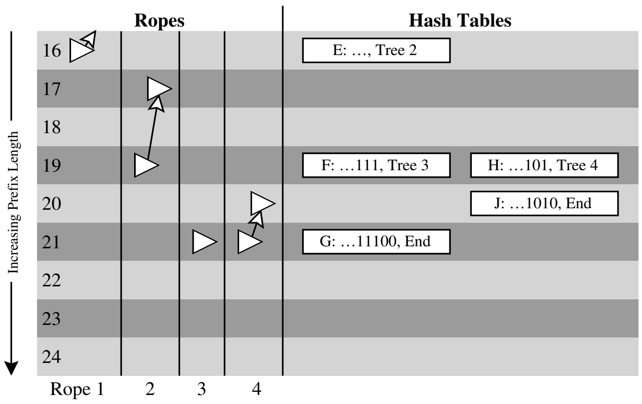



Figure 12: In terms of a trie, a rope for the trie node is the sequence of lengths starting from the median length, the quartile length, and so on, which is the same as the series of left children (see dotted oval in binary tree on right) of a perfectly balanced binary tree on the trie levels.

Figure 13 shows the Ropes containing the same information as the trees in Figure 11. Note that a Rope can be stored using only $\log_2 W$ (7 for IPv6) pointers. Since each pointer needs to only discriminate among at most $W$ possible levels, each pointer requires only $\log_2 W$ bits. For IPv6, 64 bits of Rope is more than sufficient, though it seems possible to get away with 32 bits of Rope in most practical cases. Thus a Rope is usually not longer than the storage required to store a pointer. To minimize storage in the forwarding database, a single bit can be used to decide whether the rope or only a pointer to a rope is stored in a node.




**Note:** Rope 1 contains the partially invisible layers 16, 8, 4, 2, and 1.

Figure 13: Sample Ropes

Using the Rope as the data structure has a second advantage: it simplifies the algorithm. A Rope can easily be followed, by just picking pointer after pointer in the Rope, until the next hit. Each strand in the Rope is followed in turn, until there is a hit (which starts a new Rope), or the end of the Rope is reached.

Pseudocode for the Rope variation of Mutating Binary Search is shown below. An element that is a prefix but not a marker (i.e., the "terminal" condition) specifies an empty Rope, which leads to search termination. The algorithm is initialized with a starting

Rope. The starting Rope corresponds to the default binary search tree. For example, using 32 bit IPv4 addresses, the starting Rope contains the starting level 16, followed by Levels 8, 4, 2, 1. The Levels 8, 4, 2, and 1 correspond to the "up" pointers to follow when no matches are found in the default tree. The resulting pseudocode (Figure 14) is elegant and simple to implement. It appears to be simpler than the basic algorithm.

**Function** RopeSearch($D$) (* search for address $D$ *)
Initialize Rope $R$ containing the default search sequence;
Initialize $BMP$ so far to null string;
**While** $R$ is not empty do
  Pull the first strand (pointer) off $R$ and store it in $i$;
  Extract the first $L[i].length$ bits of $D$ into $D'$;
  $M$ := Search($D'$, $L[i].hash$); (* search hash table for $D'$ *)
  **If** $M$ is not nil then
    $BMP := M.bmp$; (* update best matching prefix so far *)
    $R := M.rope$; (* get the new Rope, possibly empty *)
  **Endif**
**Endwhile**

Figure 14: Rope Search

## 4.3 Using Arrays

In cases where program complexity and memory use can be traded for speed, it might be desirable to change the first hash table lookup to a simple indexed array lookup, with the index being formed from the first $w_0$ bits of the address, with $w_0$ being the prefix length at which the search would be started. For example, if $w_0 = 16$, we would have an array for all possible $2^{16}$ values of the first 16 bits of a destination address. Each array entry for index $i$ will contain the *bmp* of $i$ as well as a Rope which will guide binary search among all prefixes that begin with $i$. An initial array lookup is not only faster than a hash lookup, but also results in reducing the average number of lookups (to around 0.5 using the current data sets we have examined.)

