Scalable High-Speed Prefix Matching

Marcel Waldvogel
Washington University in St. Louis
and
George Varghese
University of California, San Diego
and
Jon Turner
Washington University in St. Louis
and
Bernhard Plattner
ETH Zürich

Finding the longest matching prefix from a database of keywords is an old problem with a number of applications, ranging from dictionary searches to advanced memory management to computational geometry. But perhaps today's most frequent best matching prefix lookups occur in the Internet, when forwarding packets from router to router. Internet traffic volume and link speeds are rapidly increasing; at the same time, an increasing user population is increasing the size of routing tables against which packets must be matched. Both factors make router prefix matching extremely performance critical.

In this paper, we introduce a taxonomy for prefix matching technologies, which we use as a basis for describing, categorizing, and comparing existing approaches. We then present in detail a fast scheme using binary search over hash tables, which is especially suited for matching long addresses, such as the 128 bit addresses proposed for use in the next generation Internet Protocol, IPv6. We also present optimizations that exploit the structure of existing databases to further improve access time and reduce storage space.

Categories and Subject Descriptors: C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; E.2 [**Data Storage Representations**]: Hash-table representations; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: collision resolution, forwarding lookups, high-speed networking

The work of Marcel Waldvogel was supported in part by KTI grant 3221.1. The work of George Varghese was supported in part by an ONR Young Investigator Award and NSF grants NCR-940997 and NCR-9628218.
Parts of this paper were presented in ACM SIGCOMM '97 [Waldvogel et al. 1997].

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 growing explosively. Not surprisingly, network traffic is doubling every few months. The proliferation of multimedia networking applications (e.g., Napster) and devices (e.g., IP phones) is expected to give traffic another major boost.

The increasing traffic demand requires four key factors to keep pace if the Internet is to continue to provide good service: link speeds, router data throughput, packet forwarding rates, and quick adaptation to routing changes. Readily available solutions exist for the first two factors: for example, fiber-optic cables can provide faster links and switching technology can be used to move packets from the input interface of a router to the corresponding output interface at multi-gigabit speeds [Partridge et al. 1998]. Our paper deals with the other two factors: forwarding packets at high speeds while still allowing for frequent updates to the routing table.

A 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 [Spinney 1995] 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 Addressable Memories (CAM) cannot directly be used for Internet address lookups. Also, the most widely used algorithm for IP lookups, BSD Patricia Tries [Sklower 1993], has poor performance.

Prefix matching in Internet routers was introduced in the early 1990s, when it was foreseen that the number of endpoints and the amount of routing information would grow enormously. At that time, only address classes A, B, and C existed, giving individual sites either 24, 16, and 8 bits of address space, allowing up to 16 Million, 65,534, and 254 host addresses, respectively. The size of the network could easily be deduced from the first few address bits, making hashing a popular technique. The limited granularity turned out to be extremely wasteful on 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 would have resulted in massive growth of routing table entries over time. Therefore, Classless Inter-Domain Routing (CIDR) [Fuller et al. 1993] was introduced, which allowed for aggregation of networks in arbitrary powers of two to reduce routing table entries. With this aggregation, it was no longer possible to identify the number of bits relevant for the forwarding decision from the address itself, but required a prefix match, where the number of relevant bits was only known when the matching entry had already been found in the database.

To achieve maximum routing table space reduction, aggregation is done aggressively. Suppose all the subnets in a big network have identical routing information except for a single, small subnet with different information. Instead of having multiple routing entries for each subnet in the large network, just two entries are needed: one for the overall network, and one entry showing the exception for the small subnet. Now there are two matches for packets addressed to the exceptional subnet. Clearly, the exception entry should get preference there. This is achieved by preferring the more specific entry, resulting in a *Best*

Matching Prefix (BMP) operation. In summary, CIDR traded off better usage of the limited IP address space and a reduction in routing information for a more complex lookup scheme.

The upshot is that 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, directed to the next router or the destination host. For example, a forwarding database may have the prefixes $P_1 = 0000*$, $P_2 = 0000\_111*$ and $P_3 = 0000\_1111\_0000*$, with $*$ meaning all further bits are unspecified. An address whose first 12 bits are 0000_0110_1111 has longest matching prefix P_1. On the other hand, an address whose first 12 bits are 0000_1111_0000 has longest matching prefix P_3.

The use of best matching prefix in forwarding has allowed IP routers to accommodate various levels of address hierarchies, and has allowed parts of the network to be oblivious of details in other parts. 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" [Sklower 1993].

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 of $\log W$ hash lookups, 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 longer for current algorithms (see Section 2), giving an *order of magnitude performance improvement*. Using perfect hashing [Fredman et al. 1984], we can lookup 128 bit IP addresses in at most 7 memory accesses. This is significant because on current processors, the calculation of a hash function is usually much cheaper than an off-chip memory access.

In addition, we use several optimizations to significantly reduce the average number of hashes needed. For example, our analysis of the largest IPv4 forwarding tables from Internet backbone routers show that the majority of addresses can be found with at most two hashes. Also, all available databases allowed us to reduce the worst case to four accesses. In both cases, the first hash can be replaced by a simple index table lookup.

The rest of the paper is organized as follows. Section 2 introduces our taxonomy and compares 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 focuses on a series of important optimizations to the basic scheme that improve average performance. Section 5 describes ways how to build the appropriate structures and perform dynamic insertions and deletions, Section 6 introduces prefix partitioning to improve worst-case insertion and deletion time, and Section 7 explains fast hashing techniques. Section 8 describes performance measurements using our scheme for IPv4 addresses, and performance projections for IPv6 addresses. We conclude in Section 9 by assessing the theoretical and practical contributions of this paper.

2. COMPARISON OF EXISTING ALGORITHMS

As several algorithms for efficient prefix matching lookups have appeared in the literature over the last few years (including a recent paper [Srinivasan and Varghese 1999] in ACM TOCS), we feel that it is necessary to structure the presentation of related work using a taxonomy. Our classification goes beyond the lookup taxonomy recently introduced in [Ruiz-Sánchez et al. 2001]. However, the paper [Ruiz-Sánchez et al. 2001] should be consulted for a more in-depth discussion and comparison of some of the other popular schemes.

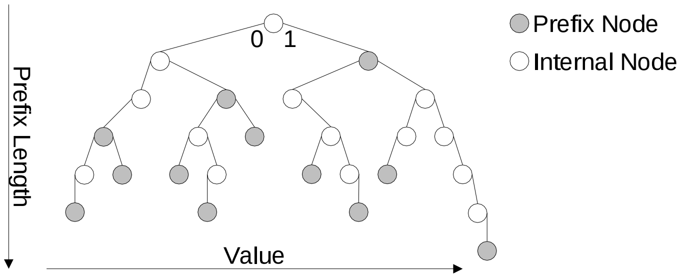


Fig. 1. Prefix Matching Overview

Traditionally, prefix matching has been done on tries [Gwehenberger 1968; Morrison 1968], with bit-wise (binary) tries being the foremost representative. Figure 1 shows such a trie. To find the longest prefix matching a given search string, the tree is traversed starting at the root (topmost) node. Depending on the value of the next bit in the search string, either the left or right link is followed, always remembering the most recent prefix node visited. When the search string is exhausted or a nonexistent link is selected, the remembered prefix node is returned as the best match.

Thus a trie has two aspects (Figure 1) that we base our taxonomy on: the first is the vertical aspect that signifies prefix length (as we travel vertically down the trie the prefixes we encounter are correspondingly longer); the second horizontal aspect is the prefix value (the value of the bit string representing the prefix, prefixes of the same length are sorted from left to right). Our simple insight, which is the basis of our taxonomy, is that *existing schemes either do linear or binary search in either the prefix length or value dimensions.* The schemes can also be augmented using parallelism, caching, and compression.

2.1 Taxonomy

Thus our taxonomy is organized along four dimensions. The two major dimensions are defined by the main search space in which to operate (see Figure 1) and the basic search algorithm used. The minor dimensions, orthogonal and largely independent of the main dimensions, identify parallelism, memory optimizations and compression, and the use of caching.

Search space: Search in prefix length or value space

Search algorithm: Linear or binary search

Parallelism: Serialized, pipelined, or parallel execution

Data Compaction and caching: Optional use of compression and caching.

2.2 Linear Search on Prefix Lengths

The basic trie scheme described above is an example of linear search in the prefix length space without compression. This is because trie traversal explores prefixes in increasing order of lengths. Many schemes have extended this idea by reducing the trie memory footprint or the number of trie nodes accessed during search.

The most commonly available IP lookup implementation is found in the BSD Unix kernel, and is a radix trie implementation [Sklower 1993]. It uses a *path-compressed trie*, where non-branching internal nodes are eliminated, improving memory utilization. The actual implementation uses potentially expensive backtracking. Even an efficient search implementation would require $O(W)$ node accesses, where W is the length of an address. Thus, search implementation requires up to 32 or 128 costly external memory accesses, for IPv4 or IPv6, respectively. Therefore, these algorithms are not directly used in high-speed networking equipment. Unlike most other algorithms, updates to these unibit tries are very fast and make them ideal candidates for data structures with a high update/search ratio.

Path compression is most useful when compressing long non-branching chains of internal nodes, which occur in sparsely populated areas of the trie. LC-Tries [Andersson and Nilsson 1994; Nilsson and Karlsson 1999] extend this notion by introducing level compression, where, for any given prefix length, dense areas with a common ancestor are aggregated into a single 2^k-ary branching node. This scheme maintains a good balance of memory usage, search speed, and update times.

For applications where search speed is much more important than update speed or worst-case memory consumption, such as for Internet forwarding lookups, more aggressive search time optimization is required. To reduce the number of levels that need to be touched, *Controlled Prefix Expansion* [Srinivasan and Varghese 1999] selects a small number of prefix lengths to be searched. All database entries that are not already of one of these lengths, are expanded into multiple entries of the next higher selected length. Depending on the length of the "strides" s between the selected lengths and the prefix length distribution, this can lead to an expansion of up to 2^{s-1}. Selecting the strides using dynamic programming techniques results in minimal expansion when used with current IP routing tables. Despite expansion, this search scheme is still *linear* in the prefix length because expansion only provides a constant factor improvement.

Prefix expansion is used generously in the scheme developed by Gupta et al. [Gupta et al. 1998] to reduce memory accesses even further. In the DIR-24-8 scheme presented there, all prefixes are expanded to at least 24 bits (the Internet backbone forwarding tables contain almost no prefixes longer than 24 bits). A typical lookup will then just use the most significant 24 bits of the address as an index into the 16M entries of the table, reducing the expected number of memory accesses to almost one.

A different approach was chosen by Degermark et al. [Degermark et al. 1997]. By first expanding to a complete trie and then using bit vectors and mapping tables, they are able to represent routing tables of up to 40,000 entries in around 150KBytes. This compact representation allows the data to be kept in on-chip caches, which provide much better performance than standard off-chip memory. A further approach to trie compression using bitmaps is described in [Eatherton 1999].

Crescenzi et al. [Crescenzi et al. 1999] present another compressed trie lookup scheme. They first fully expand the trie, so that all leaf nodes are at length W. Then, they divide the tree into multiple subtrees of identical size. These slices are then put side-by-side, say, in columns. All the neighboring identical rows are then collapsed, and a single table is created to map from the original row number to the new, compressed row number. Unlike the previous approach [Degermark et al. 1997], this does not result in a small enough table to fit into typical on-chip caches, yet it guarantees that all lookups can be done in exactly 3 indexed memory lookups.

