Faster IP Lookups using Controlled Prefix Expansion

V. Srinivasan George Varghese*

Department of Computer Science
Washington University at St. Louis
cheenu@dworkin.wustl.edu varghese@askew.wustl.edu

Abstract

Internet (IP) address lookup is a major bottleneck in high performance routers. IP address lookup is challenging because it requires a *longest matching prefix* lookup. It is compounded by increasing routing table sizes, increased traffic, higher speed links, and the migration to 128 bit IPv6 addresses. We describe how IP lookups can be made faster using a new technique called *controlled prefix expansion*. Controlled prefix expansion, together with optimization techniques based on dynamic programming, can be used to improve the speed of the best known IP lookup algorithms by at least a factor of two. When applied to trie search, our techniques provide a range of algorithms whose performance can be tuned. For example, with 1 MB of L2 cache, trie search of the MaeEast database with 38,000 prefixes can be done in a worst case search time of 181 nsec, a worst case insert/delete time of 2.5 msec, and an average insert/delete time of 4 usec. Our actual experiments used 512 KB L2 cache to obtain a worst-case search time of 226 nsec, a worst-case worst case insert/delete time of 2.5 msec and an average insert/delete time of 4 usec. We also describe how our techniques can be used to improve the speed of binary search on prefix lengths to provide a scalable solution for IPv6. Our approach to algorithm design is based on measurements using the VTune tool on a Pentium to obtain dynamic clock cycle counts.

1 Introduction

From the present intoxication of the Web to the future promise of electronic commerce, the Internet has captured the imagination of the world. It is hardly a surprise to find that the number of Internet hosts triple approximately every two years[Gra96]. Also, Internet traffic is doubling every 3 months[Tam97]), partly because of increased users, but also because of new multimedia applications. The higher bandwidth need requires faster communication links and faster network routers. Gigabit fiber links are commonplace (MCI and UUNET are upgrading their Internet backbone links to 622 Mbits/sec), and yet the fundamental limits of optical transmission have hardly been approached. Thus the key to improved Internet performance is faster routers. This perceived market opportunity has led to a flurry of startups (e.g., Avici, Juniper,Torrent) that are targeting the Gigabit and Terabit router market.

*Supported by NSF Grant NCR-9628145 and an ONR Young Investigator Award

The three central bottlenecks in router forwarding are lookups, switching, and output scheduling. Switching is well studied, and good solutions (e.g., fast busses, Banyan switches) have been developed for ATM switching. Similarly, most vendors feel that full scale fair queuing[DKS89](a form of output scheduling that guarantees tight delay bounds) is not required for a few years until video and audio usage increase. In the interim, cheaper approximations such as weighted and deficit round robin[SV95] ensure fairness and can easily be implemented. Thus a major remaining bottleneck is fast Internet lookups, the subject of this paper.

The Internet Lookup Problem: Internet address lookup would be simple if we could lookup a 32 bit IP destination address in a table that lists the output link for each assigned Internet address. In this case, lookup could be done by hashing, but a router would have to keep millions of entries. To reduce database size and routing update traffic, a router database consists of a smaller set of prefixes. This reduces database size, but at the cost of requiring a more complex lookup called *longest matching prefix*.

A metaphor can explain the compression achieved by prefixes. Consider a flight database in London. We could list the flights to a thousand U.S. cities in our database. However, suppose most flights to the U.S. hub though Boston, except flights to California that hub through LA. We can reduce the flight database from thousand entries to two prefix entries (USA.* $\rightarrow$ Boston; USA.CA.* $\rightarrow$ LA). We use '*' to denote a wildcard that can match any number of characters. The flip side of this reduction is that a destination city like USA.CA.Fresno will now match both the USA.* and USA.CA.* prefixes; we must return the longest match (USA.CA.*).

The current version of the Internet (v4) uses 32 bit destination addresses; each Internet router can have a potentially different set of prefixes, each of which we will denote by a bit string (e.g., 01*) of up to 32 bits followed by a '*'. Thus if the destination address began with 01000 and we had only two prefix entries (01* $\rightarrow l_1$; 0100* $\rightarrow l_2$), the packet should be switched to link l_2.