## 4.4 Hardware Implementations

As we have seen in both Figure 6 and Figure 14, the search functions are very simple, so ideally suited for implementation in hardware. The inner component, most likely done as a hash table in software implementations, can be implemented using (perfect) hashing hardware such as described in [Dig95]. Alternatively, a fast CAM could be used.

The outer loop in the Rope scheme can be implemented as a shift register. Using multiple shift registers, it is possible to pipeline the searches, resulting in one completed routing lookup per hash lookup time.

# 5 Implementation

Besides hashing and binary search, a predominant idea in this paper is *precomputation.* Every hash table entry has an associated *bmp* field and (possibly) a Rope field, both of which are precomputed. Precomputation allows fast search but requires more complex Insertion routines. However, as mentioned earlier, while routes to prefixes may change frequently, the addition of a new prefix (the expensive case) is much rarer. Thus it is worth paying a penalty for Insertion in return for improved search speed.

## 5.1 Basic Scheme Built from Scratch

Setting up the data structure for the Basic Scheme is straightforward, as shown in the *BuildBasic* function (Figure 15, with a complexity of $O(N \log_2 W)$). For simplicity of implementation, the list of prefixes is assumed to be sorted by increasing prefix length in advance ($O(N)$ using bucket sort). For optimal search performance, the final hash tables should ensure minimal collisions.

**Function** BuildBasic;
**For** all entries in the sorted list do
  Read next pair ($Prefix$, $Length$) from the list;
  Let $Index$ be the index for the $Length$'s hash table;
  Use Basic Algorithm on what has been built by now
    to find the BMP of $Prefix$ and store it in $BMP$;
  Add a new prefix node for $Prefix$ in the hash table for $Index$;
  (* Now insert all necessary markers "above" *)
  **For** ever do
    (* Go back one level in the binary search tree *)
    Clear the least significant one bit in $Index$;
    **If** $Index = 0$ then break; (* end reached *)
    Set $Length$ to the appropriate length for $Index$;
    Shorten $Prefix$ to $Length$ bits;
    **If** there is already an entry for $Prefix$ at $Index$ then
      Make it a marker if it isn't already;
      break; (* higher levels already do have markers *)
    **Else**
      Create a new marker $Prefix$ at $Index$' hash table;
      Set it's *bmp* field to $BMP$;
    **Endif**
  **Endfor**
**Endfor**

Figure 15: Building for the Basic Scheme

## 5.2 Rope Search from Scratch

Building a Rope Search data structure balanced for optimal search speed is more complex, since every possible binary search path needs to be optimized. To find the *bmp* values associated with markers, it helps to have an auxiliary trie. Thus we have two passes:

Pass 1 builds a conventional trie. Each trie node contains a list of all prefix lengths used by its "child" nodes (subtree length set, SLS). If a weighting function is being used to optimize accesses based on known or assumed access patterns, further statistics and forecasts should be summarized. All this additional information is kept up-to-date while inserting, in $O(NW)$ time.

In the second pass, all prefixes are inserted into the hash tables, starting with the shortest prefix: for each prefix, it's Rope and the BMP for it's markers are calculated and then the markers and the prefix are inserted. This takes $O(N \log^2 W)$, as we will see later.

Inserting from shortest to longest prefix has the nice property that all BMPs for the newly inserted markers are identical and thus only need to be calculated once. This can easily be seen by recalling that each marker is (1) a prefix of all the entries it guides the search to, (2) that the marker's BMP is also a prefix of the marker, and (3) inserting entries longer than the marker's length cannot change it's BMP.