McAuley and Francis [McAuley and Francis 1993] use standard (“binary”) content-addressable memories (CAMs) to quickly search the different prefix lengths. The first solution discussed requires multiple passes through, starting with the longest prefix. This search order was chosen to be able to terminate after the first match. The other solution is to have multiple CAMs queried in parallel. CAMs are generally much slower than conventional memory and do not provide enough entries for backbone routers are still rare, where in the near future more than 100,000 forwarding entries will be required. Nevertheless, CAMs are popular in edge routers, which typically only have up to hundreds of forwarding entries.

2.3 Binary Search on Prefix Lengths

The prior work closest to binary search on prefix lengths occurs in computational geometry. De Berg et al. [de Berg et al. 1995] describe a scheme for one-dimensional point location based on *stratified trees* [van Emde Boas 1975; van Emde Boas et al. 1977]. A stratified tree is probably best described as a self-similar tree, where each node internally has the same structure as the overall tree. The actual search is not performed on a prefix trie, but on a balanced interval tree. The scheme does not support overlapping regions, which are required to implement prefix lookups. While this could be resolved in a preprocessing step, it would degrade the incremental update time to $O(N)$. Also unlike the algorithm introduced in Section 3, it cannot take advantage of additional structure in the routing table (Section 4).

2.4 Linear Search of Values

Pure linear value search is only reasonable for very small tables. But a hardware-parallel version using ternary CAMs has become attractive in the recent years. Ternary CAMs, unlike the binary CAMs above, which require multiple stages or multiple CAMs, have a mask associated with every entry. This mask is used to describe which bits of the entry should be compared to the query key, allowing for one-pass prefix matching. Due to the higher per-entry hardware overhead, ternary CAMs typically provide for only about half the entries as comparable binary CAMs. Also, as multiple entries may match for a single search key, it becomes necessary to prioritize entries. As priorities are typically associated with an internal memory address, inserting a new entry can potentially cause a large number of other entries to be shifted around. Shah and Gupta [Shah and Gupta 2000] present an algorithmic solution to minimize these shifts while Kobayashi et al. [Kobayashi et al. 2000] modify the CAM itself to return only the longest match with little hardware overhead.

2.5 Binary Search of Values

The use of binary search on the value space was originally proposed by Butler Lampson and described in [Perlman 1992]; additional improvements were proposed in [Lampson et al. 1998]. The key ideas are to represent each prefix as a range using two values (the lowest and highest values in the range), to preprocess the table to associate matching prefixes with these values, and then to do ordinary binary search on these values. The resulting search time is $\lceil \log_2 2N \rceil$ search steps, with N being the number of routing table entries. With current routing table sizes, this gets close to the expected number of memory accesses for unibit tries, which is fairly slow. However, lookup time can be reduced using B-trees instead of binary trees and by using an initial memory lookup [Lampson et al. 1998].

2.6 Parallelism, Data Compaction, and Caches

The minor dimensions described above in our taxonomy can be applied to all the major schemes. Almost every lookup algorithm can be pipelined. Also, almost all algorithms lend themselves to more compressed representations of their data structures; however, in [Degermark et al. 1997; Crescenzi et al. 1999; Eatherton 1999], the main novelty is the manner in which a multibit trie is compressed while retaining fast lookup times.

In addition, all of the lookup schemes can take advantage of an added lookup cache, which does not store the prefixes matched, but instead stores recent lookup keys, as exact matches are generally much simpler and faster to implement. Unfortunately, with the growth of the Internet, access locality in packet streams seems to decrease, requiring larger and larger caches to achieve similar hit rates. In 1987, Feldmeier [Feldmeier 1988] found that a cache for the most recent 9 destination addresses already provided for a 90% hit rate. 8 years later, Partridge [Partridge 1996] did a similar study, where caches with close to 5000 entries were required to achieve the same hit rate. We expect this trend to continue and potentially to become even more pronounced.

2.7 Protocol Based Solutions

Finally, (leaving behind our taxonomy) we note that one way to finesse 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 were IP Switching [Newman et al. 1997] and Tag Switching [Rekhter et al. 1997], both now mostly replaced by Multi-Protocol Label Switching (MPLS [Rosen et al. 2001]. All three 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, a technique first described by Chandranmenon and Varghese [Chandranmenon and Varghese 1995]. This switching information is included by adding a "label" to each packet, a small integer that allows direct lookup in the router's forwarding table.

Neither scheme can completely avoid ordinary IP lookups. All schemes require the ingress router (to the portions of the network implementing their protocol) 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. Labels will become scarce resources, of which only a finite amount exist. Thus towards the backbone, they need to be aggregated; away from the backbone, they need to be separated again.

2.8 Summary of Existing Work

There are two basic solutions for the prefix matching problem caused by Internet growth: (1) making lookups faster or (2) reducing the number of lookups using caching or protocol modifications. As seen above, the latter mechanisms are not able to completely avoid lookups, but only reduce them to either fewer routers (label switching) or fewer per router (caching). The advantage of using caches will disappear in a few years, as Internet data rates are growing much faster than hardware speeds, to the point that all lookup memory will have to use the fastest available memory (i.e., SRAM of the kind that is currently used by cache memory).

The most popularly deployed schemes today are based on linear search of prefix lengths using multibit or unibit tries together with high speed memories and pipelining. However, these algorithms do not scale well to longer next generation IP addresses. Lookup schemes based on unibit tries and binary search are (currently) too slow and do not scale well; CAM solutions are relatively expensive and are hard to field upgrade;

In summary, all existing schemes have problems of either performance, scalability, generality, or cost, especially when addresses extend beyond the current 32 bits. We now describe a lookup scheme that has good performance, is scalable to large addresses, and does not require protocol changes. Our scheme allows a cheap, fast software implementation, and is also amenable to hardware implementations.

3. BASIC BINARY SEARCH SCHEME

Our basic algorithm is based on three significant ideas: 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 pre-computation 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 3.4 to arrive at a working basic scheme. We describe further optimizations to the basic scheme in the next section. As there are multiple ways to look at the data structure, whenever possible we will use the terms “shorter” and “longer” to signify selecting shorter or longer prefixes.

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.

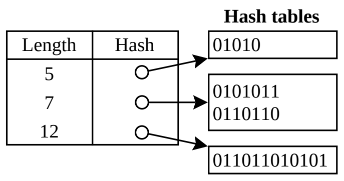


Fig. 2. Hash Tables for each possible prefix length

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. 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 2). 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 destination 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 the longest match and thus our BMP; 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.

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 $O(W)$ 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 $O(\log W_{dist})$. However, for binary search to make its branching decision, it requires the result of an ordered comparison, returning whether the probed entry is "less than," "equal," or "greater than" our search key. As we are dealing with prefix lengths, these map to indications to look at "shorter," "same length," or "longer," respectively. When dealing with hash lookups, ordered comparison does seem impossible: either there is a hit (then the entry found equals the hash key) or there is a miss and thus no comparison possible.

Let's look at the problem from the other side: In ordinary binary search, "equal" indicates that we have found the matching entry and can terminate the search. When searching among prefix lengths, having found a matching entry does not yet imply that this is also the best entry. So clearly, when we have found a match, we need to continue searching among the longer prefixes. How does this observation help? It signifies, that when an entry has been found, we should remember it as a potential candidate solution, but continue looking for longer prefixes. The only other information that we can get from the hash lookup is a miss. Due to limited choice, we start taking hash misses as an indication to inspect shorter prefixes. This results in the pseudo code given in Figure 3.

```
**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 most significant $L[i].\mathrm{length}$ bits of $D$ into $D'$;
    Search($D'$, $L[i].\mathrm{hash}$); (* search hash table for $D'$ *)
    **If** found then set $R$ := longer half of $R$ (*longer prefixes*)
        **Else** set $R$ := shorter half of $R$; (*shorter prefixes*)
    **Endif**
**Endwhile**
```

Fig. 3. Naïve Binary Search

Figure 4 illustrates binary search over 7 prefix lengths. The tree on the top indicates the binary search branching that is to be taken: Starting at the root (length 4), the current hash table is probed for the key shortened to the current prefix length. If the key is found, longer prefixes are selected, otherwise shorter prefixes are tested next. As an example, we try to find the longest prefix for "1100100." We find a match at length 4 (1100*), thus taking

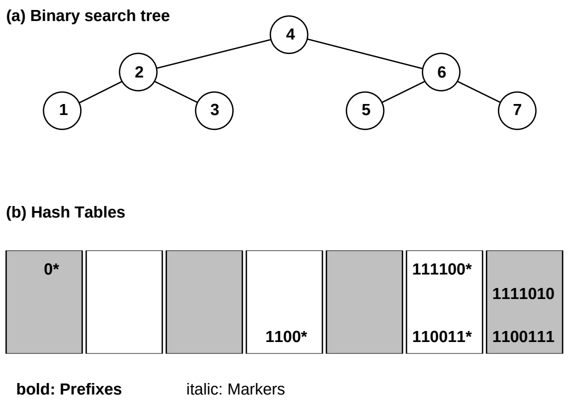


Fig. 4. Binary Search: First Attempt

the branch towards longer prefixes, namely length 6. Looking for “110010*” there fails. Therefore, we look for shorter prefixes at length 5, and miss again. The best match found during our search is “1100*,” which is correct.

Trying to locate address “1111000” fails miserably: We miss at 4, go shorter to 2, miss again, and have no luck at length 1 either. The correct match would have been “111100*” at length 6. Unlike the previous example, there had been no guiding prefixes in this case. To make sure that such guiding prefixes exist, we insert additional branching information, called *markers*. These markers look like prefixes, except that they have no associated information fields, their sheer presence is all we want for now.

But where do we need markers, and how many are there? Naïvely, it seems that for every entry, there would be a marker at all other prefix lengths, leading to a massive increase in the size of the hash tables. Luckily, markers do not need to be placed at all levels. Figure 5 again shows a binary search tree. At each node, a branching decision is made, going to either the shorter or longer subtree, until the correct entry or a leaf node is met. Clearly, at most $\log W$ internal nodes will be traversed on any search, resulting in at most $\log W$ branching decisions. Also, any search that will end up at a given node only has a single path to choose from, eliminating the need to place markers at any other levels.

3.3 Problems with Backtracking

Unfortunately, the algorithm shown in Figure 3 is not correct as it stands and *does not* take logarithmic time if fixed naïvely. The problem is that while markers are good things (they lead to potentially better, longer prefixes in the table), 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 shorter prefixes of R again. Such a naïve modification can lead us back to linear time search. An example will clarify this.

First consider the prefixes $P_1 = 1$, $P_2 = 00$, $P_3 = 111$ (Figure 6). 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 P_3). 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., P_1. The marker indicating that there will be longer prefixes, indispensable to find P_3, was misleading in this case; so

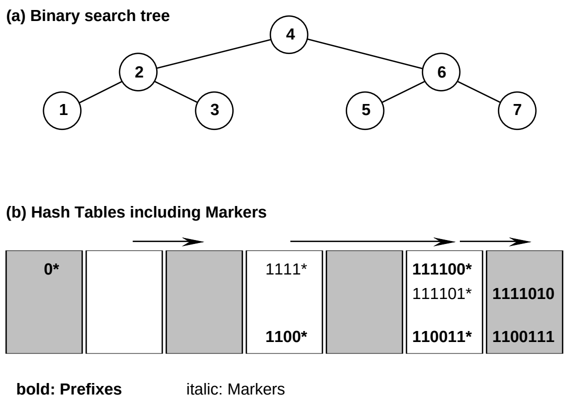


Fig. 5. Improved Branching Decisions due to Markers

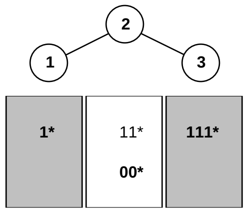


Fig. 6. Misleading Markers

apparently, we have to go back and search the shorter 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 in the longer section.)

