IP Lookups using Multiway and Multicolumn Search

B. Lampson, V. Srinivasan and G. Varghese
blampson@microsoft.com,cheenu,varghese@dworkin.wustl.edu

May 11, 1998

Abstract

IP address lookup is becoming critical because of increasing routing table size, speed, and traffic in the Internet. Our paper shows how binary search can be adapted for best matching prefix using two entries per prefix and precomputation. Next we show how to improve the performance of any best matching prefix scheme using an initial array indexed by the first X bits of the address. We then describe how to take advantage of cache line size to do a multiway search with 6-way branching. Finally, we show how to extend the binary search solution and the multiway search solution for longer 128 bit IPv6 addresses. For a database of N prefixes with address length W, naive binary search scheme would take $O(W * logN)$; we show how to reduce this to $O(W + logN)$ using multiple column binary search. Measurements using a real IP database of 38000 entries (Mae-East) resulted in a worst case lookup time of 490 nanoseconds and insertion times of around 350 msec. Measurements for smaller databases (e.g., Paix) have results that are competitive with the best previous schemes for IPv4. In addition, our scheme is particularly attractive for IPv6 because of small storage requirement (2N nodes) and speed (estimated worst case of 7 SDRAM burst READs using a burst length of 4.)

1 Introduction

Statistics show that the number of hosts on the internet is tripling approximately every two years [16]. Traffic on the Internet is also increasing exponentially. Traffic increase can be traced not only to increasing numbers of hosts, but also to new applications (e.g., the Web, video conferencing, remote imaging) which have higher bandwidth needs than traditional applications. One can only expect further increases in users, hosts, domains, and traffic. The possibility of a global Internet with multiple addresses per user (e.g., for appliances) has necessitated a transition from the older Internet routing protocol (IPv4 with 32 bit addresses) to the proposed next generation protocol (IPv6 with 128 bit addresses).

Success opens up new sets of problems. As ordinary users begin to depend on the Internet, it is increasingly important that the Internet provide adequate performance. As a first step, communication links in the Internet backbone are being upgraded with high speed fiber optic links. However, even Gigabit links are insufficient unless Internet routers are able to forward packets at Gigabit rates. High speed packet forwarding is compounded by increasing routing database sizes (due to increased number of hosts) and longer addresses (due to the transition to IPv6). Our paper deals with the problem of increasing IP packet forwarding rates in routers. In particular, we deal with a component of high speed forwarding, *address lookup*, that is a major bottleneck.

When an Internet router gets a packet P from an input link interface, it uses the destination address in packet P to lookup a routing database. The result of the lookup provides an output link interface, to which packet P is forwarded. There is some additional bookkeeping such as updating packet headers, but the major tasks in packet forwarding are address lookup and switching packets between link interfaces.

For Gigabit routing, many solutions exist which do fast switching within the router box [15]. Despite this, the problem of doing lookups at Gigabit speeds remains. For example, Ascend's product [1] has *hardware* assistance for lookups and can take up to 3 microseconds for a single lookup in the worst case and 1 microsecond on average. However, to support say 5 Gbps with an average packet size of 512 bytes, lookups need to be performed in 800 nsec per packet. By contrast, our scheme can be implemented in *software* on an ordinary PC in a worst case time of 490 nsec.

The Best Matching Prefix Problem: Address lookup can be done at high speeds if we are looking for an *exact match* of the packet destination address to a corresponding address in the routing database. Exact matching can be done using standard techniques such as hashing or binary search. Unfortunately, most routing protocols (including OSI and IP) use hierarchical addressing to avoid scaling problems. Rather than have each router store a database entry for all possible destination IP addresses, the router stores address *prefixes* that represent a group of addresses reachable through the same interface. The use of prefixes allows scaling to worldwide networks.

The use of prefixes introduces a new dimension to the lookup problem: multiple prefixes may match a given address. If a packet matches multiple prefixes, it is intuitive that the packet should be forwarded corresponding to the *most specific* prefix or *longest* matching prefix. IPv4 prefixes are arbitrary bit strings up to 32 bits in length as shown in Table 1. To see the difference between the exact matching and best matching prefix, consider a 32 bit address A whose first 8 bits are 10000111. If we searched for A in the above table, exact match would not give us a match. However prefix matches are 100* and 1000*, of which the longest matching prefix is 1000*, whose next hop is $L5$.

Paper Organization: The rest of this paper is organized as follows. Section 2 describes a model and the parameters used to evaluate schemes. Section 3 describes related work and briefly describes our contribution. Section 4 contains our basic binary search scheme. Section 5 describes a basic idea of using an array as a front end to reduce the number of keys required for binary search. Section 6 describes how we exploit the locality inherent in cache lines to do multiway binary search; we also describe measurements

Prefix	Next hop
*	L9
001*	L1
0001*	L2
011111*	L3
100*	L4
1000*	L5
10001*	L6

Table 1: A sample routing table

for a sample large IPv4 database. Section 8 describes how to do multicolumn and multiway binary search for IPv6. We also describe some measurements and projected performance estimates. Section 9 states our conclusions.

2 Performance Model

The choice of a lookup algorithm depends crucially on assumptions about the routing environment and the implementation environment. We also need a performance model with precise metrics to compare algorithms.

2.1 Routing Environment

The Internet consists of local domains which are interconnected by a backbone consisting of multiple Internet Service Providers (ISPs). Accordingly, there are two interesting kinds of routers[2]: enterprise routers (used in a campus or organization) and backbone routers (used by ISPs). The performance needs of these two routers are different.

Backbone routers today [2] can have databases of up to 45,000 prefixes (growing every day, several of them with multiple paths). The prefixes contain almost all lengths from 8 to 32; however, because of the evolution from Class B and Class C addresses, there is considerable concentration at 24 and 16 bit prefix lengths. Backbone routers typically run the Border Gateway Protocol (BGP). Because some BGP implementations exhibit considerable instability, route changes can occur up to 100 times a second[2, 21]. This requires algorithms for handling route updates that take 10 msec or less. Backbone routers may require frequent reprogramming as ISPs attempts to deal with customer requirements such as virus attacks. The packet sizes seen are bimodal, and are typically either 64 bytes control packets or 576 byte data packets.

Enterprise routers have smaller databases (up to 1000 prefixes) because of the heavy use of default routes for outside destinations. Routes are also typically much more stable, requiring route updates at most once every few seconds. The packet sizes are bimodal and are either 64 bytes or 1519 byte.[1] However, large multi-campus enterprise routers may look more like backbone routers.

Address space depletion has lead to the proposal for the next generation of IP (IPv6) with 128 bit addresses. While there are plans for aggressive aggregation to reduce table entries, the requirement for both provider based and geographical addresses, the need for connections to multiple ISPs, plans to

[1] 576 byte data packets arise in ISPs because of the use of a default size of 576 bytes for wide area traffic; 1519 byte size packets in the enterprise network probably arises from Ethernet maximum size packets.

connect control devices on the Internet, and the use of features like Anycast [7], all make it unlikely that backbone prefix tables will be smaller than in IPv4.

We use four publically available prefix databases for our comparisons. These are made available by the IPMA project[12] and are daily snapshots of the routing tables used at some major Network Access Points (NAPs). The largest of these, Mae East (about 38,000 prefixes), is a reasonable model for a large backbone router; the smallest database, PAIX (around 713 prefixes) can be considered a model for an enterprise router. We will compare lookup schemes using these four databases with respect to three metrics: search time (most crucial), storage, and insert/delete times,

2.2 Implementation Model

We will consider both hardware and software platforms for implementing lookups. Software platforms are more flexible and have smaller initial design costs; hardware platforms have higher performance and are cheaper after volume manufacturing. For example, BBN [3] uses DEC Alpha CPUs in each line card, while Torrent and Rapid City [3] use hardware forwarding engines.

Software: We will consider software platforms using modern processors such as the Pentium[17] and the Alpha[6]. These CPU s execute simple instructions very fast (few clock cycles) but take much longer to make a random access to main memory. However, if the data is in Primary (L1) or Secondary Cache (L2) access times are only a few clock cycles. The distinction arises because main memory uses slow cheap Dynamic Memory (DRAM , 60-100 nsec access time) while cache memory is expensive but fast Static Memory (SRAM , 10-20 nsec). When a READ of a single word is made to memory, an entire *cache line* is fetched into the cache. This is important because the remaining words in the cache line can be accessed cheaply for the price of a single memory READ .

Thus a first cut measure of the speed of any lookup algorithm, in either software or hardware, is the number of main memory (DRAM) accesses required, because these accesses often dominate search times. To count memory accesses, we must have an estimate of the total storage required by the algorithm to understand how much of the data structures can be placed in cache.

We chose a 300 Mhz Pentium II (cost under 5000 dollars) running Windows NT that has a 512 KBytes L2 cache and a cache line size of 32 bytes. We chose this platform because of the popularity of Wintel platforms, and the availability of useful tools. We believe the results would be similar if run on other comparable platforms such as the Alpha. For worst case comparisons, we use repeated lookups to a single address that we (independently) determine will cause the worst case traversal of the data structure. In addition we also measure average speed using accesses to a million randomly chosen IP addresses.