Paper Outline: In this paper, we introduce a set of new algorithmic techniques for improving the performance of Internet lookup algorithms. We begin in Section 2 by describing a model for evaluating lookup performance. We review previous work in Section 3 using our performance model. We begin our contributions by describing three generic techniques in Section 4 that can be applied to improve the performance of several lookup algorithms. We describe how to apply our techniques to improve the performance of trie lookup in Section 5, and binary search on prefix lengths [WVTP97] in Section 6. We describe extensions of our ideas for the next generation Internet in Section 7, and conclude in Section 8.

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

Backbone routers today [Bra97] have databases with about 45,000 prefixes (growing every day, several of them with multiple paths). Enterprise routers have smaller databases (up to 1000 prefixes) because of the heavy use of default routes for outside destinations. Address space depletion has lead to a 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 connect control devices on the net, and the use of features like *anycast* [DH95], all make it unlikely that backbone prefix tables will be smaller than in IPv4. Backbone routers typically run the Border Gateway Protocol, and some implementations exhibit considerable instability, route changes can occur up to 100 times a second[Bra97, RL95].

We use four publically available prefix databases for our comparisons. These are made available by the IPMA project[Inc] 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 representative of a large backbone router; the smallest database, PAIX (around 713 prefixes) can be considered representative of 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 provide better performance and are cheaper after volume manufacturing. For example, BBN [II97] uses DEC Alpha CPUs in each line card, while Torrent and Rapid City [II97] use hardware forwarding engines.

Software: We will consider software platforms using modern processors such as the Pentium[Int] and the Alpha[Cor]. These CPUs execute simple instructions very fast (few clock cycles) but take much longer to make a random access to main memory. The only exception is if the data is in either the *primary* (L1) or it secondary caches which allow access times of a few clock cycles. The distinction arises because main memory uses slow cheap Dynamic Memory (*DRAM,* 60-100 nsec access time) while cache memory uses expensive but fast Static Memory (*SRAM* 10-20 nsec). When a *READ* command is issued to fetch a single word of 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. 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. Finally, both cache accesses and clock cycles for instructions are important for a more refined comparison. To measure these, we must fix an implementation platform and have a performance tool capable of doing dynamic instruction counts that incorporate pipeline and superscalar effects.

We choose 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 RISC platforms such as the Alpha.

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 *READ*s 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

We compare previous schemes for IP lookup using our performance model. We divide these schemes into four categories: conventional algorithms, hardware and caching solutions, protocol based solutions, and recent algorithms. For the rest of this paper, we use **BMP** as a shorthand for Best Matching Prefix and **W** for the length of an address (32 for v4, and 128 for v6).

Conventional algorithms based on Patricia tries or binary search take too many memory accesses for a lookup in the worst case. Hardware solutions[EIE$^+$96] run the risk of being made obsolete in a few years by software running on faster processors and memory. CAM based solutions exist [MTW95],but CAM designs have not historically kept pace with improvements in RAM. Caching has not worked well in the past in backbone routers because of the need to cache full addresses (it is not clear how to cache prefixes). Some studies have shown cache hit ratios of around 50-70 percent [NMH97]. Caching can help but does not avoid the need for fast lookups. Protocol Based solutions like IP and Tag Switching[PST95, CV96, RDK$^+$97] increase the vulnerability of an already fragile set of Internet protocols (see [RL95]) by adding a new protocol that interacts with

every other IP protocol. Also neither completely avoids the BMP problem.

New algorithms: Several new techniques [DBCP97, WVTP97] for best matching prefix were discovered last year. The Lulea Scheme[DBCP97] is based on implementing multi-bit tries but compresses trie nodes to reduce storage to fit in cache. While the worst case is still $O(W)$ memory accesses where W is the address length, these accesses are to fast cache memory. Another form of compressed tries, *level compressed tries* is described in [NK98]. The Wash U scheme[WVTP97] based on binary search of the possible prefix lengths takes a worst case of $\log_2 W$ hashes, where each hash is an access to slower main memory (*DRAM*). It is much harder to determine the new schemes currently being designed or used by router vendors because they regard their schemes as trade secrets. However, Rapid City[II97] and Torrent[Tor] use schemes based on hashing that claim good average performance but have poor worst times (16 memory accesses for the Torrent ASIC scheme).

Database	Number of Prefixes	Number of 24 bit prefixes
MaeEast	38816	22872
MaeWest	14065	7850
Pac	3811	2455
Paix	713	377

Table 1: Prefix databases as on 12 Sep 97.