3.4 Pre-computation to Avoid Backtracking

We use pre-computation to avoid backtracking when we shrink the current range R to the longer 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.[1] *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 longer half, but we also *remember* the value of *M.bmp* as the current best matching prefix. Now if the longer half of R fails to produce anything interesting, we need not

[1]This can either be a pointer to the best matching node, or a copy of its value. The latter is typically preferred, as the information stored is often comparable to the size of a pointer. Very often, the BMP is an index into a next-hop table.

backtrack, because the results of the backtracking are already summarized in the value of *M.bmp*. The new code is shown in Figure 7.

```
**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 $R$ := shorter half of $R$; (* not found *)
    Else-if $M$ 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 *)
        $R$ := longer half of $R$;
    Endif
Endwhile
```

Fig. 7. Working 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 to the problem to reduce the average number of hash computations.

4.1 Asymmetric Binary Search

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. Figure 8 shows the prefix length distribution extracted from forwarding table snapshots from five major backbone sites in January 1999 and, for comparison, at Mae-East in December 1996 [2]. As can be seen, the entries are distributed over the different prefix lengths in an extremely uneven fashion. The peak at length 24

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

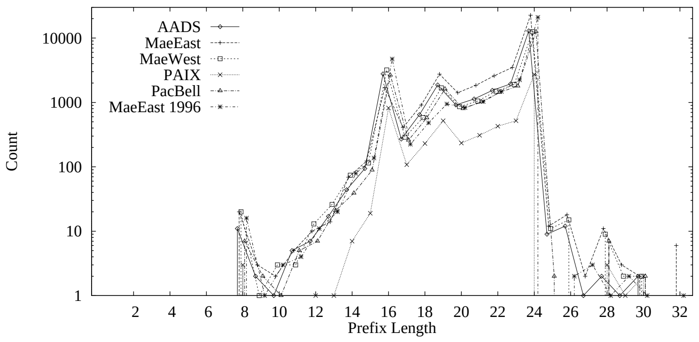


Fig. 8. Histogram of Backbone Prefix Length Distributions (log scale)

Table 1. Forwarding Tables: Total Prefixes, Distinct Lengths, and Distinct Lengths longer than 16 bit

	Prefixes	W_{dist}	$W_{dist \geq 16}$
AADS	24218	23	15
Mae-East	38031	24	16
Mae-West	23898	22	14
PAIX	5924	17	12
PacBell	22850	20	12
Mae-East 1996	33199	23	15

dominates everything by at least a factor of ten, if we ignore length 24. There are also more than 100 times as many prefixes at length 24 than at any prefix outside the range $15 \ldots 24$. This graph clearly shows the remnants of the original class A, B, and C networks with local maxima at lengths 8, 16, and 24. This distribution pattern is retained for many years now and seems to be valid for all backbone routing tables, independent of their size (Mae-East has over 38,000, while PAIX has less than 6,000 entries).

These characteristics visibly cry for optimizations. Although we will quantify the potential improvements using these forwarding tables, we believe that the optimizations introduced below apply to any current or future set of addresses.

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 $\log_2 W_{dist}$ ($4.1 \ldots 4.5$ with $W_{dist} \in [17, 24]$, according to Table 1). Figure 9 applies this to Mae-East's 1996 table. While this numerically improves the worst case, it harms the average performance, since the popular prefix lengths 8, 16, and 24 move to less favorable positions.

A more promising approach is to change the tree-shaped search pattern 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 7).

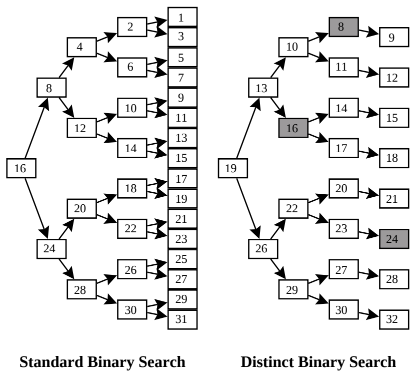


Fig. 9. Search Trees for Standard and Distinct Binary Search

Average time depends heavily on the traffic pattern seen at that location. Optimizing binary search trees according to usage pattern is an old problem [Knuth 1998]. By optimizing the average case, some data sets could degenerate towards linear search (Figure 10), which is clearly undesirable.

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 10, 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.

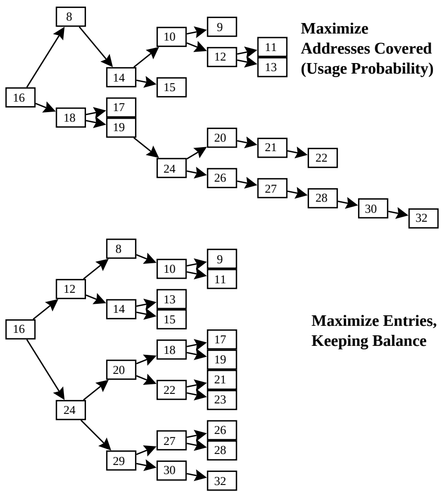


Fig. 10. Asymmetric Trees produced by two Weighting Functions

Table 2. Address (A) and Prefix (P) Count Coverage for Asymmetric Trees

Steps	Usage		Balance	
	A	P	A%	P%
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

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.

Previously, we tried to improve search time based on analysis of prefix distributions sorted by prefix lengths. The resulting histogram (Figure 8) led us to propose asymmetrical binary search, which can improve average speed. More information about prefix distributions can be extracted by further 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.

Table 3. Histogram of the Number of Distinct Prefix Lengths ≥ 16 in the 16 bit Partitions

	1	2	3	4	5	6	7	8	9
AADS	3467	740	474	287	195	62	11	2	1
Mae-East	2094	702	521	432	352	168	53	8	1
Mae-West	3881	730	454	308	158	70	17	3	—
PAIX	1471	317	139	56	41	31	1	—	—
PacBell	3421	704	442	280	168	42	9	—	—
Mae-East 1996	5051	547	383	273	166	87	27	3	—

When partitioning according to 16 bit prefixes[3], and counting the number of distinct prefix lengths in the partitions, we discover another nice property of the routing data. We recall the whole forwarding databases (Figure 8 and Table 1) showed up to 24 distinct prefix lengths with many buckets containing a significant number of entries and up to 16 prefix lengths with at least 16 bits. Looking at the sliced data in (Table 3), none of these partial histograms contain more than 9 distinct prefixes lengths; 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

[3]There is nothing magic about the 16 bit level, other than it being a natural starting length for a binary search of 32 bit IPv4 addresses.

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 naïve 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. Thus, binary search on prefix lengths has an advantage over conventional binary search: on each branch towards longer prefixes, not only the range of prefix lengths to be searched is reduced, but also the number of prefixes in each of these lengths. Binary search on prefix lengths thus narrows the search in *two dimensions* on each match, as illustrated in Figure 11.

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.

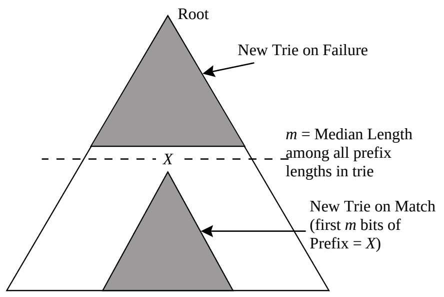

Root
New Trie on Failure
m = Median Length among all prefix lengths in trie
X
New Trie on Match (first m bits of Prefix = X)

Fig. 11. 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. The optimizations resulting from this observation improve lookups significantly:

Worst case: In all the databases we analyzed, we were able to reduce the worst case from five hashes to four hashes.

Average case: In the largest two databases, the majority of the addresses is found in at most two hash lookups. The smaller databases take a little bit longer to reach their halfway point.

Using Mutating Binary Search, looking for an address (see Figure 13) is different. First, we explain some new conventions for reading Figure 13. As in the other figures, we continue to draw a binary search tree on top. However, in this figure, we now have multiple partial trees, originating from any prefix entry. This is because the search process will move from tree to tree, starting with overall tree. Each binary tree has the "root" level (i.e., the first length to be searched) at the left; the left 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.

Consider now a search for address 1100110, matching the prefix labelled B, in the database of Figure 13. The search starts with the generic tree, so length 4 is checked, finding A. Among the prefixes starting with A, there are known to be only three distinct

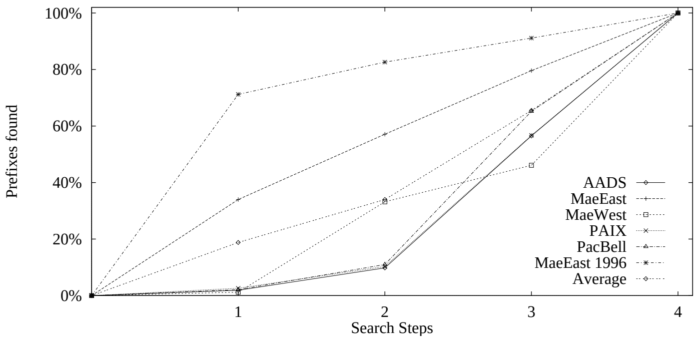


Fig. 12. Number of Hash Lookups (Note: No average-case optimizations)

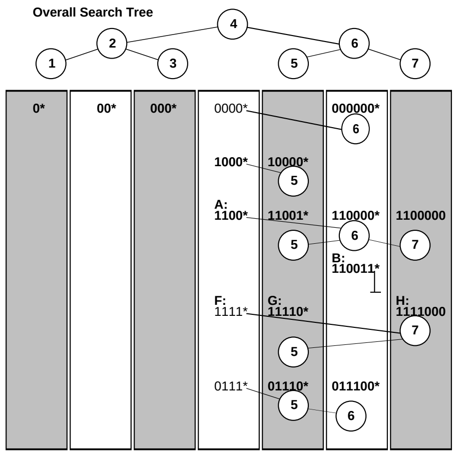


Fig. 13. Mutating Binary Search Example

lengths (4, 5, and 6). So A contains a description of the new tree, limiting the search appropriately. This tree is drawn as rooting in A. Using this tree, we find B, giving a new tree, the empty tree. The binary tree has *mutated* from the original tree of 7 lengths, to a secondary tree of 3 lengths, to a tertiary empty "tree".

Looking for 1111011, matching G, is similar. Using the overall tree, we find F. Switching to its tree, we miss at length 7. Since a miss (no entry found) can't update a tree, we follow our current tree upwards to length 5, where we find G.

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 *pre-computation* 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 the forwarding information for a given prefix may frequently change in cost or next hop, the addition or deletion of a new prefix (which is the expensive case) is be much rarer. We proceed to deal with the space issue by compactly encoding the network of trees.

4.2.1 *Bitmap.* One short encoding method would be to store a bitmap, with each bit set to one representing a valid level of the binary search tree. While this only uses W bits, computing a binary tree to follow next is an expensive task with current processors. The use of lookup tables to determine the middle bit is possible with short addresses (such as IPv4) and a binary search root close to the middle. Then, after the first lookup, there remain around 16 bits (less in upcoming steps), which lend themselves to a small (2^{16} bytes) lookup table.

4.2.2 *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 11), 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 search trees.

If we consider a binary search tree, 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 14. 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), and so on. The sequence $m, n, o, \ldots$ is the Rope for the trie.