Hardware: To compare IP lookup schemes in hardware, we assume a cheap forwarding engine (say 20 dollars assuming high volumes) operating at a clock rate of 10 nsec. We assume the chip can place its data structure in SRAM (with 10nsec access times) and/or DRAM (60-100 nsec access times). We assume, following current prices, that SRAM costs six times as much as DRAM per byte. While it possible to buy SRAM in small granularities, DRAM comes in large granularities (e.g., 2M bits). As in the software case, if we make a single READ to memory followed by subsequent READS to adjacent locations, we can use Page Mode DRAM to speed up the access to these adjacent locations. Thus, as in the software case, it pays to align data structures, so that memory references are made to adjacent locations.

3 Previous Work and Our Contributions

In this section, we review previous work on the longest matching prefix problem.

The current NetBSD implementation of IP lookup [25, 23] uses a *Patricia Trie* which processes an address one bit at a time. On a 200 MHz pentium, with about 33,000 entries in the routing table, this takes 1.5 to 2.5 microseconds on the average. These numbers will worsen with larger databases. [23] mentions that the expected number of bit tests for the patricia tree is 1.44 log N, where N is the number of entries in the table. For N=32000, this is over 21 bit tests. With memory accesses being very slow for modern CPUs, 21 memory accesses is excessive. Patricia tries also use skip counts to compress one way branches, which necessitates backtracking. Such backtracking slows down the algorithm and makes pipelining difficult.

Many authors have proposed tries of high radix [19] but only for exact matching of addresses. OSI address lookups are done naturally using trie search 4 bits at a time [18]. This works because OSI prefix lengths are always multiples of 4.

Our basic binary tree method is described very briefly in a page in a popular textbook[18], based on work by the first author of this paper. The main ideas have never been published in a technical paper. Further, the ideas of using an initial array, multicolumn and multiway binary search (which are crucial to the competitiveness of our scheme) have never been described before. Our description also has actual measurements, and an explanation of algorithm correctness.

[15] claims that it is possible to do a lookup in 200 nsec using SRAMs (with 10 nsec cycle times) to store the entire routing database. We note that large SRAMs are extremely expensive and are typically limited to caches in ordinary processors.

[4] uses a hardware solution based on content-addressable memories(CAMs) for implementing best matching prefix. CAMs use brute force hardware parallelism to simultaneously compare every prefix against the incoming address. The scheme in [4] uses a separate CAM for each possible prefix length. For IPv4 this can require 32 CAMs and 128 CAMs for IPv6, which is expensive. A cheaper solution (one CAM that stores all prefixes of all lengths) is to use CAMs that allow each entry in the CAM to be bitwise maskable; however, this requires more transistors per CAM cell, which reduces CAM density. This in turn results in the need for more CAMs (and hence greater expense) to handle large IP databases.

Caching is a standard solution for improving average performance. However, experimental studies have shown poor cache hit ratios for backbone routers[15]. This is partly due to the fact that caches typically store whole addresses. Finally, schemes like Tag and Flow Switching suggest protocol changes to avoid the lookup problem altogether. These proposals depend on widespread acceptance, and do not completely eliminate the need for lookups at network boundaries.

In the last year, two new techniques [5, 27] for doing best matching prefix have been announced. The approach in [5] is based on compressing trie nodes so that they will fit into the cache. The approach in [27] is based on doing binary search on the *possible prefix lengths*. Another form of compressed tries, *LC tries* is presented in [14].

Another approach invented by us based on prefix expansion [24] seems to be simple and fast. However, the technique that we present in this paper is attractive for IPv6 because of its bounded worst case memory requirement. For IPv4, the binary search scheme presented in this paper would be more appropriate than other schemes for small enterprise routers that have less than 1000 prefixes. We present measured comparisons of our scheme with other schemes in Section 7.

Our Contributions: In this paper, we start by showing how to modify binary search to do best matching prefix. Our basic binary search technique has been described briefly in Perlman's book [18] but is due to the first author of this paper and has never been published before. The enhancements to the use of an initial array, multicolumn and multiway search, implementation details, and the measurements have never been described before.

Modified binary search requires two ideas: first, we treat each prefix as a range and encode it using

the start and end of range; second, we arrange range entries in a binary search table and precompute a mapping between consecutive regions in the binary search table and the corresponding prefix.

Our approach is completely different from either [5, 27] as we do binary search on the *number of possible prefixes* as opposed to *the number of possible prefix lengths*. For example, the naive complexity of our scheme is $\log_2 N + 1$ memory accesses, where N is the number of prefixes; by contrast, the complexity of the [27] scheme is $\log_2 W$ hash computations plus memory accesses, where W is the length of the address in bits.

At a first glance, it would appear that the scheme in [27] would be faster (except potentially for hash computation, which is not required in our scheme) than our scheme, especially for large prefix databases. However, we show that we can exploit the locality inherent in processor caches and fast cache line reads using SDRAM or RDRAM to do multiway binary search in $\log_{k+1} N + 1$ steps, where $k > 1$. We have found good results using $k = 5$. By contrast, it appears to be impossible to modify the scheme in [27] to do multiway search on prefix lengths because each search in a hash table only gives two possible outcomes.

Further, for long addresses (e.g., 128 bit IPv6 addresses), the true complexity of the scheme in [27] is closer to $O((W/M) \log_2 W)$, where M is the word size of the machine.[2] This is because computing a hash on a W bit address takes $O(W/M)$ time. By contrast, we introduce a multicolumn binary search scheme for IPv6 and OSI addresses that takes $\log_2 N + W/M + 1$. Notice that the W/M factor is additive and not multiplicative. Using a machine word size of $M = 32$ and an address width W of 128, this is a potential multiplicative factor of 4 that is avoided in our scheme. We contrast the schemes in greater detail later.

The approach in [5] is based on compressing k-bit trie nodes and takes O(W/k) time. It differs entirely from our approach. While the approach in [5] has search times comparable to ours for IPv4 (around 500 nsec), our approach should scale better for IPv6 when W becomes 128.

We also describe a simple scheme of using an initial array as a front end to reduce the number of keys required to be searched in binary search. Essentially, we partition the original database according to every possible combination of the first X bits. Our measurements use $X = 16$. Since the number of possible prefixes that begin with a particular combination of the first X bits is much smaller than the total number of prefixes, this is a big win in practice. [8] presents a scheme that uses $X = 24$, and uses compressed trie nodes to represent prefixes longer than 24 bits.

Our paper describes the results of several other measurements of speed and memory usage for our implementations of these two schemes. The measurements allow us to isolate the effects of individual optimizations and architectural features of the CPUs we used. We describe results using several publically available routing databases (especially the Mae-East NAP) for IPv4, and by using randomly chosen 128 bit addresses for IPv6.

Measurements using the (Mae-East) database of 30000 entries yield a worst case lookup time of 490 nanoseconds, five times faster than the performance of the Patricia trie scheme used in BSD UNIX used on the same database. We estimate a worst case insertion time of around 300 msec, which seems adequate for routing updates in an enterprise routing environment. We also estimate the performance of our scheme for IPv6 using a special SDRAM or RDRAM memory (which is now commercially available, though we could not obtain one to do actual experiments). This memory allows fast access to data within a page of memory, which enables us to speed up multiway search. Thus, in the worst case we estimate 7 cache line reads to search a large database of IPv6 entries.

Please note that in this paper, by *memory reference* we mean accesses to main memory. Cache hits are not counted as memory references. If a cache line of 32 bytes is read, then accessing two different bytes in the 32 byte line is counted as one memory reference. This is justifiable, as a main memory read

[2]The scheme in mvt starts by doing a hash of $W/2$ bits; it can then do a hash on $3W/4$ bits, followed by $7W/8$ bits etc. Thus in the worst case, each hash may operate on roughly $3W/4$ bits.

has an access time of 60 nsec while the on-chip L1 cache can be read at the clock speed (5 nsec on an Intel Pentium Pro). With SDRAM or RDRAM, a cache line fill is counted as one memory access. With SDRAM a cache line fill is a burst read with burst length 4. While the first read has an access time of 60 nsec, the remaining 3 reads have access times of only 10 nsec each [13]. With RDRAM, an entire 32 byte cache line can be filled in 101 nsec [20]. Thus prefetching an entire cache line costs only a little more than a random memory access.

4 Adapting Binary search for Longest Matching Prefix Search

Having described the model and performance measures, we now proceed to describe the main ideas behind our IP lookup scheme. The root idea behind our scheme is binary search. Recall, however, that binary search is typically used for *exact match* among fixed length keys while we need to find a *longest match* among variable length prefixes. Thus adapting binary search to longest matching prefix requires several subtle modifications.

To illustrate the progression of ideas, we use a simple contrived example in which we have 6 bit addresses and just three prefixes: 1*, 101*, and 10101*. Our first problem is that binary search does not work with variable length strings. Thus the simplest approach is to pad each prefix to be a 6 bit string by adding zeroes. This is shown in Figure 1.

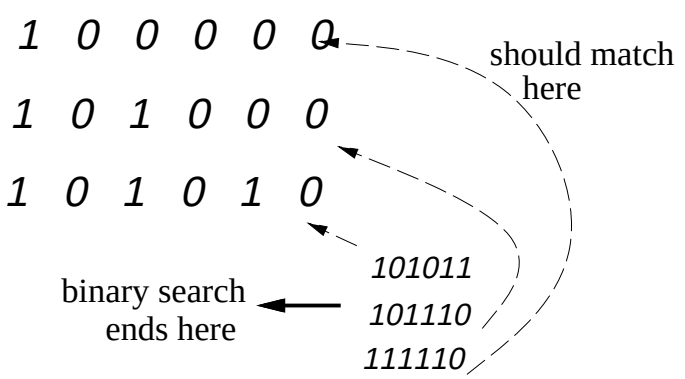

Figure 1: Placing the three prefixes 1*, 101*, and 10101* in a binary search table by padding each prefix with 0's to make 6 bit strings and sorting the resulting strings. Note that the addresses 101011, 101110, and 111110 all end up in the same region in the binary search table (after the last entry) despite the fact that they have different longest matching prefixes.