There are at most $O(\log W)$ markers to insert for each real prefix, and each prefix and marker needs a rope, which can be calculated from the SLS in $O(\log W)$.[6] The overall work thus is

[6] using *find first bit* instructions; precomputed arrays would be $O(1)$

| Algorithm | Build | Search | Memory |
|---|---|---|---|
| Binary Search | $O(N \log N)$ | $O(\log(2N))$ | $O(N)$ |
| Trie | $O(N\,W)$ | $O(W)$ | $O(N\,W)$ |
| Radix Trie[7] | $O(N\,W)$ | $O(W)$ | $O(N)$ |
| Basic Scheme | $O(N \log W)$ | $O(\log W)$ | $O(N \log W)$ |
| Asymmetric BS | $O(N \log W)$ | $O(\log W)$ | $O(N \log W)$ |
| Rope Search | $O(N\,W)$ | $O(\log W)$ | $O(N \log W)$ |
| Ternary CAMs | $O(N)$ | $O(1)$[8] | $O(N)$ |

Table 5: Speed and Memory Usage Complexity

$O(N \max(W, \log^2(W)))$. We are working on a faster and more elegant algorithm for building the Rope Search data structure in time $O(N \log^2(W))$ (that also does not require building a trie). We will describe this and other optimizations in a future paper.

One problem for the insertion is that the number of markers for each length is not known in advance, which makes it difficult to allocate memory for the hash table in advance. This problem can be avoided by putting all entries in a *single* hash table and including the prefix length in the hash calculation. Since there is an upper limit of log $W$ markers per real prefix. we can size the single hash table. For typical IPv4 forwarding tables, about half of this maximum number is being used.

## 5.3 Insertions and Deletions

Adding and removing single entries from the tree can also be done, but since no rebalancing occurs, the performance of the lookups might slowly degrade over time. However, addition and deletion are not trivial. Adding or deleting a single prefix can change the $bmp$ values of a large number of markers, and thus insertion is potentially expensive in the worst case. Similarly, adding or deleting a new prefix that causes a new prefix length to be added or deleted can cause the Ropes of a number of entries to change. The simplest solution is to batch a number of changes and do a complete build of the search structure. Such solutions will have adequate throughput (because whenever the build process falls behind, we will batch more efficiently), but have poor latency. We are working on fast incremental insertion and deletion algorithms, but we do not describe them here for want of space. Our incremental insertion and deletion algorithms still require the tree to be rebuilt after a large number of different inserts and deletes.

# 6 Performance Evaluation

Recollecting some of the data mentioned earlier, we show measured and expected performance for our scheme.

## 6.1 Complexity Comparison

Table 5 collects the (worst case) complexity necessary for the different schemes mentioned here. Be aware that these complexity numbers do not say anything about the absolute speed or memory usage. See Section 2 for a comparison between the schemes. For Radix Tries, Basic Scheme, Asymmetric Binary Search, and Rope Search, $W$ is the number of distinct lengths. Memory complexity is given in $W$ bit words.

| | | Basic | Rope | Array | Radix |
|---|---|---|---|---|---|
| Memory usage | [MB] | 1.4 | 1.4 | 1.2 | 1.2 |
| “Primary” Memory | [MB] | 0.6 | 0.6 | 0.3 | |
| First step(s) (cached) | [ns] | 40 | 40 | 15 | |
| Later steps (not cached) | [ns] | 100 | 110 | 110 | |
| Average lookup | [ns] | 180 | 100 | 80 | 1400 |
| Worst case lookup[9] | [ns] | 850 | 600 | 450 | 2000 |

Table 6: Performance Comparison

## 6.2 Measurements for IPv4

So far we have described how long our algorithm takes (in the average or worst case) in terms of the number of hash computations required. It remains to quantify the time taken for a computation on an arbitrary prefix length using software. To do so, we ran the following experiments on a 200 MHz Pentium Pro from C code using the compiler's maximum optimization (Table 6). The forwarding table was the same 33,000 entry forwarding table [Mer96] used before.

**Basic Scheme** Memory usage is close to 1.2 MByte, for the primary data structures (the most commonly accessed hash tables for length 8, 16, and 24) fit mostly into second level cache, so the first two steps (which is the average number needed) are very likely to be found in the cache. Later steps, seldom needed, will be remarkably slower.