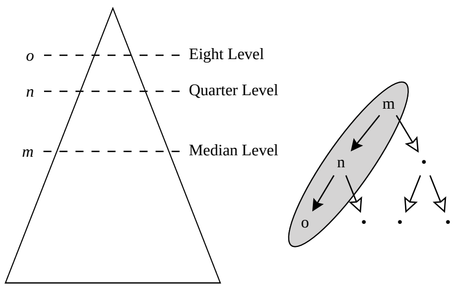


Fig. 14. In terms of a trie, a rope for the trie node is the sequence of lengths starting from the median length, the quartile length, and so on, which is the same as the series of left children (see dotted oval in binary tree on right) of a perfectly balanced binary tree on the trie levels.

Figure 15 shows the Ropes containing the same information as the trees in Figure 13. 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.

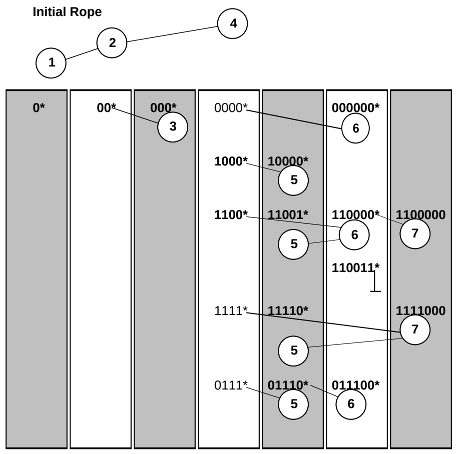


Fig. 15. Sample Ropes

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. Following the Rope on processors is easily done using “shift right” instructions.

Pseudo-code 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 “left” pointers to follow when no matches are found in the default tree. The resulting pseudo-code (Figure 16) is elegant and simple to implement. It appears to be simpler than the basic algorithm.

4.3 Trading Speed Against Memory

The following sections will discuss a number of mechanisms that allow tuning the tradeoff between search speed and memory requirements according to the application’s desires.

4.3.1 *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

```
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
```

Fig. 16. Rope Search

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, since there will be no misses at the starting level, which could direct the search below w_0.

4.3.2 *Halving the Prefix Lengths.* It is possible to reduce the worst case search time by another memory access. For that, we halve the number of prefix lengths by e.g. only allowing all the even prefix lengths, decreasing the $\log W$ search complexity by one. All the prefixes with odd lengths would then be expanded to two prefixes, each one bit longer. For one of them, the additional bit would be set to zero, for the other, to one. Together, they would cover the same range as the original prefix. At first sight, this looks like the memory requirement will be doubled. It can be shown that the worst case memory consumption is not affected, since the number of markers is reduced at the same time.

With W bits length, each entry could possibly require up to $\log(W) - 1$ markers (the entry itself is the $\log W$th entry). When expanding prefixes as described above, some of the prefixes will be doubled. At the same time, W is halved, thus each of the prefixes requires at most $\log(W/2) - 1 = \log(W) - 2$ markers. Since they match in all but their least bit, they will share all the markers, resulting again in at most $\log W$ entries in the hash tables.

A second halving of the number of prefixes again decreases the worst case search time, but this time increases the amount of memory, since each prefix can be extended by up to two bits, resulting in four entries to be stored, expanding the maximum number of entries needed per prefix to $\log(W) + 1$. For many cases the search speed improvement will warrant the small increase in memory.

4.3.3 *Internal Caching.* Figure 8 showed that the prefixes with lengths 8, 16, and 24 cover most of the address space used. Using binary search, these three lengths can be covered in just two memory accesses. To speed up the search, each address that requires more than two memory accesses to search for will be cached in one of these address lengths according to Figure 17. Compared to traditional caching of complete addresses, these cache prefixes cover a larger area and thus allow for a better utilization.

```
**Function** CacheInternally($A$, $P$, $L$, $M$)
(* found prefix $P$ at length $L$ after taking $M$ memory accesses
    searching for $A$ *)
**If** $M > 2$ then (* Caching can be of advantage *)
    Round up prefix length $L$ to next multiple of 8;
    Insert copy of $P$'s entry at $L$, using the $L$ first bits of $A$;
**Endif**
```

Fig. 17. Building the Internal Cache

4.4 Very Long Addresses

All the calculations above assume the processor's registers are big enough to hold entire addresses. For long addresses, such as those used for IP version 6, this does not always hold. We define w as the number of bits the registers hold. Instead of working on the entire address at once, the database is set up similar to a multibit trie [Srinivasan and Varghese 1999] of stride w, resulting in a depth of $k := W/w$. Each of these "trie nodes" is then implemented using binary search. If the "trie nodes" used conventional technology, *each of them* would require $O(2^w)$ memory, clearly impractical with modern processors, which manipulate 32 or 64 bits at a time.

Slicing the database into chunks of w bits also requires less storage than unsliced databases, since not the entire long addresses do not need to be stored with every element. The smaller footprint of an entry also helps with hash collisions (Section 7).

This storage advantage comes at a premium: Slower access. The number of memory accesses changes from $\log_2 W$ to $k + \log_2 w$, if the search in the intermediate "trie nodes" begins at their maximum length. This has no impact on IPv6 searches on modern 64 bit processors (Alpha, UltraSparc, Merced), which stay at 7 accesses. For 32 bit processors, the worst case using the basic scheme raises by 1, to 8 accesses.

4.5 Hardware Implementations

As we have seen in both Figure 7 and Figure 16, 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 [Spinney 1995], which stores all collisions from the hash table in a CAM. Instead of the hashing/CAM combinations, a large binary CAM could be used. Besides the hashing function described in [Spinney 1995], Cyclic Redundancy Check (CRC) generator polynomials are known to result in good hashing behavior (see also the comparison to other hashing functions in Section 7).

The outer loop in the Rope scheme can be implemented as a shift register, which is reloaded on every match found, as shown in Figure 18. This makes for a very simple hardware unit. For higher performances, the loop can be unrolled into a pipelined architecture. Pipelining is cheaper than replicating the entire lookup mechanism: in a pipelined implementation, each of the RAMs can be smaller, since it only needs to contain the entries that can be retrieved in its pipeline stage (recall that the step during which an entry is found depends only on the structure of the database, and not on the search key). Consult Figure 12 for a distribution of the entries among the different search steps. As is true for software search, Rope search will reduce the number of steps per lookup to at most 4 for IP version 4 addresses, and hardware may also use an initial array. Pipeline depth would therefore

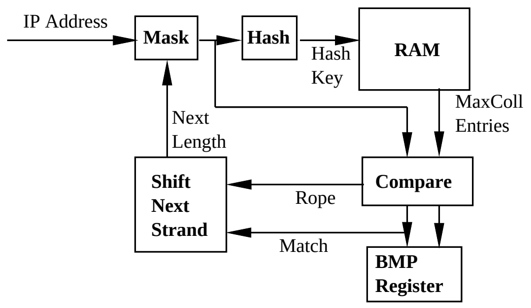


Fig. 18. Hardware Block Schematic

be four (or five, in a conservative design). Besides pipelining, converting binary branching to k-ary would provide another way around the relatively high memory access latencies. Instead of a single probe, as required for the binary decision, $k-1$ parallel probes would need to be taken. In our implementation [Braun et al. 2001], using parallel search engines turned out to be more efficient than using higher branching degrees when only a single external dynamic RAM (DRAM) module was available.

The highest speeds can be achieved using a pipelined approach, where each stage has its own memory. As of this writing, DRAM technology (DDR SDRAMs at 133 MHz), with information appropriately distributed and copied among the banks of the SDRAM, enables a throughput of 8 lookup every 9 cycles, resulting in 118 million packets per second with inexpensive hardware. This speed is roughly equivalent to 50 Gbit/s with minimum size packets (40 bytes) or more than 400 Gbit/s using measured packet distributions (354 bytes average) from June 1997.[4] Using custom hardware and pipelining, we thus expect a significant speedup to software performance, allowing for affordable IP forwarding reaching far beyond the single-device transmission speeds currently reached in high-tech research labs.

5. BUILDING AND UPDATING

Besides hashing and binary search, a predominant idea in this paper is *pre-computation*. Every hash table entry has an associated bmp field and (possibly) a Rope field, both of which are precomputed. Pre-computation allows fast search but requires more complex Insertion routines. However, as mentioned earlier, while the routes stored with the 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 Figure 19, requiring a complexity of $O(N \log W)$. 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 (see Section 7).

To build a basic search structure which eliminates unused levels or to take advantage of asymmetries, it is necessary to build the binary search tree first. Then, instead of clearing

[4]http://www.nlanr.net/NA/Learn/packetsizes.html

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

Fig. 19. Building for the Basic Scheme

the least significant bit, as outlined in Figure 19, the build algorithm really has to follow the binary search tree back up to find the "parent" prefix length. Some of these parents may be at longer prefix lengths, as illustrated in Figure 5. Since markers only need to be set at shorter prefix lengths, any parent associated with longer prefixes is just ignored.

5.2 Rope Search from Scratch

There are two ways to build the data structure suitable for Rope Search:

Simple: The search order does not divert from the overall binary search tree, only missing levels are left out. This results in only minor improvements on the search speed and can be implemented as a straightforward enhancement to Figure 19.

Optimal: Calculating the shortest Ropes on all branching levels requires the solution to an optimization problem in two dimensions. As we have seen, each branch towards longer prefix lengths also limits the set of remaining prefixes.

We present the algorithm which globally calculates the minimum Ropes, based on dynamic programming. The algorithm can be split up into three main phases:

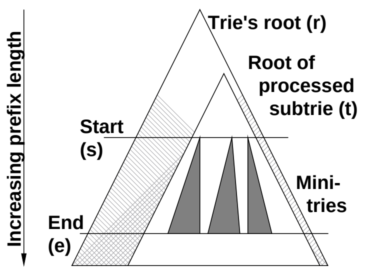
(1) Build a conventional (uncompressed) trie structure with $O(NW)$ nodes containing all the prefixes ($O(NW)$ time and space).

(2) Walk through the trie bottom-up, calculating the cost of selecting different branching points and combining them on the way up using dynamic programming ($O(NW^3)$ time and space).

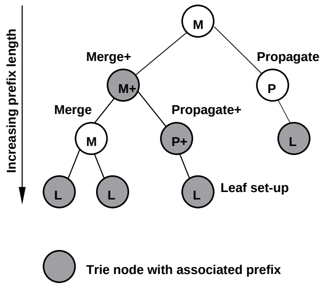
(3) Walk through the trie top-down, build the Ropes using the results from phase 2, and insert the entries into the hash tables ($O(NW \log W)$ time, working on the space allocated in phase 2).

To understand the bottom-up merging of the information in phase 2, let us first look at the information that is necessary for bottom-up merging. Recall the Ropes in Figure 15. At each branching point, the search either turns towards longer prefixes and a more specific branching tree, or towards shorter prefixes without changing the set of levels. The goal is to minimize worst-case search cost, or the number of hash lookups required. The overall cost of putting a decision point at prefix length x is the maximum path length on either side plus one for the newly inserted decision. Looking at Figure 15, the longest path on the left of our starting point has length two (the paths to $0*$ or $000*$). When looking at the right hand side, the longest of the individual searches require two lookups ($11001*$, 1100000, $11110*$, and 0111000).

Generalizing, for each range R covered and each possible prefix length x splitting this range into two halves, R_l and R_r, the program needs to calculate the maximum depth of the *aggregate* left-hand tree R_l, covering shorter prefixes, and the maximum depth of the *individual* right-hand trees R_r. When trying to find an optimal solution, the goal is to minimize these maxima, of course. Clearly, this process can be applied recursively. Instead of implementing a simple-minded recursive algorithm in exponential time, we use dynamic programming to solve it in polynomial time.