Now consider a search for the three 6 bit addresses 101011, 101110, and 111110. Since none of these addresses are in the table, binary search will fail. Unfortunately, on a failure all three of these addresses will end up at the end of the table because all of them are greater than 101010, which is the last element in the binary search table. Notice however that each of these three addresses (see Figure 1) has a different best matching prefix.

Thus we have two problems with naive binary search: first, when we search for an address we end up far away from the matching prefix (potentially requiring a linear search); second, multiple addresses that match to different prefixes, end up in the same region in the binary table (Figure 1).

Encoding Prefixes as Ranges To solve the second problem, we recognize that a prefix like 1* is really a range of addresses from 100000 to 111111. Thus instead of encoding 1* by just 100000 (the start of the range), we encode it using both the start and end of range (100000 *and* 111111). Thus each prefix is encoded by two full length bit strings. These bit strings are then sorted. The result for the same three prefixes is shown in Figure 2.

We connect the start and end of a range (corresponding to a prefix) by a line in Figure 2. Notice how the ranges are nested. If we now try to search for the same set of addresses, they each end in a different region in the table. To be more precise, the search for address 101011 ends in an exact match. The search for address 101110 ends in a failure in the region between 101011 and 101111 (Figure 2), and the search for address 111110 ends in a failure in the region between 101111 and 111111. Thus it appears that the second problem (multiple addresses that match different prefixes ending in the same region of the table) has disappeared. Compare Figure 1 and Figure 2.

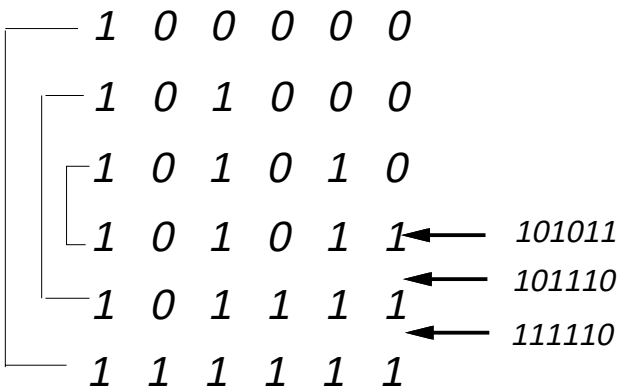

Figure 2: We now encode each prefix in the table as a range using two values: the start and end of range. This time the addresses that match different prefixes end up in different ranges.

To see that this is a general phenomenon, consider Figure 3. The figure shows an arbitrary binary search table after every prefix has been encoded by the low (marked **L** in Figure 3) and its high points (marked **H**) of the corresponding range. Consider an arbitrary position indicated by the solid arrow. If binary search for address A ends up at this point, which prefix should we map A to? It is easy to see the answer visually from Figure 3. If we start from the point shown by the solid arrow and we go back up the table, the prefix corresponding to A is the first **L** that is not followed by a corresponding **H** (see dotted arrow in Figure 3.)

Why does this work? Since we did not encounter an **H** corresponding to this **L**, it clearly means that A is contained in the range corresponding to this prefix. Since this is the first such **L**, this is the smallest such range. Essentially, this works because the best matching prefix has been translated to the problem of finding the *narrowest enclosing range.*

4.1 Using Precomputation to Avoid Search

Unfortunately, the solution depicted in Figure 2 and Figure 3 does not solve the first problem: notice that binary search ends in a position that is far away (potentially) from the actual prefix. If we were to search for the prefix (as described earlier), we could have a linear time search.

However, the modified binary search table shown in Figure 3 has a nice property we can exploit. *Any region in the binary search between two consecutive numbers corresponds to a unique prefix.* As described earlier, the prefix corresponds to the first **L** before this region that is not matched by a corresponding **H** that also occurs before this region. Similarly, every exact match corresponds to a unique prefix.

Since this is the case, we can precompute the prefix corresponding to each region and the prefix corresponding to each exact match. This can potentially slow down insertion. However, the insertion or deletion of a new prefix should be a rare event (the next hop to reach a prefix may change rapidly, but the addition of a new prefix should be rare) compared to packet forwarding. Thus slowing down insertion costs for the sake of faster forwarding is a good idea. (In Section 7.5 we show that insertion into even

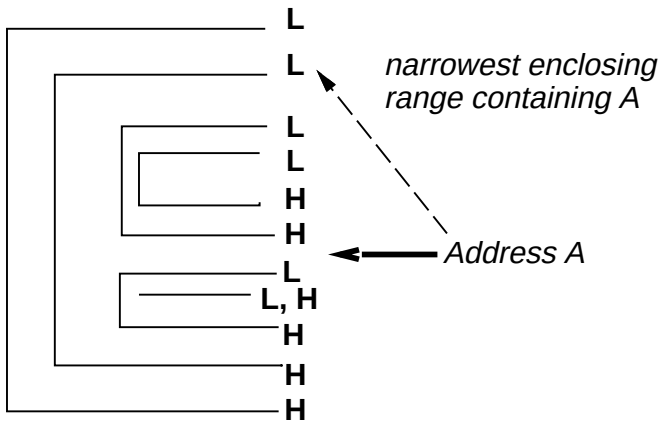

Figure 3: Why longest matching prefix corresponds to narrowest enclosing range, and why each range in the modified binary search table maps to a unique prefix.

large databases can be done in several hundred msec). Essentially, the idea is to add the dotted line pointer shown in Figure 3 to every region.

The final table corresponding to Figure 3 is shown in Figure 4. Notice that with each table entry E, there are two precomputed prefix values. If binary search for address A ends in a failure at E, it is because $A > E$. In that case, we use the $>$ pointer corresponding to E. On the other hand, if binary search for address A ends in a match at E, we use the $=$ pointer.

Notice that for an entry like 101011, the two entries are different. If address A ends up at this point and is greater than 101011, clearly the right prefix is $P2 = 101^*$. On the other hand, if address A ends up at this point with equality, the correct prefix is $P3 = 10101^*$. Intuitively, if an address A ends up equal to the high point of a range R, then A fall within the range R; if A ends up greater than the high point of range R, then A falls within the smallest range that encloses range R.

Our description is somewhat different from the description of our scheme in [18]. We use two pointers per entry instead of just one pointer. The description of our scheme in [18] suggests padding every address by an extra bit; this avoids the need for an extra pointer but it makes the implementation grossly inefficient because it works on 33 bit (i.e., for IPv4) or 129 bit (i.e., for IPv6) quantities. If there are less than 2^{16} different choices of next hop, then the two pointers can be packed into a 32 bit quantity, which is probably the minimum storage needed.

4.2 Insertion into a Modified Binary Search Table

The simplest way to build a modified binary search table from scratch is to first sort all the entries, after marking each entry as a high or a low point of a range. Next, we process the entries, using a stack, from the lowest down to the highest to precompute the corresponding best matching prefixes. Whenever we encounter a low point (**L** in the figures), we stack the corresponding prefix; whenever we see the corresponding high point, we unstack the prefix. Intuitively, as we move down the table, we are keeping track of the currently active ranges; the top of the stack keeps track of the innermost active range. The prefix on top of the stack can be used to set the $>$ pointers for each entry, and the $=$ pointers can be computed trivially. This is an $O(N)$ algorithm if there are N prefixes in the table.

One might hope for a faster insertion algorithm if we had to only add (or delete) a prefix. First, we could represent the binary search table as a binary tree in the usual way. This avoids the need to shift entries to make room for a new entry. Unfortunately, the addition of a new prefix can affect the precomputed information in $O(N)$ prefixes. This is illustrated in Figure 5. The figure shows an

							>	=
P1)	1	0	0	0	0	0	P1	P1
P2)	1	0	1	0	0	0	P2	P2
P3)	1	0	1	0	1	0	P3	P3
	1	0	1	0	1	1	P2	P3
	1	0	1	1	1	1	P1	P2
	1	1	1	1	1	1	–	P1

Figure 4: The final modified binary search table with precomputed prefixes for every region of the binary table. We need to distinguish between a search that ends in a success at a given point (= pointer) and search that ends in a failure at a given point (> pointer).

outermost range corresponding to prefix P; inside this range are $N-1$ smaller ranges (prefixes) that do not intersect. In the regions not covered by these smaller prefixes, we map to P. Unfortunately, if we now add Q (Figure 5), we cause all these regions to map to Q, an $O(N)$ update process.

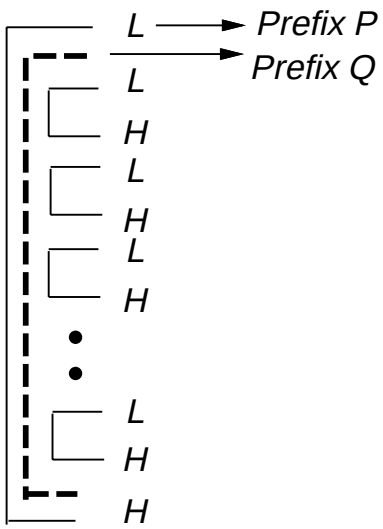

Figure 5: Adding a new prefix Q (dotted line) can cause all regions between an **H** and an **L** to move from Prefix P to Prefix Q.