Performance Comparison: If we rule out pure caching and protocol based solutions, it is important to compare the other schemes using a common implementation platform and a common set of databases. We extracted the BSD lookup code that uses Patricia tries into our Wintel platform; we also implemented binary search on prefixes [LSV98, Per92] and the binary search on prefix lengths [WVTP97] scheme and evaluated them using the largest (Mae East) database. We project the worst case evaluation presented [1] in [DBCP97] to the 300 MHz Pentium II platform with 15 nsec L2 cache. The results are shown in Table 2. Throughout the paper, for performance measurements we use the databases in Table 1, which were obtained from [Inc].

	Average (24 bit prefix (nsec)	Worst case (nsec)	Memory required for Mae East database(KBytes)
Patricia trie	1500	2500	3262
Binary search on prefixes	1200	1500	1280
Binary search on prefix lengths	250	650	1600
Lulea scheme	349	409	160

Table 2: Lookup times for various schemes on a 300 MHz pentium II. The times for binary search on hash tables are projected assuming good hash functions can be found. The average performance is determined by the time taken when the best matching prefix is a 24 bit prefix as there are very few prefixes with length 25 and above.

[1] [DBCP97] presents performance measurements done on a 200 MHz Petium platform

Summary: The measurements in Table 2 indicate that the two conventional schemes take around 1.5 usec; the Lulea scheme takes around 400 nsec in the worst case and the binary search on prefix lengths takes around 650 nsec. Thus using a software model and a reasonably priced processor, these algorithms allow us to do 2 million lookups per second while the older algorithms allow roughly 1/2 million lookups per second. Ordinary binary search will perform somewhat better in an enterprise router with a smaller number of prefixes; the worst case times of other schemes are not sensitive to the number of prefixes used. Given that the average packet size is around 2000 bits [Bra97], even the old schemes (that take up to 2 usec and forward at 1/2 million packets per second) allow us to keep up with line rates for a gigabit link.

Despite this, we claim that it is worth looking for faster schemes. First, the forwarding rate is sensitive to the assumption of the average packet size. If the average packet size goes down to 64 bytes, we will need a factor of 3 speed up. Second, the Internet will surely require Terabit routers in a few years which will require a factor of improvement in lookup times to around 100 nsec. Finally, it is worth looking for other improvements besides raw lookup speed: for instance, schemes with faster update times, and schemes with smaller storage requirements. While these numbers are not available, it seems clear that the faster schemes in [DBCP97, WVTP97] will have slow insert/delete times because they do not allow incremental solutions: the addition or deletion of a single prefix potentially requires complete database reconstruction.

3.1 Our Contribution:

By applying our generic techniques, we propose a trie based lookup algorithm with stringent worst case time bounds for *lookup, prefix insert, prefix delete and prefix route change*.

- We get a worst case lookup time of 200 nsec, which is twice as fast as other known algorithms on the same platform.
- IPMA [Inc] reports that number of routing prefixes announced and withdrawn by BGP in a single day was 1899851 (on Nov 1, 97), which is about 25 prefix updates/sec. Some implementations are more unstable [Bra97] requiring 100 prefix updates/sec. The trie scheme can be configured to *add* or *delete* a prefix in a worst case time of 2.5 millisec, and so can handle upto 400 prefix updates/sec in the worst case. While the worst case for add/delete is 2.5 millisec for a node of large stride like 16, the expected case is of the order of tens of microseconds.
- From the same report [Inc], we see that while there are 1899851 prefix updates, there are only 47138 unique prefixes! That is, while there are prefixes being added and deleted, many of the routing updates are *route changes*. The report also shows that many prefixes are reachable all the time through some route. Using the trie scheme, changing the route associated with a prefix takes a worst case of 3 usecs as opposed to the 2.5 millisecs for adding or deleting a prefix.

4 New Techniques

We describe the three main techniques on which our specific lookup algorithms are based. We start with a simple technique called *controlled expansion* that converts a set

of prefixes with M distinct lengths to a set of prefixes with $N < M$ distinct lengths. This is useful because the lookup time of many existing lookup algorithms (e.g., tries, binary search) increases as the number of distinct prefix lengths increases. Naive expansion can considerably increase storage. Thus, our second technique uses *dynamic programming* to solve the problem of picking optimal expansion levels to reduce storage. Finally, our third technique, *local restructuring*, is a collection of techniques to restructure data structures to reduce storage and to increase cache locality. We will apply these three techniques to tries and binary search on levels in the following sections.