(a) Structures



(b) Cases treated

Fig. 20. Rope Construction, Phase 2

Figure 20(a) shows the information needed to solve this minimization problem. For each subtree t matching a prefix P, a table containing information about the depth associated with the subrange R ranging from start length s to end length e is kept. Specifically, we keep (1) the maximum over all the *individual* minimal-depth trees (T_I), as used for branching towards longer prefixes and (2) the minimal *aggregate* tree (T_A), for going to shorter prefixes. Each of these trees in turn consists of both a left-hand aggregate tree and right-hand individual branching trees.

Using the dynamic programming paradigm, we start building a table (or in this case, a table per trie node) from the bottom of the trie towards the root. At each node, we combine the information the children have accumulated with our local state, i.e. whether this node is an entry. Five cases can be identified: (L) setting up a leaf node, (P) propagating the aggregate/individual tables up one level, (P+) same, plus including the fact that this node contains a valid prefix, (M) merging the child's aggregate/individual tables, and (M+)

merging and including the current node's prefix. As can be seen, all operations are a subset of (M+), working on less children or not adding the current node's prefix. Figure 21 lists the pseudo-code for this operation.

```
Function Phase2MergePlus;
Set $p$ to the current prefix length;

(* Merge the children's $T_I$ below $p$ *)
Forall $s, e$ where $s \in [p+1 \ldots W], e \in [s \ldots W]$;
    (* Merge the $T_I$ mini-trees between Start $s$ and End $e$ *)
    If both children's depth for $T_I[s, e]$ is 0 then
        (* No prefixes in either mini-tree *)
        Set this node's depth for $T_I[s, e]$ to 0;
    Else
        Set this node's depth for $T_I[s, e]$ to the
            the max of the children's $T_I[s, e]$ depths;
    Endif
Endforall

(* "Calculate" the depth of the trees covering just this node *)
If the current entry is a valid prefix then
    Set $T_I[p, p] = T_A[p, p] = 1$; (* A tree with a single entry *)
Else
    Set $T_I[p, p] = T_A[p, p] = 0$; (* An empty tree *)
Endif

(* Merge the children's $T_A$, extend to current level *)
For $s \in [p \ldots W]$;
    For $e \in [s+1 \ldots W]$;
        (* Find the best next branching length $i$ *)
        Set $T_A[s, e]$'s depth to $\min(T_I[s+1, e] + 1)$, (* split at $s$ *)
            $\min_{i=s+1}^{e}(\max(T_A[s, i-1] + 1, T_I[i, e])))$; (* split below *)
        (* Since $T_A[s, i-1]$ is only searched after missing at $i$, add 1 *)
    Endfor
Endfor

(* "Calculate" the $T_I$ at $p$ also *)
Set $T_I[p, *]$ to $T_A[p, *$; (* Only one tree, so aggregated=individual *)
```

Fig. 21. Phase 2 Pseudo-code, run at each trie node

As can be seen from Figure 21, merging the T_As takes $O(W^3)$ time per node, with a total of $O(NW)$ nodes. The full merging is only necessary at nodes with two children, shown as (M) and (M+) in Figure 20(b). In any trie, there can be only $O(N)$ of them, resulting in an overall build time of only $O(NW^3)$.

If the optimal next branching point is stored alongside each $T_A[s, e]$, building the rope for any prefix in Phase 3 is a simple matter of following the chain set by these branching points, by always following $T_A[s_{\mathit{prev}} + 1, \mathit{previous\ branching\ point}]$. A node will be used as a marker, if the higher-level rope lists its prefix length.

5.2.1 *Degrees of Freedom.* The only goal of the algorithm shown in Figure 21 is to minimize the worst-case number of search steps. Most of the time multiple branching points will result in the same minimal T_A depth. Therefore, choosing the split point gives a further degree of freedom to optimize other factors within the bounds set by the calculated worst case. This freedom can be used to (1) reduce the number of entries requiring the worst case lookup time, (2) improve the average search time, (3) reduce the number of markers placed, (4) reduce the number of hash collisions, or (5) improve update behavior (see below). Because of limitations in space and scope, they will not be discussed in more depth.

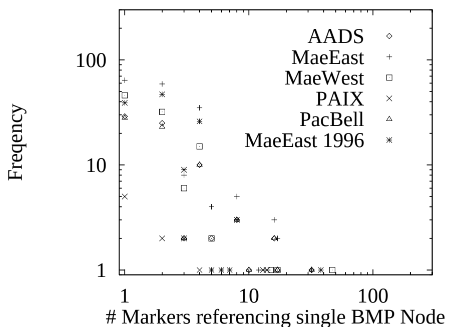
5.3 Insertions and Deletions

As shown in [Labovitz et al. 1997], some routers receive routing update messages at high frequencies, requiring the routers to handle these messages within a few milliseconds. Luckily for the forwarding tables, most of the routing messages in these bursts are of pathological nature and do not require any change in the routing or forwarding tables. Also, most routing updates involve only a change in the route and do not add or delete prefixes. Additionally, many wide-area routing protocols such as BGP [Rekhter and Li 1995] use timers to reduce the rate of route changes, thereby delaying and batching them. Nevertheless, algorithms in want of being ready for further Internet growth should support sub-second updates under most circumstances.

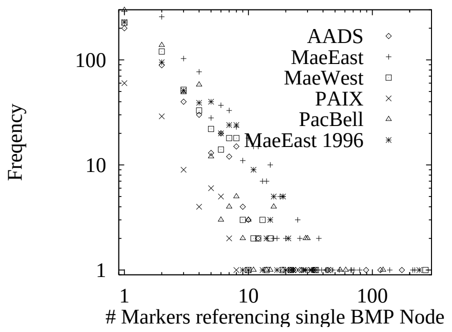
Adding entries to the forwarding database or deleting entries may be done without rebuilding the whole database. The less optimized the data structure is, the easier it is to change it.

5.3.1 *Updating Basic and Asymmetric Schemes.* We therefore start with basic and asymmetric schemes, which have only eliminated prefix lengths which will never be used. Insertion and deletion of leaf prefixes, i.e. prefixes, that do not cover others, is trivial. Insertion is done as during initial build (Figure 19). For deletion, a simple possibility is to just remove the entry itself and not care for the remaining markers. When unused markers should be deleted immediately, it is necessary to maintain per-marker reference counters. On deletion, the marker placement algorithm from Figure 19 is used to determine where markers would be set, decreasing their reference count and deleting the marker when the counter reaches zero.

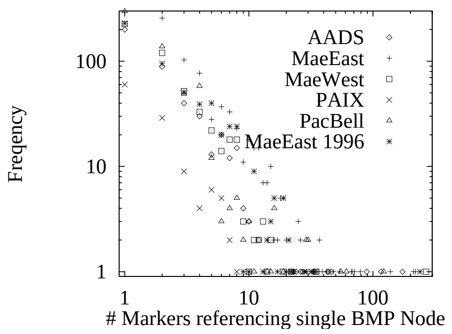
Should the prefix p being inserted or deleted cover any markers, these markers need to be updated to point to their changed BMP. There are a number of possibilities to find all the underlying markers. One that does not require any helper data structures, but lacks efficiency, is to either enumerate all possible longer prefixes matching our modified entry, or to walk through all hash tables associated with longer prefixes. On deletion, every marker pointing to p will be changed to point to p's BMP. On insertion, every marker pointing p's current BMP and matching p will be updated to point to p. A more efficient solution is to chain all markers pointing to a given BMP in a linked list. Still, this method could require $O(N \log W)$ effort, since p can cover any amount of prefixes and markers from the entire forwarding database. Although the number of markers covered by any given prefix was small in the databases we analyzed (see Figure 22), Section 6 presents a solution to bound the update efforts, which is important for applications requiring real-time guarantees.



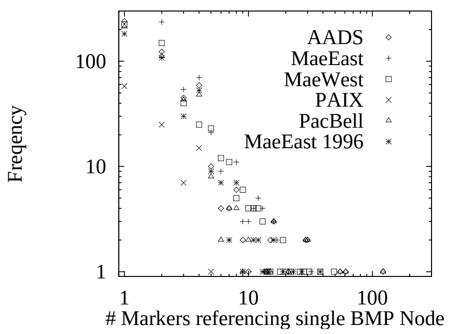
(a) “Pure Basic” (without Length Elimination)



(b) Basic



(c) Asymmetric



(d) Rope

Fig. 22. Histogram of Markers depending on a Prefix (log scales)

During the previous explanation, we have assumed that the prefix being inserted had a length which was already used in the database. In Asymmetric Search, this may not always be true. Depending on the structure of the binary search trie around the new prefix length, adding it is trivial. The addition of length 5 in Figure 23(a) is one of these examples. Adding length 6 in Figure 23(b) is not as easy. One possibility, shown in the upper example, is to re-balance the trie structure, which unlike balancing a B-tree can result in several markers being inserted: One for each pre-existing prefix not covered by our newly inserted prefix, but covered by its parent. This structural change can also adversely affect the average case behavior. Another possibility, shown in the lower right, is to immediately add the new prefix length, possibly increasing the worst case for this single prefix. Then we wait for a complete rebuild of the tree which takes care of the correct re-balancing.

We prefer the second solution, since it does not need more than the plain existing insertion procedures. It also allows for updates to take effect immediately, and only incurs a negligible performance penalty until the database has been rebuilt. To reduce the frequency

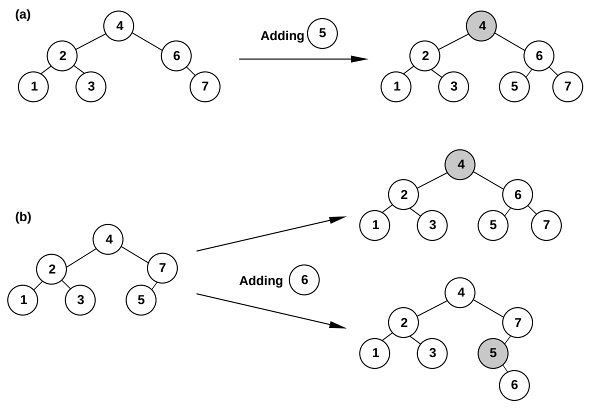


Fig. 23. Adding Prefix Lengths (Gray Nodes change Rope)

of rebuilds, the binary search tree may be constructed as to leave room for inserting the missing prefix lengths at minimal cost. A third solution would be to split a prefix into multiple longer prefixes, similar to the one used by Causal Collision Resolution Section 7.1.

5.3.2 *Updating Ropes.* All the above insights also apply to Rope Search, and even more so, since it uses many local asymmetric binary search trees, containing a large number of uncovered prefix lengths. Inserting a prefix has a higher chance of adding a new prefix length to the current search tree, but it will also confine the necessary re-balancing to a small subset of prefixes. Therefore, we believe the simplest, yet still very efficient, strategy is to add a marker at the longest prefix length shorter than p's, pointing to p. If this should degrade the worst-case search time, or anyway after a large number of these insertions, a background rebuild of the whole structure is ordered. The overall calculation of the optimal branching points in phase 2 (Figure 21) is very expensive, $O(NW^3)$, far more expensive than calculating the ropes and inserting the entries Table 4. Just recalculating to incorporate the changes induced by a routing update is much cheaper, as only the path from this entry to the root needs to be updated, at most $O(W^4)$, giving a speed advantage over simple rebuild of around three orders of magnitude. Even though Rope Search is optimized to very closely fit around the prefix database, Rope Search still keeps enough flexibility to quickly adapt to any of the changes of the database.