Thus there does not appear to be any update technique that is faster than just building a table from scratch. Of course, many insertions can be batched; if the update process falls behind, the batching will lead to more efficient updates. We will see shortly that using an initial 16 bit table lookup can decompose the single binary search table into multiple (but smaller) binary search tables; since the decomposed binary search tables are smaller, this decomposition not only speeds up search time but also speeds up insertion. We defer estimates of insertion times to Section 7.5. First, we describe two crucial optimizations that considerably speed up the basic binary search scheme.

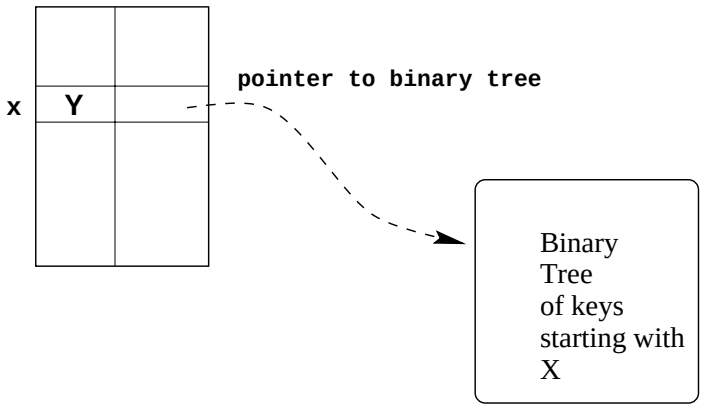

Figure 6: The array element with index X will have the best matching prefix of X (say Y) and a pointer to a binary tree/table of all prefixes that have X as a prefix.

5 Precomputed 16 bit prefix table

We can improve the worst case number of memory accesses of the basic binary search scheme with a precomputed table of best matching prefixes for the first Y bits. The main idea is to effectively partition the single binary search table into multiple binary search tables for each value of the first Y bits. This is illustrated in Figure 6. We choose $Y = 16$ for what follows as the table size is about as large as we can afford, while providing maximum partitioning.

Without the initial table, the worst case possible number of memory accesses is $log_2 N + 1$, which for large databases could be 16 or more memory accesses. For a sample database, this simple trick of using an array as a front end reduces the maximum number of prefixes in each partitioned table to 336 from the maximum value of over 38,000.

The best matching prefixes for the first 16 bit prefixes can be precomputed and stored in a table. This table would then have $Max = 65536$ elements. For each index X of the array, the corresponding array element stores best matching prefix of X. Additionally, if there are prefixes of longer length with that prefix X, the array element stores a pointer to a binary search table/tree that contains all such prefixes. Insertion, deletion, and search in the individual binary search tables are identical to the techniques described earlier in Section 4.

Figure 7 shows the distribution of the number of keys that would occur in the individual binary search trees for a publically available IP backbone router database [12] after going through an initial 16 bit array. The largest number of keys in any binary table is found to be 336, which leads to a worst case of 10 memory accesses.

Precomputation of the initial array: While the initial array makes search faster, it requires that every array element I be tagged with the longest matching prefix corresponding to I. Whenever information in the database changes (e.g., because of a prefix insertion or deletion), the precomputed longest matching information for many (if not all) array elements may need to be recalculated.

The best matching prefixes of the elements of the array can be easily precomputed using a form of prefix expansion ([24]). The simplest conceptual idea is to expand all prefixes, starting with the lowest length prefixes, one bit at a time until they reach 16 bits. A prefix like P can be expanded into $P0$ and $P1$. However, if there is already a higher length prefix corresponding to either $P0$ or $P1$ then we stop the expansion process as the longer length prefix captures the (expanded) lower length prefix. At the end we have a list of 16 bit elements tagged with the appropriate prefix; this can be used to fill the table.

The above idea takes $O(Max)$ time and requires storage for Max list elements in addition to the

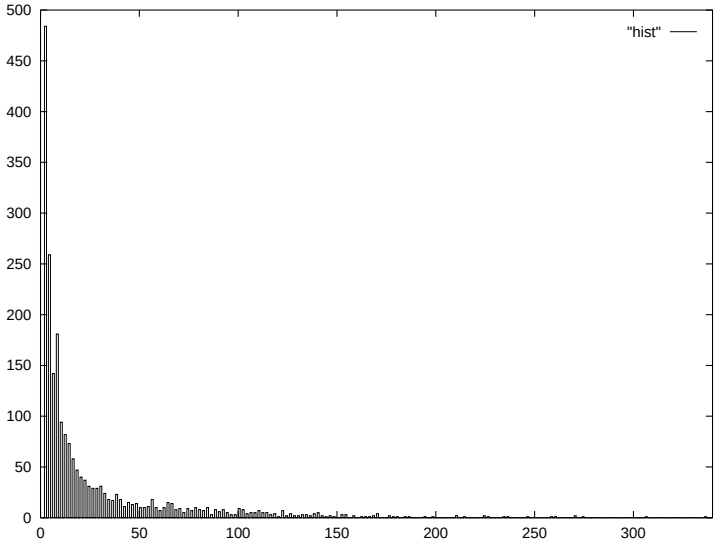

Figure 7: For each 16 bit prefix X, let $N(X)$ be the number of prefixes that have X as a prefix. The histogram shows the distribution of N(X) for the Mae-East NAP Routing database [12]. The horizontal axis represents $N(X)$ and the vertical axis represents the number of tables with a given value of $N(X)$. Thus the peak of the histogram says that there are 484 binary search tables with only 2 keys. There is only 1 binary search table with the worst case number of 336 keys.

initial array. Another possible scheme is to build a binary search table (see Section 4 and Figure 4) on the prefixes of lengths no greater than 16. Then we can proceed through the array elements and use the binary search table to compute the best matching prefix of each array element. This would require $Max \log N$ time which would be too slow.

However, since the final binary search table (see Figure 4) is essentially a sorted table of range endpoints we do not need to do a binary search for each element. We only need to simultaneously scan the array and the binary search table. We use two pointers, a pointer into the array (that starts at the lowest element of the array) and a pointer into the binary search table (that starts at the lowest element of the table). The current range is indicated by the current binary search entry and the following entry. The algorithm keeps incrementing the array pointer, updating each array element using the next hop corresponding to the current range; as soon as the array pointer goes beyond the current range, the algorithm increments the binary search table pointer. The algorithm terminates when all array elements have been scanned. Thus the algorithm terminates after $O(N + Max)$ time. We stress again that this precomputation only needs to be done when prefixes are inserted and deleted, and not when a search is done.

6 Multiway binary search: Exploiting the cache line

Todays processors have wide cache lines. The Intel Pentium Pro has a 64 bit data bus and a cache line size of 32 bytes. Main memory is usually arranged in a matrix form, with rows and columns. Accessing data given a random row address and column address has an access time of 50 to 60 nsec. However, using SDRAM or RDRAM, filling a cache line of 32 bytes (which is a burst access to 4 contiguous 64 bit DRAM locations), is much faster than accessing 4 random DRAM locations. When accessing a burst of contiguous columns in the same row, while the first piece of data would be available only after 60 nsec, further columns would be available much faster. SDRAMs (Synchronous DRAMs) are available (at $205 for 8MB [22]) that have a column access time of 10 nsec. Timing diagrams of micron SDRAMs are available through [13]. RDRAMs [20] are available that can fill a cache line in 101 nsec. Detailed

descriptions of main memory organization can be found in [9].

The significance of this observation is that it pays to restructure data structures to improve locality of access. To make use of the cache line fill and the burst mode, keys and pointers in search tables can be laid out to allow multiway search instead of binary search. This effectively allows us to reduce the search time of binary search from $\log_2 N$ to $\log_{k+1} N$, where k is the number of keys in a search node. The main idea is to make k as large as possible so that a single search node (containing k keys and $2k+1$ pointers) fits into a single cache line. If this can be arranged, an access to the first word in the search node will result in the entire node being prefetched into cache. Thus the accesses to the remaining keys in the search node are much cheaper than a memory access.

We did our experiments using a Pentium Pro; the parameters of the Pentium Pro resulted in us choosing $k = 5$ (i.e, doing a six way search). For our case, if we use k keys per node, then we need $2k+1$ pointers, each of which is a 16 bit quantity. (This is because we need a pointer for the ranges between keys as well as a pointer for exact matches with each key.) Thus in 32 bytes we can place 5 keys and hence can do a 6-way search. For example, if there are keys k1..k8, a 3-way tree is given in Figure 8. The initial full array of 16 bits followed by the 6-way search is depicted in Figure 9.