4.1 Controlled Prefix Expansion

It is not hard to show that if the number of distinct prefix lengths, is L and $L < W$, then we can refine the worst case bounds for tries and binary search on levels to $O(L)$ and $O(\log_2 L)$ respectively. The fact that restricted prefix lengths leads to faster search is well known. For example, OSI Routing uses prefixes just as in IP, but the prefix lengths are multiples of 4. Thus it was well known that trie search of OSI addresses could proceed in strides of length 4 [Per92]

Thus if we could restrict IP prefix lengths we could get faster search. But the backbone routers we examined [Inc] had all prefix lengths from 8 to 32! Thus prefix lengths are almost arbitrary. Our first idea is *to reduce a set of arbitrary length prefixes to a predefined set of lengths using a technique that we call controlled prefix expansion.*

Original	Expanded (3 levels)	
P5 = 0*	00*(P5)	Length 2
P1 = 10 *	01*(P5)	
P2 = 111 *	10*(P1)	
P3 = 11001*	11*(P4)	
P4 = 1*	11100*(P2)	Length 5
P6 = 1000 *	11101*(P2)	
P7 = 100000*	11110*(P2)	
P8 = 1000000*	11111*(P2)	
	11001*(P3)	
	10000*(P6)	
	10001*(P6)	
	1000001*(P7)	Length 7
	1000000*(P8)	

Figure 1: Controlled Expansion of the original database shown on the Left (which has 7 prefix lengths from $1 \dots 7$) to an Expanded Database (which has only 3 prefix lengths $2, 5$ and 7). Notice that the Expanded Database has more prefixes but less distinct lengths.

Suppose we have the IP database shown in the left of Figure 1 that we will use as a running example. Notice that this database has prefixes that range from length 1 (e.g., P4) all the way to length 7 (e.g., P8). Thus we have 7 distinct lengths. Suppose we want to reduce the database to an equivalent database with prefixes of lengths 2, 5, and 7 (3 distinct lengths).

Clearly a prefix like P4 = 1* that is of length 1 cannot remain unchanged because the closest admissible length is 2. The solution is to expand P1 into *two prefixes of length 2* that are equivalent. This is easy if we see that 1* represents all addresses that start with 1. Clearly of these addresses, some will start with 10 and the rest will start with 11. Thus, the prefix 1* (of length 1) is equivalent to the union of the two prefixes 10* and 11* (both of length 2). In particular, both the expanded prefixes will inherit the output link of the original prefix (i.e., P4) that was expanded. In the same way, we can easily expand any prefix of any length m into multiple prefixes of length $n > m$. For example, we can expand P2 (111*) into four prefixes of length 5 (11100*, 11101*, 11110*, 11111*).

We also need an accompanying concept called *prefix capture*. In Figure 1, we have two prefixes P1 = 10* and P4 = 1*. When we expand P4 into the two prefixes 10* and 11*, we find we already have the prefix P1 = 10*. Since we do not want multiple copies of the same prefix, we must pick one of them. But when P1 and P4 overlap, P1 is the longer matching prefix. In general, when a lower length prefix is expanded in length and one of its expansions "collides" with an existing prefix, then we say that the existing prefix *captures* the expansion prefix. When that happens, we simply get rid of the expansion prefix. For example, we would get rid of the expansion 10* corresponding to 1*, because it is captured by the existing prefix P1 =10*.

Thus our first technique, *controlled prefix expansion* combines prefix expansion and prefix capture to reduce any set of arbitrary length prefixes into an expanded set of prefixes of any prespecified sequence of lengths $L_1, L_2, \dots, L_k$. The complete expanded database is shown on the right of Figure 1 together with the original prefix that each expanded prefix descends from. For example, 11101* descends from P2 = 111*.

4.2 Picking Optimal Expansion Levels

Expansion by itself is a simple idea. Some papers [WVTP97, GLM98] have proposed using an initial X bit ($X = 16$ or 24) array lookup as a front-end lookup before using other schemes to determine a longer matching prefix. This amounts to a limited form of expansion where all prefixes of length less than X are expanded to prefixes of length X. What distinguishes our idea is its generality (expansion to any target set of levels), its orthogonality (expansion can used to improve most lookup schemes), and the accompanying notion of optimality (picking optimal expansion levels).