Table 4. Build Speed Comparisons (Built from Trie)

	Basic	Rope			Entries
	Hash	Phase 2	Ropes	Hash	
AADS	0.56s	11.84s	0.59s	0.79s	24218
Mae-East	1.82s	14.10s	0.85s	1.69s	38031
Mae-West	0.58s	11.71s	0.60s	0.85s	23898
PAIX	0.09s	4.16s	0.18s	0.07s	5924
PacBell	0.48s	11.04s	0.57s	0.73s	22850
Mae-East 1996	1.14s	13.08s	0.75s	1.12s	33199

The times in Table 4 were measured using completely unoptimized code on a 300 MHz

UltraSparc-II. We would expect large improvements from optimizing the code. “Hash” refers to building the hash tables, “Phase 2” is phase 2 of the rope search, “Ropes” calculates the ropes and sets the markers. Just adding or deleting a single entry takes orders of magnitudes less time.

6. MARKER PARTITIONING

The scheme introduced below, *recursive marker partitioning*, significantly reduces the cost of marker updates identified as a problem above. It does this by requiring at most one additional memory access per entire search, whenever the last match in the search was on a marker. Using rope search on the examined databases, an additional memory lookup is required for $2 \ldots 11\%$ of the addresses, a negligible impact on the average search time. Of the searches that require the identified worst case of four steps, only $0 \ldots 2\%$ require an additional fifth memory access.

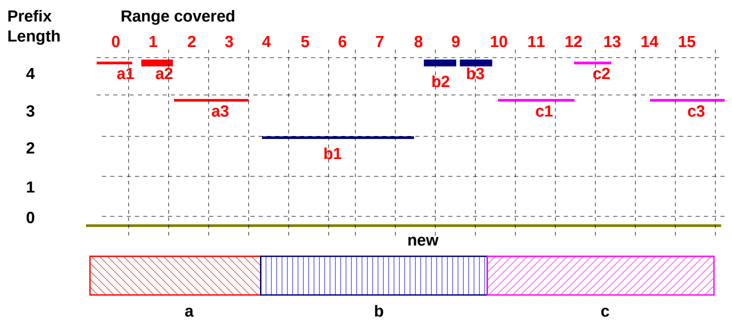
Furthermore, prefix partitioning offers a tunable tradeoff between the penalty incurred for updates and searches, which makes it very convenient for a wide range of applications.

6.1 Basic Partitioning

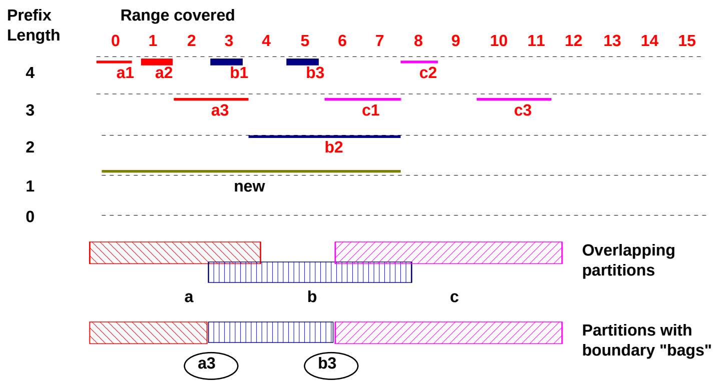
To understand the concept and implications of partitioning, we start with a single layer of partitions. Assume an address space of 4 bits with addresses ranging from 0 to 15, inclusive. This space also contains nine markers, labeled $a1$ to $c3$, as shown in Figure 24(a). For simplicity, the prefixes themselves are not shown. Recall that each marker contains a pointer to its BMP. This information requires update whenever the closest covering prefix is changed.

Assume the prefix designated *new* is inserted. Traditional approaches would require the insert procedure to walk through all the markers covered by *new* and correct their BMP, taking up to $N \log W$ steps. *Marker partitioning* groups these markers together. Assume we had grouped markers a_1 to a_3 in group a, markers b_1 to b_3 in b, and c_1 to c_3 in c. Note that the prefixes in the group are disjoint and hence, we can store a single overlapping BMP pointer information for all of them instead of at each of them individually. Thus, in this example, we would remember only three such entries — one per group or partition. This improves the time required from updating each entry to just modifying the information common to the group. In our example above (Figure 24(a)), when adding the *new* prefix, we see that it entirely covers the partitions a, b and c. Thus, our basic scheme works well as long as the partition boundaries can be chosen so that no marker overlaps them and the new prefix covers entire groups.

But when looking at one more example in Figure 24(b), where partition A contains markers a_1, a_2, a_3, partition B contains b_1, b_2, b_3 and partition C contains c_1, c_2, c_3. Clearly, the partition boundaries now overlap. Although in this example it is possible to find partitionings without overlaps, prefixes covering a large part of the address space would severely limit the ability to find enough partitions. Thus, in the more general case, the boundaries between the splits are no longer well-defined; there are overlaps. Because of the nature of prefix-style ranges, at most W distinct ranges may enclose any given point. This is also true for the markers crossing boundary locations. So at each boundary, we could store the at most W markers that overlap it and test against these special cases individually when adding or deleting a prefix like *new*. It turns out to be enough to store these overlapping markers at only a single one of the boundaries it crosses. This is enough, since its BMP will only need to change when a modification is done to an entry covering our



(a) Simple Partitioning Example



(b) Partitions with Overlaps

Fig. 24. Marker partitioning explained

prefix.

For simplicity of the remaining explanations in this section, it is assumed that it is possible to split the prefixes in a non-overlapping fashion. One way to achieve that would be to keep a separate marker partition for each prefix length. Clearly, this separation will not introduce any extra storage and the search time will be affected by at most a factor of W.

Continuing our example above (Figure 24(b)), when adding the *new* prefix, we see that it entirely covers the partitions a, b and partially covers c. For all the covered partitions, we update the partitions' Best Match. Only for the partially covered partitions, we need to process their individual elements. The changes for the BMP pointers are outlined in bold in the Table 5. The real value of the BMP pointer is the entry's value, if it is set, or the partition's value otherwise. If neither the entry nor the entry's containing partition contain any information, as is the case for c_3, the packet does not match a prefix (filter) at this level.

Generalizing to p partitions of e markers each, we can see that any prefix will cover at most p partitions, requiring at most p updates.

At most two partitions can be partially covered, one at the start of the new prefix, one at

Table 5. Updating Best Matching Prefixes

Entry/Group	Old BMP stored	New BMP stored	Resulting BMP
a_1	—	—	new
a_2	—	—	new
a_3	—	—	new
a	—	new	(N/A)
b_1	a_3	a_3	a_3
b_2	—	—	new
b_3	b_2	b_2	b_2
b	—	new	(N/A)
c_1	—	new	new
c_2	—	—	—
c_3	—	—	—
c	—	—	(N/A)

the end. In a simple-minded implementation, at most e entries need to be updated in each of the split partitions. If more than $e/2$ entries require updating, instead of updating the majority of entries in this partition, it is also possible to relabel the container and update the minority to store the container's original value. This reduces the update to at most $e/2$ per partially covered marker, resulting in a worst-case total of $p + 2e/2 = p + e$ updates.

As $p * e$ was chosen to be N, minimizing $p + e$ results in $p = e = \sqrt{N}$. Thus, the optimal splitting solution is to split the database into $\sqrt{N}$ sets of $\sqrt{N}$ entries each. This reduces update time from $O(N)$ to $O(\sqrt{N})$ at the expense of at most a single additional memory access during search. This memory access is needed only if the entry does not store its own BMP value and we need to revert to checking the container's value.

6.2 Dynamic Behavior

Insertion and deletion of prefixes often goes ahead with the insertion or deletion of markers. Over time, the number of elements per partition and also in the total number of entries, N, will change. The implications of these changes are discussed below. For readability, S will be used to represent $\sqrt{N}$, the optimal number of partitions and entries per partition.

The naïve solution of re-balancing the whole structure is to make all partitions equal size after every change to keep them between $\lfloor S \rfloor$ and $\lceil S \rceil$. This can be done by 'shifting' entries through the list of partitions in $O(S)$ time. This breaks as soon as the number of partitions needs to be changed when S crosses an integer boundary. Then, $O(S)$ entries need to be shifted to the partition that is being created or from the partition that is being destroyed, resulting in $O(N)$ entries to be moved. This obviously does not fit into our bounded update time.

We need to be able to create or destroy a partition without touching more than $O(S)$ entries. We thus introduce a deviation factor, d, which defines how much the number of partitions, p, and the number of elements in each partition, e_i, may deviate from the optimum, S. The smallest value for d which allows to split a maximum-sized partition (size Sd) into two partitions not below the minimum size S/d and vice versa is $d = \sqrt{2}$. This value will also satisfy all other conditions, as we will see.

Until now, we have only tried to keep the elements e_i in each partition within the bounds set by S and d. As it turns out, this is satisfactory to also force the number of partitions p within these bounds, since $N/\min e_i > S/d$ and $N/\max e_i < Sd$.

Whenever a partition grows too big, it is *split* into two or *distributes* some of its contents across one or both of its neighbors, as illustrated in Figure 25. Conversely, if an entry is getting too small, it either *borrows* from one or both of its neighbors, or *merges* with a suitably small neighbor. Clearly, all these operations can be done with touching at most Sd entries and at most 3 partitions.

The *split* operation is sufficient to keep the partitions from exceeding their maximum size, since it can be done at any time. Keeping partitions from shrinking beyond the lower limit requires both *borrow* (as long as at least one of the neighbors is still above the minimum) and *merge* (as soon as one of them has reached the minimum).

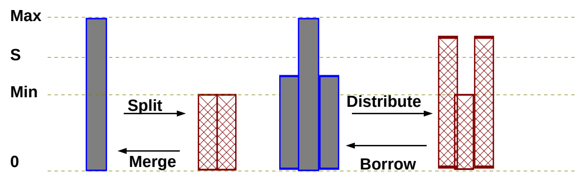


Fig. 25. Dynamic Operations

S crossing an integer boundary may result in all partitions to become either too big or too small in one instant. Obviously, not all of them can be *split* or *merged* at the same time without violating the $O(S)$ bound. Observe that there will be at least $2S+1$ further insertions or $2S-1$ deletions until S crosses the next boundary. Also observe that there will be at most S/d maximum-sized entries and Sd minimum-sized entries reaching the boundaries.[5] If we extend the boundaries by one on each side, there is plenty of time to perform the necessary splits or merges one by one before the boundaries change again.

Instead of being 'retro-active' with splitting and joining, it can also be imagined to be pro-active. Then, always the partition furthest away from the optimal value would try to get closer to the optimum. This would make the updates even more predictable, but at the expense of always performing splits or joins.

To summarize, with the new bounds of $S/d-1$ to $Sd+1$, each insertion or deletion of a node requires at most $2(Sd+1)$ updates of BMP pointers, moving $Sd/2$ entries to a new partition, and on boundary crossing $Sd+1$ checks for minimal size partitions. This results in $O(Sd)$ work, or with d chosen a constant $\sqrt{2}$, $O(S) = O(\sqrt{N})$. All further explanations will consider $d=\sqrt{2}$. Also, since we have $O(s)$ partitions, each with $O(s)$ pointers, the total amount of memory needed for the partitions is $O(N)$.

6.3 Multiple Layers of Partitioning

We have shown that with a single layer of partitions, update complexity can be limited to $O(\sqrt{N})$ with at most a single additional memory access during search.