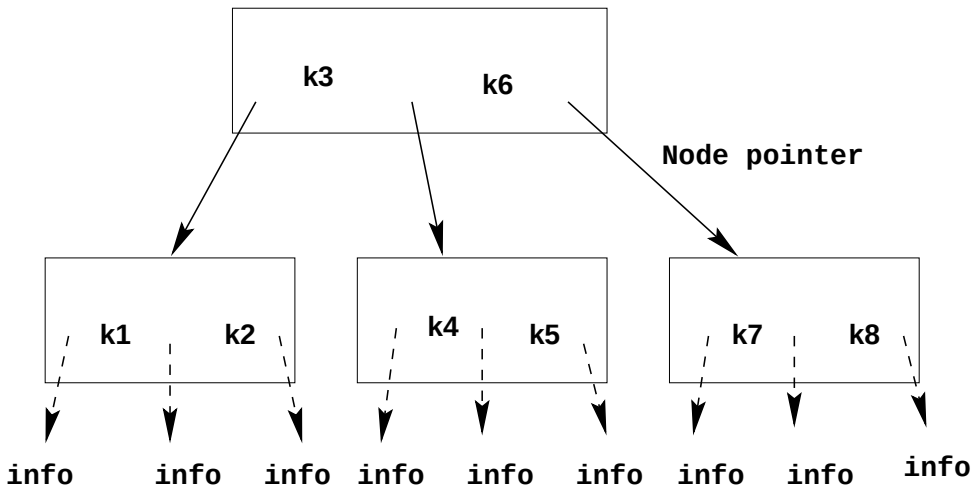

Figure 8: 3-way tree for 8 keys

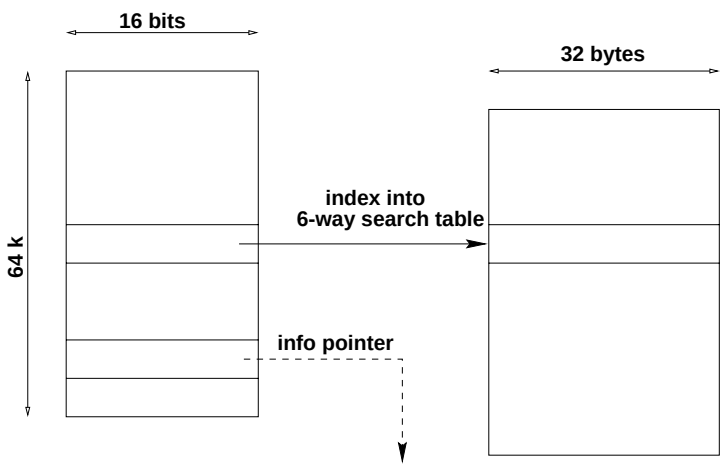

Figure 9: The initial 16 bit array, with pointers to the corresponding 6-way search nodes.

Since the worst case (for the Mae East database after using a 16 bit initial array) has 336 entries in a table, this leads to a worst case of 4 memory accesses (since $6^4 = 1296$ takes only 4 memory accesses when doing a 6-way search) in addition to the initial table lookup.

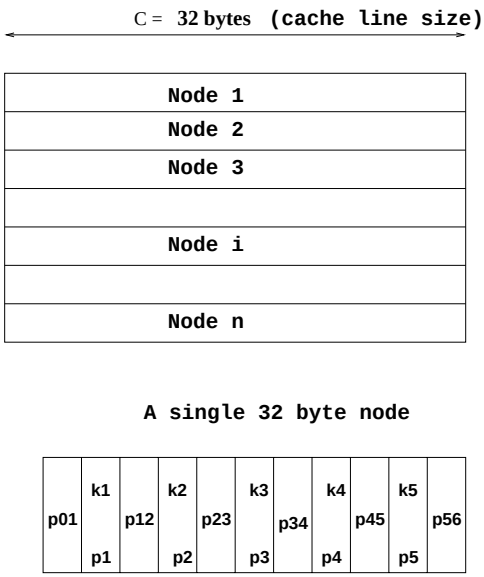

Figure 10: The structure of the 6-way search node. There are k keys and $2k+1$ pointers.

Each node in the 6-way search table has 5 keys k_1 to k_5, each of which is 16 bits. There are *equal to* pointers p_1 to p_5 corresponding to each of these keys. Pointers p_{01} to p_{56} correspond to ranges demarcated by the keys. This is shown in Figure 10 . Among the keys we have the relation $k_i \leq k_{i+1}$. Each pointer has a bit which says it is an *information* pointer or a next node pointer.

6.1 Search

The following search procedure can be used for both IPv4 and IPv6 (see next section for details on the IPv6 search procedure).

1. Index into the first 16 bit array using the first 16 bits of the address.

2. If the pointer at the location is an *information* pointer, return it. Otherwise enter the 6-way search with the initial node given by the pointer, and the key being the next 16 bits of the address.

3. In the current 6-way node locate the position of the key among the keys in the 6-way node. We use binary search among the keys within a node. If the key equals any of the keys key_i in the node, use the corresponding pointer ptr_i. If the key falls in any range formed by the keys, use the pointer $ptr_{i,i+1}$. If this pointer is an *information* pointer, return it; otherwise repeat this step with the new 6-way node given by the pointer.

   In addition, we allow multicolumn search for IPv6 (see Section 8) as follows. If we encounter an *equal to* pointer, the search shifts to the next 16 bits of the input address. This feature can be ignored for now and will be understood after reading Section 8.

As the data structure itself is designed with a node size equal to a cache line size, good caching behavior is a consequence. All the frequently accessed nodes will stay in the cache. To reduce the worst case access time, the first few levels in a worst case depth tree can be cached.

7 Measurements and Comparison for IPv4

We used a 200 Mhz *Pentium Pro* [11] based machine (cost under 5000 dollars) with a 8 KByte four-way set-associative primary instruction cache and a 8 KByte dual ported two-way set associative primary data cache. The L2 cache is 256 KBytes of SRAM that is coupled to the core processor through a full clock-speed, 64-bit, cache bus. We reproduce the system architecture and the interface to the memory subsystem for the Intel architecture [10] in Figure 11.

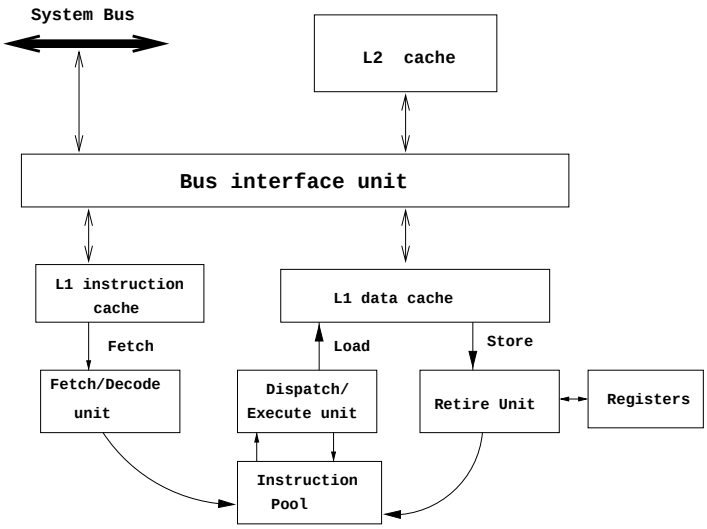

Figure 11: The processing units in the Pentium Pro microarchitecture and their interface with the memory subsystem.

We used practical routing tables (primarily Mae East with over 38000 entries, all of which were obtained from [12]) for our experiments. Our tables list results for the BSD Patricia Trie implementation (extracted from the BSD kernel into user space), basic binary search, binary search with a 16 bit initial array, and 6-way search.

We performed several experiments. First, we quantified the speed of the the new algorithms. We did so in two ways. First, we used repeated lookup to a number of IP addresses to determine worst case times. Second, we computed average lookup times assuming each IP address was equally likely. We also computed storage and table building costs for the schemes. We also used the table building costs to estimate the worst case costs for incremental insertion and deletion in order to determine whether updates can be done quickly. Finally, we use these numbers to compare our scheme with other schemes reported in the literature.

7.1 Worst Case Lookup Times

After adding the routes in the route database [12], several IP addresses were generated and a lookup performed 10 million times for each such address. The lookup time for an address was then calculated by taking the total elapsed time (measured by reading the clock before and iteration) dividing by the number of iterations (10 million). As one would expect, the numbers vary depending on the number of levels of the binary tree that must be traversed before the prefix of the address is resolved.

For example, in the 6-way search scheme, any address whose prefix is less than 16 bits will be resolved in 1 array lookup (depth 0); an address will take 1 more access to a 6-way node to resolve (depth 1) if the number of keys starting with the 16 bit prefix of the address is less than 5; others will take accesses to a second 6-way node if the number of keys starting with the 16 bit prefix of the address is less than

35 (depth 2); and so on. Recall that as we are doing 6-way search, the depth of an address A is equal to $\lceil \log_6 n(A) \rceil$, where $n(A)$ is the number of keys that that start with the 16 bit prefix of address A. Since we have seen that $n(A) \leq 336$ for Mae East, the depth of an address is no greater than 4 in our experiments. Table 3 describes the distribution of the $n(A)$.

Since the worst case lookup times varied with depth, we picked 7 address lookups to display in Table 2. For example, the first two rows corresponds to Depth 4 addresses, while the last two correspond to Depth 0 addresses. The third, fourth, and fifth rows correspond to Depth 1, Depth 2, and Depth 3 addresses respectively. Table 2 compares the lookup times for these 7 addresses for each of four schemes: Patricia (baseline), basic binary search, binary search with the initial array, and 6-way search. The time for Patricia does vary a great deal (from around 1.5 microsec to 2.5 microsec). The search time for basic binary search varies only a little (from around 1 to 1.3 microsec). The time for binary search with the initial table clearly depends on the depth of the address. The lower rows have very small lookup times (90 nsec), but the upper rows have larger times (e.g. 730 nsec) corresponding to a binary search of up to 336 keys.