**Rope Search** Although the average number of search levels and thus the number of marker entries needed decreases, the memory needed per node increases.

**Rope Search starting with Array Lookup** This array fully fits into the cache, leaving ample space for the hash tables. The array lookup is much quicker, and there will be less total lookups needed than for the Rope scheme.

**Radix Tries** The Radix Trie functions were extracted from optimized NetBSD kernel code and put into user space for measurement.

## 6.3 Projections for IP Version 6

IPv6 address assignment principles have not been finally decided upon. However, three methods are currently being discussed in the IPng working group of the Internet Engineering Task Force (IETF). All of them use hierarchical schemes to provide as much routing aggregation as possible: provider-based addressing [R$^+$97], geographical addressing, and “GSE” (Global, Site, End-system) [O'D97].

All these schemes help to reduce routing information. In the optimal case of a strictly hierarchical environment, it can go down to a handful of entries. But with massive growth of the Internet together with the increasing forces for connectivity to multiple ISPs (“multihoming”) and meshing between the ISPs, we expect the routing tables to grow. Another new feature of IPv6, Anycast addresses [HD96, DH96], may (depending on how popular they will become) add a very large number of host routes and other routes with very long prefixes.

So most sites will still have to cope with a large number of routing entries at different prefix lengths. There is likely to be more distinct prefix lengths, so the improvements achieved by binary search will be similar or better than those achieved on IPv4.

For the array access improvement shown in Section 4.3, the improvement may not be as dramatic as for IPv4. Although it will improve performance for IPv6, it is less attractive, because addresses will be longer. Good starting points may require rather large prefixes (i.e. 24 bits or longer). With $2^{24}$ necessary entries, it is no longer feasible to have a whole array stored in memory, requiring us to select a less optimal starting point to still gain improvement from the array access. Depending on the actual data, this may still be a win. All other optimizations are expected to yield similar improvements.

## 7 Conclusions and Future Work

We have designed a new algorithm for best matching search. The best matching prefix problem has been around for twenty years in theoretical computer science; to the best of our knowledge, the best theoretical algorithms are based on tries. While inefficient algorithms based on hashing [Skl93] were known, we have discovered an extremely efficient algorithm that scales with the logarithm of the address size and so is very close to the theoretical limit of $O(\log \log N)$.

Our algorithm contains both intellectual and practical contributions. On the intellectual side, after the basic notion of binary searching on hash tables, we found that we had to add markers and use precomputation, to ensure logarithmic time in the worst-case. Algorithms that only use binary search of hash tables are unlikely to provide logarithmic time in the worst case. Among our optimizations, we single out mutating binary trees as an aesthetically pleasing idea that leverages off the extra structure inherent in our particular form of binary search.

On the practical side, we have a fast, scalable solution for IP lookups that can be implemented in either software or hardware. Our software projections for IPv4 are 80 ns and we expect 150–200 ns for IPv6. Our average case speed projections are based on the structure of existing routing databases that we examined. We expect most of the characteristics of this address structure to strengthen in the future, especially with the transition to IPv6. Even if our predictions, based on the little evidence available today, should prove to be wrong, the overall performance can easily be restricted to that of the basic algorithm which already performs well.

With algorithms such as ours, we believe that there is no more reason for router throughputs to be limited by the speed of their lookup engine. We also do not believe that hardware lookup engines are required because our algorithm can be implemented in software and still perform well. For similar reasons, we do not believe that there is a compelling need for protocol changes to avoid lookups as proposed in Tag and IP Switching. Even if these protocol changes were accepted, fast lookup algorithms such as ours are likely to be needed at several places in the network.

Future work on our algorithm includes theoretical work on a choice of balancing function, hopefully yielding an improvement over our ad-hoc heuristic functions. Other avenues of research include the choice of a heuristic function based on actual network traffic, and work on faster insertion algorithms. We are also trying to optimize the building and modification processes. Our algorithm belongs to a class of algorithms that speed up search at the expense of insertion; we are looking for other applications and generalizations of our algorithm.