It seems natural to extend this to more than one layer of grouping and to split the partitions into sub-partitions and sub-sub-partitions, similar to a tree. Assume we defined a tree of α layers (including the leaves). Each of the layers would then contain $s = \sqrt[\alpha]{N}$ entries or sub-partitions of the enclosed layer. As will be shown below, the update time is

[5] If there are more than $Sd/2$ minimum-sized entries, than some of them have to be right beside each other. Then a single *merge* will eliminate two of them. Therefore, there will be at most $Sd/2$ operations necessary to eliminate all minimum-sized entries.

then reduced to $O(\alpha \sqrt[\alpha]{N})$ at the expense of up to $\alpha - 1$ memory accesses to find the Best Match associated with the innermost container level who has it set.

Prefix updates At the outermost layer, at most sd containers will be covered, with at most two of them partially. These two in turn will contain at most sd entries each, of which at the most $sd/2$ need to be updated, and at most one further split partition. We continue this until the innermost level is found, resulting in at most $sd + (\alpha - 1)2sd/2$ changes, or $O(s)$.

Splitting and Joining At any one level, the effort is s. In the worst case, α levels are affected, giving $O(s\alpha)$.

Boundary Crossing of s The number of insertions or deletions between boundary crossings is $(s + 1)^{\alpha} - s^{\alpha}$, while the number of minimal-sized partitions is $\sum_{i=1}^{a-1} s^i = (s^{\alpha} - s)/(s - 1)$. So there is enough time to amortize the necessary changes over time one by one during operations that do not themselves cause a split or join.

6.4 Further Improvements

For many filter databases it would make sense to choose α dynamically, based on the real number of entries. The total number of markers for most databases will be much less than the worst case. If optimal search time should be achieved with bounded worst-case insertion, it seems reasonable to reduce the partition nesting depth to match the worst-case update. Often, this will reduce the nesting to a single level or even eliminate it.

7. FAST HASHING WITH BOUNDED COLLISIONS

Many algorithms are known for hashing. Since we have mentioned a single memory access per lookup, the number of collisions needs to be tightly bounded. One well-known solution is *perfect hashing* [Fredman et al. 1984]. Unfortunately, true perfect hashing requires enormous amounts of time to build the hash tables and also requires complex functions to locate the entries. While perfect hashing is a solution that satisfies the $O(1)$ access requirement, it is often impractical. An improvement, dynamic perfect hashing [Dietzfelbinger et al. 1994], also achieves $O(1)$ lookup time at amortized cost of $O(1)$ per insertion, by having a two-level hierarchy of randomly chosen hashing functions. Thus, it requires two memory accesses per hash lookup, making it an attractive option.

With memory prices dropping, memory cost is no longer one of the main limiting factor in router design. Therefore, it is possible to relax the hashing requirements. First, we no longer enforce optimal compaction, but allow for sparse hash tables. This already greatly reduces the chances for collisions.

Second, we increase the hash bucket size. With current DRAM technologies, the cost of a random access to a single bit is almost indistinguishable from accessing many bytes sequentially. Modern CPUs take advantage of this and always read multiple consecutive words, even if only a single byte is requested. The amount of memory fetched per access, called a *cache line*, ranges from 128 to 256 bits in modern CPUs. This cache line fetching us to store a (small) number of entries in the same hash bucket, with no additional memory access penalty (recall that for most current processors, access to main memory is much slower than access to on-chip memory and caches or instruction execution.)

We have seen several key ingredients: randomized hash functions (usually only a single parameter is variable), over-provisioning memory, and allowing a limited number of collisions, as bounded by the bucket size. By combining these ingredients into a hash function, we were able to achieve single memory access lookup with almost $O(1)$ amortized insertion time.

In our implementations, we have been using several hash functions. One group of functions consists of non-parametric functions, each one utilizing several cheap processor instructions to achieve data scrambling. Switching between these functions is achieved by changing to a completely new search function, either by changing a function pointer or by overwriting the existing function with the new one.

The other group consists of a single function which can be configured by a single parameter, using $f(\text{Key} * \text{Scramble}) * \text{BucketCount}$, where f is a function returning the fractional part, *Key* is the key to be hashed, Scramble $\in (0 \ldots 1]$ is a configurable scrambling parameter, and *BucketCount* is the number of available hash buckets. This function does not require floating point and can be implemented as fixed-point arithmetic using integer operations. Since multiplication is generally fast on modern processors, calculation of the hash function can be hidden behind other operations. Knuth [Knuth 1998] recommends the scrambling factor to be close to the conjugated golden ratio $((\sqrt{5} - 1)/2)$. This function itself gives a good tradeoff between the collision rate and the additional allocation space needed.

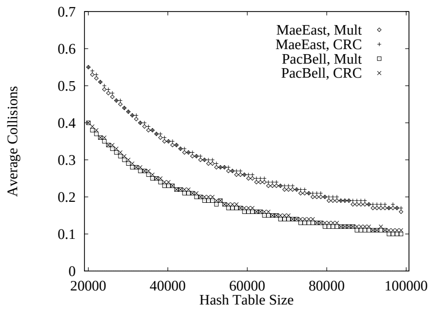
It is possible to put all the hash entries of all prefix lengths into one big hash table, by using just one more bit for the address and setting the first bit below the prefix length to 1. This reduces the collision rate even further with the same total memory consumption. Since multiplication is considered costly in hardware, we also provide a comparison with a 32-bit Cyclic Redundancy Check code (CRC-32), as used in the ISO 3309 standard, in ITU recommendation V.42, and the GZIP compression program [Deutsch 1996]. In Figure 26(b), a soft lowpass filter has been applied to increase readability of the graph, eliminating single peaks of $+1$. Since only primes in steps of about 1000 apart are used for the table sizes, there is always a prime hash table size available nearby which fulfills the limit.

Depending on the width of the available data path, it might thus be more efficient to allow for more collisions, thus saving memory. Memory requirements are still modest. A single hash table entry for 32 bit lookups (IPv4) can be stored in as little as 6 or 8 bytes, for the basic schemes or rope search, respectively. Allowing for five entries per hash bucket, the largest database (Mae East) will fit into 1.8 to 2.4 megabytes. Allowing for six collisions, it will fit into 0.9 to 1.2 MB.

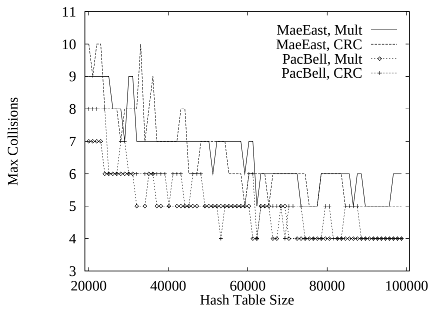
7.1 Causal Collision Resolution

As can be seen from Figure 26, only very few entries create collisions. If we could reduce collisions further, especially at these few “hot spots”, we could optimize memory usage or reduce the number of operations or the data path width. In this section, we present a technique called “Causal Collision Resolution” (CCR), which allows us to reduce collisions by adapting the marker placement and by relocating hash table entries into different buckets. We have seen that there are several degrees of freedom available when defining the binary search (sub-)trees for Asymmetric and Rope Search (Section 5.2.1), which help to move markers.

Moving prefixes is also possible by turning one prefix colliding with other hash table entries into two. Figure 27(a) illustrates the expansion of a prefix from length l to two prefixes at $l+1$, covering the same set of addresses. This well-known operation is possible whenever the l is not a marker level for $l+1$ (otherwise, a marker with the same hash key as



(a) Average Collisions



(b) Maximum Collisions

Fig. 26. Collisions versus Hash Table Size

the original prefix would be inserted at l, nullifying our efforts). When expansion doesn't work, it is possible to "contract" the prefix (Figure 27(b)). It is then moved to length $l-1$, thus covering too large a range. By adding a prefix C at l, complementing the original prefix within the excessive range at $l-1$, the range can be corrected. C stores the original BMP associated with that range.

The two binary search trees shown in Figure 27 are only for illustrative purposes. Expansion and contraction also work with other tree structures. When other prefixes already exist at the newly created entries, precedence is naturally given to the entries originating from longer prefix lengths. Expansion and contraction can also be generalized in a straightforward way to work on more than ±1 prefix lengths.

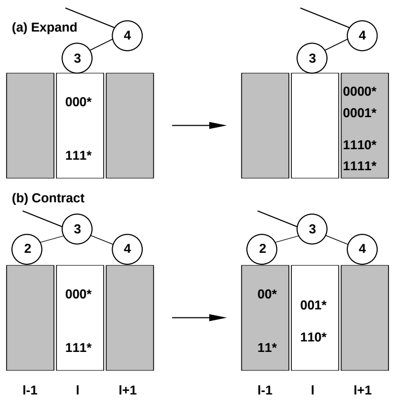
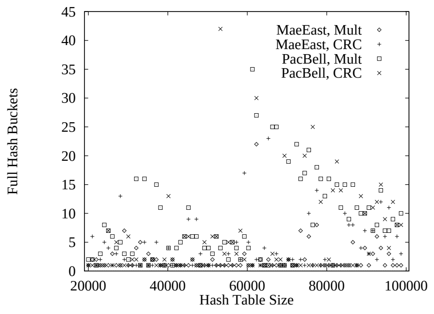
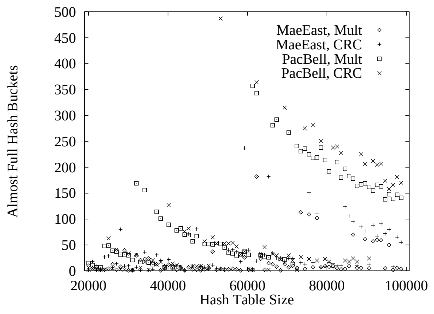


Fig. 27. Causal Collision Resolution

In Figure 28 the number of buckets containing the most collisions and those containing just one entry less are shown. As can be seen, for the vast majority of hash table configurations, only less than a handful of entries define the maximum bucket size. In almost half of the cases, it is a single entry. Even for the buckets with one entry less than the



(a) Number of Hash Buckets with Maximum Collisions



(b) Number of Hash Buckets with One Entry Less

Fig. 28. Number of (Almost) Full Hash Buckets

maximum size (Figure 28(b)), a negligible amount of buckets (less than 1 per thousand for most configurations) require that capacity.

Using causal collision resolution, it is possible to move one of the "surplus" entries in the biggest buckets to other buckets. This makes it possible to shrink the bucket sizes by one or two, reducing the existing modest memory requirements by up to a factor of two.

8. PERFORMANCE EVALUATION

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

8.1 Marker Requirements

Although we have seen that adding markers could extend the number of entries required by a factor $\log_2 W$. In the typical case, many prefixes will share markers (Table 6), reducing the marker storage further. Notice the difference between "Max Markers", the number of markers requested by the entries, and "Effective Markers", how many markers really needed to be inserted, thanks to marker sharing. In our sample routing databases, the additional storage required due to markers was only a fraction of the database size. 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). The numbers listed below are taking from "Plain Basic" scheme, but the amount of sharing is comparable with other schemes.

8.2 Complexity Comparison

Table 7 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.