Patricia	Basic Binary Search	16 bit table + binary search	16 bit table+6-way Search
Time (nsec)	Time (nsec)	Time (nsec)	Time(nsec)
1530	1175	730	490
1525	990	620	490
1450	1140	470	390
2585	1210	400	300
1980	1440	330	210
810	1220	90	95
1170	1310	90	90

Table 2: Time taken for single address lookup on a Pentium pro. Several addresses were searched and the search times noted. Shown in the table are addresses picked to illustrate the variation in time of the 16 bit initial table+6-way search method. Thus the first two rows correspond to the maximum depth of the search tree while the last two rows correspond to the minimum depth (i.e, no prefixes in search table).

The time taken for the 6-way search varies in the same way. The lookup times for depth 0 nodes are nearly identical for 6-way and binary search + 16 bit table because both involve only a lookup of the initial array. However, the times at each depth greater than depth 0 are much lower than that of using binary search + the 16-bit table because of the benefits of using 6-way versus 2-way search. Recall that we have defined depth of an address with respect to the 6-way scheme. An address at depth 4 in the 6-way scheme will be at a greater depth in the other binary search schemes.

Notice also that in Table 2 that the lookup times for 6-way search increase by around 100 nsec for each memory access, yielding a lookup time of around $100 * (D(A) + 1)$ for an address A, where $D(A)$ is the depth of address A in the 6-way scheme. This fits in well with our intuition that the primary measure of lookup speed is the number of memory accesses, and that the speed of a random memory access (including memory costs, processing costs and costs of cache misses) in the Pentium is around 100 nsec.

Instruction count: The static instruction count using a full 16 bit initial table followed by a 6-way search table is less than 100 instructions on the Pentium Pro. We also note that the gcc compiler uses only 386 instructions and does not use special instructions available in the Pentium Pro, using which it may be possible to further reduce the number of instructions.

7.2 Average Lookup time

For the Mae-East database [12], Table 3 gives the number of 16 bit prefix slots that lead to 6-way trees that would take a maximum of 1 memory access, 2 memory accesses, and so on. Assuming that all IP addresses are equally likely, the data in Table 3 allows us to easily calculate the average time for a lookup. Since each 16 bit slot is equally likely, we can calculate a weighted average in which we weight the lookup times for each depth d by the fraction of 16 bit slots that have depth d.

Number of keys in tree	No. of 16 bit prefixes leading to trees containing the number of keys in this range	Measured time when key in tree with total number of keys in this range
0	63414	95
1..5	743	210
6..35	916	300
36..215	449	390
215..max=336	13	490

Table 3: The table shows that 63414 of the 16 bit array elements require no memory accesses beyond the initial array access. It goes on to show that 743 array elements lead to trees having only 1..5 keys and hence only need 1 memory access in a 6-way tree. Similarly with 36..215 keys, we need at most 3 accesses.

The average, assuming random accesses to all possible IP addresses, calculated based on Table 3 is 100 nsec for the 16 bit+6-way search. For the 16 bit+binary search, a similar calculation yields 110 nsec. This is not very different from the 6-way search because a huge number of the initial array entries do not have any longer prefixes; thus a huge fraction (in both algorithms) of addresses takes only 90 nsec to resolve.

However, in practice, the distribution of addresses received by a router is unlikely to be random. In order to claim wire speed routing, a router must be able to repeatedly lookup packets that have worst case lookup times. Since worst case search time is an important consideration for IP lookups, 6-way search appears to be a good choice.

7.3 Memory Requirements and Worst case times

We have seen that 6-way search improves the worst case time to 490 seconds. What storage penalty is paid for this extra speed? To answer this question, we used the Mae East database and compared the three schemes (Basic Binary Search, Binary Search with 16 bit initial table, and Binary Search with 16 bit initial table plus 6 way search) in Table 4. We also used the Patricia trie code (that we extracted from the BSD code) as a baseline comparison.

Table 4 shows that 6-way search takes a worst case time of 490 nsec (compared to 2585 nsec for Patricia and 1310 nsec for ordinary binary search) and, surprisingly, has a memory usage (0.95 Mbytes for search structure) that is less than Patricia (3.2 Mbytes) and even less than for basic binary search (1 Mbyte). The fact that the memory for the 6-way search is less than that for basic binary search appears counter-intuitive because we have added a large initial array in the 6-way search algorithm. The explanation is that while the keys used in the regular binary search are 32 bits and the pointers involved are also 32 bits, in the 16 bit table followed by the 6-way search case, both the keys and the pointers are 16 bits. This is possible because the initial 16 bit lookup allows us to work with 16 bit quantities after the lookup. The use of smaller size quantities is only partially offset by the size of the added initial table.

	Patricia	Basic Binary Search	16 bit initial table + binary search	16 bit initial table + 6-way search
Total mem req. for building the structure(MB)	3.2	3.6	1	1
Memory for searchable structure (MB)	3.2	1	0.75	0.95
Worst case search time(nsec)	2585	1310	730	490
Comparison with Patricia scheme (Worst case faster by:)	1	2	3.5	5

Table 4: Memory Requirement and Worst case time

From Table 4 we can see that the initial array improves the performance of the binary search from a worst case of 1310 nsec to 730 nsec; multiway search further improves the search time to 490 nsec. Thus both optimizations (initial array, multiway search) appear to be worthwhile.

7.4 Variation of Lookup Times and Memory with Database Size

There are at least three other databases in [12] besides the large Mae East database. The Mae East database may be characteristic of of a router used by a large service provider. The other databases (Mae West, Pac, and Paix) may reflect more common database sizes. In particular, the smallest Paix database may be an accurate model of a large enterprise router in a campus or a company.

Table 5 shows the number of prefixes, the worst case search times, and the memory requirements for the four databases using 6-way search. Notice that the storage goes down from 950 kbytes to around 265 kbytes for the smallest database (small enough to fit into cache). Also notice that the worst case lookup time goes down (though not dramatically) from 490 nsec (MaeEast) to 300 nsec (Paix). This is because the worst case tree depth often decreases as we reduce the number of prefixes.

	MaeEast	MaeWest	Pac	Paix
Number of Prefixes	38816	14072	3812	713
Worst Case Search Time(nsec)	490	490	390	300
Memory Required (kbytes)	950	512	530	265
Build Time (seconds)	5.8	2.16	0.78	0.35

Table 5: Memory Requirement, Worst Case Time, and Build Times for four prefix databases using 6-way search

We note that for the Paix database the storage is completely dominated by the initial 16-bit array (250 kbytes out of a total of 265 kbytes). This suggests that for such small databases we should use a smaller initial array (e.g., 12 bits) which can reduce storage greatly at the potential cost of a small increase in worst case tree depth. Picking the optimal (or near optimal) initial array size for each database is an interesting possibility that we have not yet explored.

7.5 Building and Incremental Insertion Times

So far we have ignored insertion and table building times. The heavy use of precomputation in the basic binary search scheme suggests that insertion times can be quite slow. While many existing schemes suggest that lookup times are far more important than insertion times[5, 27], the existence of rapid BGP updates in the backbone makes rapid insertion (and deletion) an issue at least for backbone routers (though not for enterprise routers).

Table 5 also lists the time to build the complete search structure for 6-way search for each of the four databases. We made no attempt to optimize the table building process; it seems very likely that the numbers can be improved. From the table, we see that the cost of building a complete table can range from 5.8 seconds (for Mae East) to 350 msec (for Paix). At first glance, this seems to preclude the use of binary search for large backbone routers because of the potential need for rapid insertion times (10-100 msec). However, incremental insertion times can be much smaller even for large databases like Mae-East.

The worst case insertion/deletion cost can be estimated as follows. For prefixes of size less than 16 bits, we may have to scan the entire initial table to update any table entries that may be affected by this prefix's insertion or deletion. This will require potentially scanning 2^{16} trie nodes entries. On a Pentium with a *Write Back* cache, while the first word in a cache line would cost 100 nanoseconds for the read, the writes would be only to cache and the entire cache line will take at most 200 nanoseconds to update. A single cache line holds 10 entries, so at 20 nsec/entry for 2^{16} entries we get 1.25 millisec.

For a prefix P of greater than 16 bits, we simply have to insert/delete P from the binary search tree corresponding to the 16 bit prefix of P. Our results indicate that even for Mae East, the worst case number of keys corresponding to a 16 bit initial prefix is at most 336. Thus we can estimate the insertion/deletion time (looking at the worst case building time for the Paix database) to be no more than 352 msec. In practice, the average insertion time should be much faster because i) the individual binary search trees are typically much smaller than the worst case of 336 ii) our estimates are very conservative iii) we have made no effort to optimize insertion costs. An average insertion time of around 20 msec seems easily achievable even for very large databases like Mae West. Note that the technique of using an initial array actually speeds up incremental insertion because it replace a single large binary tree (into which insertion/deletion can be expensive) by several smaller binary trees (into which insertion/deletion is cheap).