In spite of potential improvements, we believe our algorithm is ready for practical use. To prove this, it will be incorporated into the Crossbow project [D+97], a joint project between ETH and Washington University. The goal of Crossbow is to build a extensible framework for IPv6 as well as a high-speed IPv6 cell-switched router with QoS guarantees.

## Acknowledgements

We would like to thank the whole Crossbow team for their valuable input, feedback, and support, especially Hari Adiseshu, Dan Decasper, Zubin Dittia, and Guru Parulkar. We also thank Cheenu (V. Srinivasan) for providing us with the Radix Trie code and suggesting fast hash functions for the Pentium. The work of George Varghese was supported in part by an ONR Young Investigator Award and NSF grants NCR-940997 and NCR-9628218.

## References

[CV95] Girish Chandranmenon and George Varghese. Trading packet headers for packet processing. In *Proceedings of SIGCOMM 95*, Boston, August 1995.

[CV96] Girish Chandranmenon and George Varghese. Trading packet headers for packet processing. *IEEE Transactions on Networking*, April 1996.

[D+97] Dan Decasper et al. Crossbow — a toolkit for integrated services over cell switched IPv6. In *Proceedings of the IEEE ATM'97 workshop*, Lisboa, Portugal, May 1997.

[DH96] Steven Deering and Robert Hinden. Internet protocol, version 6 (IPv6) specification (RFC1883). ftp://ds.internic.net/rfc/rfc1883.txt, 1996.

[Dig95] Digital. GIGAswitch/FDDI networking switch. http://www.networks.europe.digital.com/html/products_guide/hp-swch3.html, 1995.

[F+93] Vince Fuller et al. Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy (RFC1519). ftp://ds.internic.net/rfc/rfc1519.txt, 1993.

[HD96] Robert Hinden and Steven Deering. IP version 6 addressing architecture (RFC1884). ftp://ds.internic.net/rfc/rfc1884.txt, 1996.

[Lab96] Craig Labovitz. Routing analysis. http://www.merit.edu/ipma/analysis/routing.html, 1996.

[Mer96] Merit Network, Inc. 12/19/96 routing table snapshot at Mae-East NAP. http://www.merit.edu/ipma/routing_table/, January 1996.

[MF93] A. McAuley and P. Francis. Fast routing table lookup using CAMs. In *Proceedings of INFOCOM*, pages 1382–1391, March-April 1993.

[MTW95] Anthony J. McAuley, Paul F. Tsuchiya, and Daniel V. Wilson. Fast multilevel hierarchical routing table using content-addressable memory. U.S. Patent serial number 034444. Assignee Bell Communications research Inc Livingston NJ, January 1995.

[NMH97] Peter Newman, Greg Minshall, and Larry Huston. IP Switching and gigabit routers. *IEEE Communications Magazine*, January 1997.

[O'D97] Mike O'Dell. GSE - an alternate addressing architecture for IPv6. ftp://ds.internic.net/internet-drafts/draft-ietf-ipngwg-gseaddr-00.txt, 1997.

[Par96] Craig Partridge. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis*, San Diego, CA, USA, February 1996.

[Per92] Radia Perlman. *Interconnections, Bridges and Routers*. Addison-Wesley, 1992.

[R+96] Yakov Rekhter et al. Tag switching architecture overview. ftp://ds.internic.net/internet-drafts/draft-rfced-info-rekhter-00.txt, 1996.

[R+97] Yakov Rekhter et al. An IPv6 provider-based unicast address format (RFC2073). ftp://ds.internic.net/rfc/rfc2073.txt, 1997.

[Rob97] Erica Roberts. IP on speed. *Data Communications Magazine*, pages 84–96, March 1997.

[Skl93] Keith Sklower. A tree-based routing table for Berkeley Unix. Technical report, University of California, Berkeley, 1993.