Table 6. Marker Overhead for Backbone Forwarding Tables

	Total	Basic: Request for					Max	Effective
	Entries	0	1	2	3	4	Markers	Markers
AADS	24218	2787	14767	4628	2036	0	30131	9392
Mae-East	38031	1728	25363	7312	3622	6	50877	13584
Mae-West	23898	3205	14303	4366	2024	0	29107	9151
PAIX	5924	823	3294	1266	541	0	7449	3225
PacBell	22850	2664	14154	4143	1889	0	28107	8806
Mae-East 1996	33199	4742	22505	3562	2389	1	36800	8342

Table 7. Speed and Memory Usage Complexity

Algorithm	Build	Search	Memory	Update
Binary Search	$O(N \log N)$	$O(\log N)$	$O(N)$	$O(N)$
Trie	$O(NW)$	$O(W)$	$O(NW)$	$O(W)$
Radix Trie[6]	$O(NW)$	$O(W)$	$O(N)$	$O(W)$
Basic Scheme	$O(N \log W)$	$O(\log W)$	$O(N \log W)$	$O(N)$
or		$O(\alpha + \log W)$		$O(\alpha \sqrt[\alpha]{N} W \log W)$
Asymmetric BS	$O(N \log W)$	$O(\log W)$	$O(N \log W)$	$O(N)$
or		$O(\alpha + \log W)$		$O(\alpha \sqrt[\alpha]{N} W \log W)$
Rope Search	$O(NW^3)$	$O(\log W)$	$O(N \log W)$[7]	$O(N)$
or		$O(\alpha + \log W)$		$O(\alpha \sqrt[\alpha]{N} W \log W)$
Ternary CAMs	$O(N)$	$O(1)$[8]	$O(N)$	$O(N)$

8.3 Measurements for IPv4

Many measurements on real-world data have already been included earlier in this paper. To summarize, we have shown that with modest memory requirements of less than a megabyte and simple hardware or software, it is possible to achieve fast best matching prefix lookups with at most four memory accesses, some of them may even be resolved from cache.

8.4 Projections for IP Version 6

Although there originally were several proposals for IPv6 address assignment principles, the *aggregatable global unicast address format* [Hinden et al. 1998] is at the verge of being deployed. 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 ("multi-homing") and meshing between the ISPs, we expect the routing tables to grow. Another new feature of IPv6, Anycast addresses [Hinden and Deering 1998; Deering and Hinden 1998], 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 4.3.1, the improvement may not be as dramatic as for IPv4. Although it will improve performance for IPv6, after length 16 (which happens to be a "magic length" for the aggregatable global unicast address format), only a smaller percentage of the address space will have been covered. Only time will tell whether this initial step will be of advantage. All other optimizations are expected to yield similar improvements.

9. 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 [Sklower 1993] were known, we have discovered an efficient algorithm that scales with the logarithm of the address size and so is 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 pre-computation, 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, reducing the number of expensive memory accesses required considerably. 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.

We have also shown that updates to our data structure can be very simple, with a tight bound around the expected update efforts. Furthermore, we have introduced causal collision resolution which exploits domain knowledge to simplify collision resolution.

With algorithms such as ours and that of others, 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. If processor speeds do not keep up with these expectations, extremely affordable hardware (around US$ 100) enables forwarding speeds of around 250 Gbit/s, much faster than any single transmitter can currently achieve even in the research laboratories. Therefore, 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 throughout the network.

Our algorithm has already been successfully included into the BBN multi-gigabit per second router [Partridge et al. 1998], which can do the required Internet packet processing and forwarding decisions for $10 \ldots 13$ million packets per second using a single off-the-shelf microprocessor. Besides performance for IPv6, our algorithm was also chosen as it could naturally and efficiently handle 64 bit wide prefixes (which occur while concatenating destination and source addresses when forwarding IP multicast packets).

A more challenging topic beyond prefix lookups is packet classification, where multi-dimension prefix matches have to be performed, often combined with exact and range matches. Many of the one-dimensional lookup techniques (including the one described in this paper) have been used as lookups on individual fields, whose results are then combined later [Gupta and McKeown 1999; Srinivasan et al. 1998]. The main idea of this paper, namely working non-linearly in the prefix length space, has been directly generalized to multi-dimensional packet classification schemes such as tuple space search [Srinivasan et al. 1999] and line search [Waldvogel 2000].

We believe that trie-based and CAM-based schemes will continue to dominate in IPv4-based products. However, the slow, but ongoing, trend towards IPv6 will give a strong edge to schemes scalable in terms of prefix lengths. Except for tables where path compression is very effective[9], we believe that our algorithm will be better than trie-based algorithms for IPv6 routers. Perhaps our algorithm was adopted in the BBN router in anticipation of such a trend.

For future work, we are attempting to fine-tune the algorithm and are looking for other applications. Thus we are working to improve the update behavior of the hash functions even further, and are studying the effects of internal caching. 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; besides packet classification, we believe that our algorithm and its improvements may be applicable in other domains besides Internet packet forwarding. Potential applications we are investigating include memory management using variable size pages, access protection in object-oriented operating systems, and access permission management for web servers and distributed file systems.

ACKNOWLEDGMENTS

We thank V. Srinivasan, Thomas Meyer, Milind Buddhikot, Subhash Suri, and Marcel Dasen for many helpful interactions, which resulted in substantial improvements of this paper. We are also extremely grateful to have received extensive and useful comments by the anonymous reviewers.

REFERENCES

ANDERSSON, A. AND NILSSON, S. 1994. Faster searching in tries and quadtrees – an analysis of level compression. In *Second Annual European Symposium on Algorithms* (1994), pp. 82–93.

BRAUN, F., WALDVOGEL, M., AND LOCKWOOD, J. 2001. OBIWAN – an internet protocol router in reconfigurable hardware. Technical Report WU-CS-01-11 (May), Washington University in St. Louis.

CHANDRANMENON, G. AND VARGHESE, G. 1995. Trading packet headers for packet processing. In *Proceedings of SIGCOMM '95* (Boston, Aug. 1995). Also in *IEEE Transactions on Networking*, April 1996.

CRESCENZI, P., DARDINI, L., AND GROSSI, R. 1999. IP lookups made fast and simple. In *7th Annual European Symposium on Algorithms* (July 1999). Also available as technical report TR-99-01, Dipartimento di Informatica, Università di Pisa.

DE BERG, M., VAN KREVELD, M., AND SNOEYINK, J. 1995. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms 18*, 2, 256–277.

DEERING, S. AND HINDEN, R. 1998. Internet protocol, version 6 (IPv6) specification. Internet RFC 2460.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proceedings of ACM SIGCOMM '97* (Sept. 1997), pp. 3–14.

DEUTSCH, L. P. 1996. GZIP file format specification. Internet RFC 1952.

DIETZFELBINGER, M., MEHLHORN, K., ROHNERT, H., KARLIN, A., MEYER AUF DER HEIDE, F., AND TARJAN, R. E. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing 23*, 4, 748–761.

[9] For instance, the initial IPv6 tables may be derived from IPv4 tables by adding a long prefix. In such cases, path compression will be very effective.

EATHERTON, W. N. 1999. Hardware-based Internet protocol prefix lookups. Master's thesis, Washington University in St. Louis, St. Louis, MO, USA.

FELDMEIER, D. C. 1988. Improving gateway performance with a routing-table cache. In *Proceedings of IEEE Infocom '88* (New Orleans, March 1988), pp. 298–307.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM 31*, 3, 538–544.

FULLER, V., LI, T., YU, J., AND VARADHAN, K. 1993. Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy. Internet RFC 1519.

GUPTA, P., LIN, S., AND MCKEOWN, N. 1998. Routing lookups in hardware at memory access speeds. In *Proceedings of IEEE Infocom* (April 1998), pp. 1240–1247.

GUPTA, P. AND MCKEOWN, N. 1999. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM '99* (Cambridge, Massachusetts, USA, Sept. 1999), pp. 147–160.

GWEHENBERGER, G. 1968. Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen (Use of a binary tree structure for processing files). *Elektronische Rechenanlagen 10*, 223–226.

HINDEN, R. AND DEERING, S. 1998. IP version 6 addressing architecture. Internet RFC 2373.

HINDEN, R., O'DELL, M., AND DEERING, S. 1998. An IPv6 aggregatable global unicast address format. Internet RFC 2374.

KNUTH, D. E. 1998. *Sorting and Searching* (2nd ed.), Volume 3 of *The Art of Computer Programming*. Addison-Wesley.

KOBAYASHI, M., MURASE, T., AND KURIYAMA, A. 2000. A longest prefix match search engine for multi-gigabit ip processing. In *Proceedings of the International Conference on Communications* (June 2000).

LABOVITZ, C., MALAN, G. R., AND JAHANIAN, F. 1997. Internet routing instability. In *Proceedings of ACM SIGCOMM '97* (1997), pp. 115–126.

LAMPSON, B., SRINIVASAN, V., AND VARGHESE, G. 1998. IP lookups using multiway and multicolumn search. In *Proceedings of IEEE Infocom '98* (San Francisco, 1998).

MCAULEY, A. J. AND FRANCIS, P. 1993. Fast routing table lookup using CAMs. In *Proceedings of Infocom '93* (March–April 1993), pp. 1382–1391.

MORRISON, D. R. 1968. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15*, 514–534.

NEWMAN, P., MINSHALL, G., AND HUSTON, L. 1997. IP Switching and gigabit routers. *IEEE Communications Magazine 35*, 1 (Jan.), 64–69.

NILSSON, S. AND KARLSSON, G. 1999. Ip address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications 15*, 4 (June), 1083–1092.

PARTRIDGE, C. 1996. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis* (San Diego, CA, USA, Feb. 1996). Available at http://www.caida.org/outreach/9602/positions/partridge.html.

PARTRIDGE, C., CARVEY, P. P., ET AL. 1998. A 50-gb/s IP router. *IEEE/ACM Transactions on Networking 6*, 3 (June), 237–248.

PERLMAN, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley.

REKHTER, Y., DAVIE, B., KATZ, D., ROSEN, E., AND SWALLOW, G. 1997. Cisco systems' tag switching architecture overview. Internet RFC 2105.

REKHTER, Y. AND LI, T. 1995. A border gateway protocol 4 (BGP-4). Internet RFC 1771.

ROSEN, E. C., VISWANATHAN, A., AND CALLON, R. 2001. Multiprotocol label switching architecture. Internet RFC 3031.

RUIZ-SÁNCHEZ, M. A., BIERSACK, E. W., AND DABBOUS, W. 2001. Survey and taxonomy of ip address lookup algorithms. *IEEE Network 15*, 2 (March–April), 8–23.

SHAH, D. AND GUPTA, P. 2000. Fast incremental updates on ternary-cams for routing lookups and packet classification. In *Proceedings of Hot Interconnects* (2000).

SKLOWER, K. 1993. A tree-based packet routing table for Berkeley Unix. Technical report, University of California, Berkeley. Also at http://www.cs.berkeley.edu/˜sklower/routing.ps.

SPINNEY, B. A. 1995. Address lookup in packet data communications link, using hashing and content-addressable memory. U.S. Patent number 5,414,704. Assignee Digital Equipment Corporation, Maynard, MA.

SRINIVASAN, V., SURI, S., AND VARGHESE, G. 1999. Packet classification using tuple space search. In *Proceedings of ACM SIGCOMM '99* (Cambridge, Massachusetts, USA, Sept. 1999), pp. 135–146.

SRINIVASAN, V. AND VARGHESE, G. 1999. Fast address lookups using controlled prefix expansion. *Transactions on Computer Systems 17*, 1 (Feb.), 1–40.

SRINIVASAN, V., VARGHESE, G., SURI, S., AND WALDVOGEL, M. 1998. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98* (Sept. 1998), pp. 191–202.

VAN EMDE BOAS, P. 1975. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science* (1975), pp. 75–84.

VAN EMDE BOAS, P., KAAS, R., AND ZULSTRA, E. 1977. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10*, 99–127.

WALDVOGEL, M. 2000. Multi-dimensional prefix matching using line search. In *Proceedings of IEEE Local Computer Networks* (Nov. 2000), pp. 200–207.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM '97* (Sept. 1997), pp. 25–36.