7.6 Comparing 6-way Search to Other Algorithms

So far we have been using the Patricia Trie code extracted from NetBSD as a baseline comparison. However, this is a somewhat unfair comparison for the following reasons. First, the NetBSD code [23] uses backtracking which slows down the code, and can be removed by simple optimizations. Second, the NetBSD code performs other functions besides basic IP lookup, which may increase lookup times and increase memory usage. However, in the last two years there have been a number of other algorithms that have attracted considerable attention. These include binary search on hash tables [27], the Lulea scheme for compressed multibit tries [5], the Level Compressed Trie scheme (LC tries) [14], and multibit tries using prefix expansion [24].

While we have not implemented all these schemes, we were fortunate in that the two schemes we have not implemented (LC tries and Lulea scheme) have been implemented by their inventors on a comparable Pentium platform using the same Mae East database. This allows us to do a reasonable comparison of all 5 schemes. We borrow the numbers for binary search on hash tables and multibit tries from our earlier paper [24]. The numbers for LC tries are taken from [14], and the numbers for the Lulea scheme are taken from [5].

The comparison is not without flaws; since the Mae East database is constantly changing, and the

reported measurements were done based on different snapshots, the number of prefixes is not exactly the same (but is roughly 40,000 prefixes) in all measurements. The Lulea scheme was implemented and measured on a 200 MHz Pentium, the LC trie on a 133 MHz Pentium, and the multibit trie scheme using a 300 MHz Pentium. In those cases, we have (somewhat generously) scaled the schemes that proportionately to a common 300 Mhz clock speed used in [24]. This is rather generous because scaling up the clock speed does not necessarily speed up lookup times by the same factor because memory access times do not speed up with the faster clock. Despite these limitations, we believe such a comparison is still useful as an initial study.

Table 6 describes a comparison of the various schemes in terms of lookup times and memory usage. Many implementors believe that while the absolute worst case times are interesting, since almost all prefixes are 24 bits or less in length, the worst case time to resolve 24 bit prefixes is an important metric. This figure is especially favorable to trie schemes that can terminate faster (i.e., at a smaller trie depth) when the best matching prefix is 24 bits or less. Thus we also have a column corresponding to the worst case lookup times for 24 bit prefixes. The numbers for Patricia trie and 6-way search have been scaled to 300 MHz which is why they differ from the numbers reported earlier. The numbers for Lulea and LC tries have also been scaled similarly from 133 Mhz and 200 Mhz respectively.

	Average (24 bit prefix (nsec)	Worst case (nsec)	Memory required for Mae East database (KBytes)
Patricia trie	1000	1650	3262
6-way search on prefixes	330	330	950
Binary search on hash tables	250	650	1600
Lulea scheme	349	409	160
Multibit trie	180	236	640
LC tries	1000	-	700

Table 6: Comparing the average lookup times (when the address matches a prefix of 24 bits or less), worst case lookup time, and storage for 6-way search versus the five other IP lookup schemes that we know of. Note that numbers measured on 200 and 133 MHz Pentiums have been projected to 300 MHz.

We note that the LC trie and Patricia trie implementations are fairly slow but the memory needed for LC tries is much smaller than Patricia tries. (The worst case times for LC tries are not reported in [14].) The memory for the Lulea scheme is easily the smallest (160 kbytes) and multibit tries have the fastest lookup times (180 nsec). 6-way search does reasonably well, with 950 kbytes of memory and a lookup time (projected to 300 MHz) of 330 nsec.

An important factor not shown in the table is the cost of insertion and deletion of prefixes. This is not described in the papers on LC tries, binary search on hash tables, and Lulea compressed tries. Thus a complete comparison is not possible. Since the Lulea, LC trie, and binary search on hash table schemes all potentially require changing the complete data structure on a prefix insertion/deletion, they are likely to have slow insertion/deletion times. The multibit trie scheme estimates insertion/deletion at 2.5 msec. Recall that we estimated an insertion deletion time of around 350 msec for 6-way search.

Based on the comparison, the Lulea scheme appears to be the candidate of choice if small memory (useful especially if SRAM is used for memory and cost is an issue) is the dominant factor. The multibit trie scheme seems to have the best lookup and insertion times, while allowing reasonably small memory.

However, we note that the memory usage of the multibit trie schemes is only small when a complicated dynamic programming algorithm is used to lay out the variable stride trie [24]. This algorithm may need to be run only rarely but adds complexity and can slow down insertion and deletion when it is run.

Recall from the introduction that there are two types of routers: backbone and enterprise routers. Enterprise routers have small databases and have less stringent requirements in terms of route changes. For such routers and small database, 6-way search is simple and fast (see earlier results for the Paix database) and has adequate insertion and deletion times. Thus we believe that 6-way search is a reasonable candidate for implementation in an Enterprise router.

8 Using Multiway and Multicolumn Search for IPv6

In this section we describe the problems of searching for identifiers of large width (e.g., 128 bit IPv6 address or 20 byte OSI addresses). We first describe the basic ideas behind multicolumn search and then proceed to describe an implementation for IPv6 that uses both multicolumn and multiway search. We then describe sample measurements using randomly generated IPv6 addresses.

8.1 Multicolumn Binary Search of Large Identifiers

The scheme we have described in earlier sections can be implemented efficiently for searching 32 bit IPv4 addresses. Unfortunately, a naive implementation for IPv6 can lead to inefficiency. Assume that the word size M of the machine implementing this algorithm is 32 bits. Since IPv6 addresses are 128 bits (4 machine words), a naive implementation would take $4 \cdot \log_2(2N)$ memory accesses. For a reasonable sized table of around 32,000 entries this is around 60 memory accesses! 6-way search does not help this problem (entries after the initial 16 bit table need only be 112 bits long, but that still requires 4 machine words.)

In general, suppose each identifier in the table is W/M words long (for IPv6 addresses on a 32 bit machine, $W/M = 4$). Naive binary search will take $W/M \cdot \log N$ comparisons which is expensive. Yet, this seems obviously wasteful. If all the identifiers have the same first $W/M - 1$ words, then clearly $\log N$ comparisons are sufficient. We show how to modify Binary Search to take $\log N + W/M$ comparisons. It is important to note that this optimization we describe can be *useful for any use of binary search on long identifiers*, not just the best matching prefix problem.

The strategy is to work in columns, starting with the most significant word and doing binary search in that column until we get equality in that column. At that point, we move to the next column to the right and continue the binary search where we left off. Unfortunately, this does not quite work.

In Figure 12, which has $W/M = 3$, suppose we are searching for the three word identifier BMW (pretend each character is a word). We start by comparing in the leftmost column in the middle element (shown by the arrow labeled 1). Since the B in BMW matches the B at the arrow labeled 1 we move to the right (not shown) and compare the M in BMW with the N in the middle location of the second column. Since $N < M$, we do the second probe at the quarter position of the second column. This time the two M's match and we move rightward and we find W, but (oops!) we have found AMW, not BMW which we were looking for.

The problem is caused by the fact that when we moved to the quarter position in column 2, we assumed that all elements in the second quarter begin with B. This assumption is false in general. The trick is to add state to each element in each column which can contain the binary search to stay within a guard range.

In the figure, for each word like B in the leftmost (most significant) column, we add a pointer to the the range of all other words that also contain B in this position. Thus the first probe of the binary search for BMW starts with the B in BNX. On equality, we move to the second column as before. However,

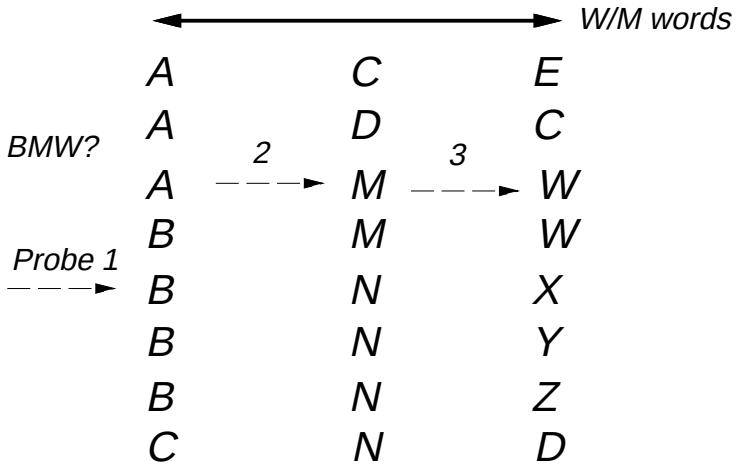

Figure 12: Binary Search by columns does not work when searching for BMW

we also keep track of the guard range corresponding to the B's in the first column. The guard range (rows 4 through 6) is stored with the first B we compared.

Thus when we move to column 2 and we find that M in BMW is less than the N in BNX, we attempt to half the range as before and try a second probe at the third entry (the M in AMW). However the third entry is lower than the high point of the current guard range (4 through 6). So without doing a compare, we try to halve the binary search range again. This time we try entry 4 which is in the guard range. We get equality and move to the right, and find BMW as desired.

In general, every multiword entry $W_1, W_2, \ldots, W_n$ will store a guard range with every word. The range for W_i, points to the range of entries that have $W_1, W_2, \ldots, W_i$ in the first i words. This ensures that when we get a match with W_i in the i-the column, the binary search in column $i+1$ will only search in this guard range. For example, the N entry in BNY (second column) has a guard range of $5-7$, because these entries all have BN in the first two words.

The naive way to implement guard ranges is to change the guard range when we move between columns. However, the guard ranges may not be powers of 2, which will result in expensive divide operations. A simpler way is to follow the usual binary search probing. If the table size is a power of 2, this can easily be implemented. If the probe is not within the guard range, we simply keep halving the range until the probe is within the guard. Only then do we do a compare.

The resulting search strategy takes $\log_2 N + W/M$ probes if there are N identifiers. The cost is the addition of two 16 bit pointers to each word. Since most word sizes are at least 32 bits, this results in adding 32 bits of pointer space for each word, which can at most double the memory usage. Once again, the dominant idea is to use precomputation to trade a slower insertion time for a faster search.

We note that the whole scheme can be elegantly represented by a binary search tree with each node having the usual $>$ and $<$ pointers, but also an $=$ pointer which corresponds to moving to the next column to the right as shown above. The subtree corresponding to the $=$ pointer naturally represents the guard range.

8.2 Using Multicolumn and Multiway Search for IPv6

In this section we explore several possible ways of using the k-way search scheme for IPv6. With the 128 bit address, if we used columns of 16 bits each, then we would need 8 columns. With 16 bit keys we can do a 6-way search. So the number of memory accesses in the worst case would be log_6 (2N) + 8. For N

around 50,000 this is 15 memory accesses. In general, if we used columns of M bits, the worst case time would be $log_{k+1}N + W/M$ where $W = 128$ for IPv6. The value of k depends on the cache linesize C. Since k keys requires $2k + 1$ pointers, the following inequality must hold. If we use pointers that are p bits long,

$kM + (2k + 1) * p \leq C$

For the Intel Pentium pro, C is 32 bytes, i.e. $32 * 8 = 256$ bits. If we use $p = 16$,

$k(M + 32) \leq 240$, with the worst case time being $log_{k+1}N + 128/M$.

In general, the worst case number of memory accesses needed is $T = \lceil (log_{k+1}(2N) \rceil + \lceil (W/m) \rceil$, with the inequality $Mk + (2k + 1)p \leq C$, where

- N is the number of prefixes
- W is the number of bits in the address
- M is the number of bits per column in the multiple column binary search
- k is the number of keys in one node
- C is the cache linesize in bits
- p is the number of bits to represent the pointers within the structure
- T is the worst case number of memory accesses

Fig 13 shows that the W bits in an IP address are divided into M bits per column. Each of these M bits make up a M bit key, k of which must fit in the search node of length C bits along with $2k + 1$ pointers of length p bits.

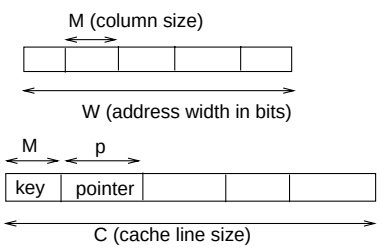

Figure 13: Symbols used in expressing the number of memory accesses needed.

For typical values of N, the number of prefixes, the following table gives the value of the corresponding worst case number of memory accesses.

No. of Prefixes	$M = 16$	$M = 32$	$M = 64$
128	12	10	8
256	13	10	8
1024	14	11	9
50000	17	15	13
100000	17	16	14

Table 7: Worst case number of memory accesses for various values of N (total number of prefixes) and M (number of bits in a column for the multicolumn search), with $W = 128, C = 256$ (32 bytes), and $p = 24$ bits.

However, using an initial 16-bit array, the number of prefixes in a single tree can be reduced. For IPv4 the maximum number in a single tree was 336 for a practical database with N more than 38000 (i.e., the number of prefixes that have the same first 16 bits is 168, leading to 336 keys). For IPv6, with $p = 16$, even if there is an increase of 10 times in the number of prefixes that share the same first 16 bits, for 2048 prefixes in a tree we get a worst case of 9 cache line fills with a 32 byte cache line. For a 64 byte cache line machine, we get a worst case of 7 cache line fills. This would lead to worst case lookup times of less than 800 nsec, which is competitive with the scheme presented in [27].

8.3 Measurements

We generated random IPv6 prefixes and inserted into a k-way search with an initial 16 bit array. Clearly, the use of randomly generated IPv6 address database is no substitute for measurements with actual IPv6 databases. (In particular, random generation will make the use of the 16-bit table appear beneficial. However, it is likely that IPv6 addresses will initially consist of IPv4 addresses padded with a constant prefix. Such an address assignment policy would cause the initial 16 bit table to do badly!). However, until realistic databases become available, randomly generated address databases are a reasonable method to compare algorithms.

From the Mae East database, it was seen that with N about 38000, the maximum number which shared the first 16 bits was about 300, which is about 1 percent of the total number of prefixes. To capture this, when generating IPv6 prefixes, we generated the last 112 bits randomly and distributed them among the slots in the first 16 bit table such that the maximum number that falls in any slot is around 1000. This is necessary because if the whole IPv6 prefix is generated randomly, even with N about 60000, only 1 prefix will be expected to fall in any first 16 bit slot. On a Pentium Pro which has a cache line of 32 bytes, the worst case search time was found to be 970 nsec, using $M = 64$ and $p = 16$.

9 Conclusion

We have described a basic binary search scheme for the longest matching prefix problem. Basic binary search requires two new ideas: encoding a prefix as the start and end of a range, and precomputing the best matching prefix associated with a range. Then we have presented three crucial enhancements: use of an initial array as a front end, multiway search, and multicolumn search of identifiers with large lengths.

We have shown how using an initial precomputed 16 bit array can reduce the number of required memory accesses from 16 to 9 in a typical database; we expect similar improvements in other databases. We then presented the multiway search technique which exploits the fact that most processors prefetch an entire cache line when doing a memory access. A 6-way branching search leads to a worst case of 5 cache line fills in a Pentium Pro which has a 32 byte cache line. We presented measurements for IPv4. Using a typical database of over 38,000 prefixes we obtain a worst case time of 490 nsec, an average time of 100 nsec, storage of 0.95 Mbytes, and estimated insertion times of around 350 msec. For smaller databases (less than 1000 prefixes), we obtain search times of around 300 nsec and storage of 265 kbytes. The simplicity of the scheme makes it attractive for use in enterprise routers that have small routing tables.

For IPv6 and other long addresses, we introduced multicolumn search that avoided the multiplicative factor of W/M inherent in basic binary search by doing binary search in columns of M bits, and moving between columns using precomputed information. We have estimated that this scheme potentially has a worst case of 7 cache line fills for a database with over 50,000 IPv6 prefixes database.

For future work, we are considering the problem of using different number of bits in each column of the multicolumn search. We are also considering the possibility of laying out the search structure to make use of the page mode load to the L2 cache by prefetching. Another possible extension is to retrofit our

Pentium Pro with an SDRAM or RDRAM to improve cache loading performance; this should allow us to obtain better measured performance.

References

[1] Ascend. Ascend GRF IP Switch Frequently Asked Questions. http://www.ascend.com/299.html#15.

[2] Scott Bradner. Next Generation Routers Overview. Proceedings of Networld Interop 97.

[3] Internet II. Big Fast Routers: Multi-Megapacket Forwarding Engines for Internet Routing. Proceedings of Networld Interop 97.

[4] Anthony J. Bloomfeld NJ McAuley, Paul F. Lake Hopatcong NJ Tsuchiya, and Daniel V. Rockaway Township Morris County NJ Wilson. Fast Multilevel heirarchical routing table using content-addressable memory. U.S. Patent serial number 034444.

[5] Andrej Brodnik, Svante Carlsson, Mikael Degermark, and Stephen Pink. Small Forwarding Table for Fast Routing Lookups. *Proceedings ACM SIGCOMM 97*, October 1997.

[6] Digital Electronics Corporation. The dec alpha. http://www.dec.com.

[7] S Deering and R Hinden. Internet protocol, version 6 (ipv6) specification rfc 1883. http://ds.internic.net/rfc/rfc1883.txt.

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

[9] Hennessey and Patterson. *Computer Architecture A quantitative approach, 2nd Edn.* Morgan Kaufmann Publishers Inc, 1996.

[10] Intel. Intel Architecture Software developer's Manual, Vol 1: Basic Architecture. http://www.intel.com/design/litcentr/litweb/pro.htm.

[11] Intel. Pentium Pro. http://pentium.intel.com/.

[12] Merit. Routing table snapshots. ftp://ftp.merit.edu/statistics/ipma.

[13] Micron. Micron Technology Inc. http://www.micron.com/.

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

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

[16] McGray Massachussetts Institute of Technology. Internet Growth Summary. http://www.mit.edu/people/mkgray/net/internet-growth-summary.html.

[17] Intel. The pentium processor. http://www.pentium.com.

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

[19] Tong-Bi Pei and Charles Zukowski. Putting Routing Tables in Silicon. *IEEE Network Magazine*, January 1992.

[20] Rambus. Rdram. http://www.rambus.com/.

[21] Y Rechter and T Li. A border gateway protocol 4 (bgp-4) rfc 1771. http://ds.internic.net/rfc/rfc1771.txt.

[22] SimpleTech. Simple Technology Inc. http://www.simpletech.com/.

[23] Keith Sklower. A Tree-Based Routing Table for Berkeley Unix. Technical report, University of California, Berkeley.

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

[25] W. Richard Stevens and Gary R Wright. *TCP/IP Illustrated, Volume 2 The Implementation.* Addison-Wesley, 1995.

[26] Torrent. Torrent systems, inc. http://www.torrent.com.

[27] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. *Proceedings ACM SIGCOMM 97*, October 1997.