We briefly describe the optimality problem. In controlled expansion, we did not specify how we pick target prefix lengths (i.e., $L_1 \dots L_k$). Clearly, we wish to make k as small as possible. For example, if we were doing a trie lookup and we wanted to guarantee a worst case trie path of 4 nodes, we would choose $k = 4$. But which 4 target lengths do we choose? We chose the target lengths to minimize the memory used using dynamic programming [CLR90].

4.3 Local Restructuring

Local restructuring refers to a collection of heuristics that can be used to reduce storage and improve data structure locality. We describe two heuristics: leaf pushing and cache alignment.

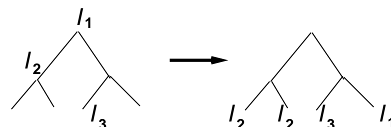

Figure 2: Leaf pushing pushes the final information associated with every path from a root to a leaf to the corresponding leaf. This saves storage because each non-leaf node need only have a pointer, while leaf nodes (that used to have nil pointers) can carry information.

Leaf Pushing: Consider a tree of nodes (see left side of Figure 2) each of which carry information and a pointer. Suppose that we navigate the data structure from the root to some leaf node and the final answer is some function of the information in the nodes on the path. Thus on the left side of Figure 2, if we follow the path to the leftmost leaf we encounter I_1 and I_2. The final answer is some function of I_1 and I_2. For concreteness, suppose it is the last information seen — i.e., I_2.

In *leaf pushing*, we precompute the answer associated with each leaf. Thus on the right side of Figure 2 we have pushed all answers to the leaves, assuming that the answer is the last information seen in the path. Originally, we had a pointer plus an information field at each node except at the leaves which have information but only a null pointer. After leaf pushing, every node has either information or a non-null pointer, but not both. Thus we have reduced storage and improved locality. However, there is cost for incremental rebuilding: if information changes at a node close to the root can potentially change a large number of leaves.

Cache Line Alignment: Recall that a *READ* to an address A in the Pentium will prefetch an entire cache line (32 bytes) into cache. We use two simple applications of this idea.

First, if we have a large sparse array with a small amount of actual data that can fit into a cache line, we can replace the array with the actual data placed contiguously. While this saves storage, it can potentially increase time because if we have to access an array element X we have to search through the compressed list of data using say binary search. This can still improve time because the entire compressed array now fits into a cache line and the search (linear or binary) is done with processor registers[DBCP97]. We refer to such compressed arrays as *packed arrays.*

A second application is for perfect hashing [CLR90]. Perfect hashing refers to the selection of a collision free hash function for a given set of hash entries. Good perfect hash functions are hard to find. In Section 6, where we use hashing, we settle for what we call *semi-perfect hashing.* The idea is to settle for hash functions that have no more than 6 collisions per entry. Since 6 entries can fit into a cache line on the Pentium, we can ensure that each hash takes 1 true memory access in the worst case although there may be up to 6 collisions. The collision resolution process will take up to 5 more cache accesses.

5 Tries with Expansion

5.1 Expanded Prefix tries

A one-bit trie[Knu73] is a tree in which each node has a 0-pointer and a 1-pointer. If the path of bits followed from the root of the tree to a node X is P, then the subtree rooted at X stores all prefixes that begin with P. Further, the 0-pointer at node X points to a subtree containing (if any exist) all prefixes that begin with the string $P0$; similarly, the 1-pointer at node X points to a subtree containing (if any exist) all prefixes that begin with the string $P1$.

Using our crude performance model, 1-bit tries perform badly because they can require as many as 32 READs to memory. We would like to navigate the trie in strides that are larger than 1 bit. The central difficulty is that if we navigate in say strides of 8 bits, we must ensure that we do not miss prefixes that have lengths that are not multiples of 8. The solution is to use the prefix expansion idea (Section 4) and build a trie with strides larger than 1 bit on the expanded set of prefixes.

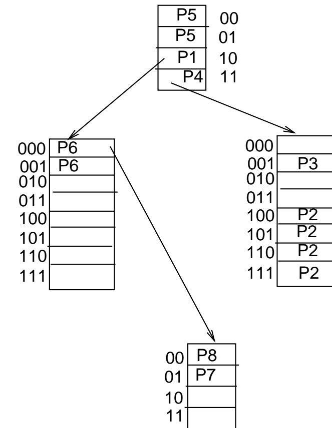

Figure 3: Expanded trie corresponding to the database of Figure 1. Note that the expanded trie only has a maximum path length of 3 compared to the 1 bit trie that has a maximum path length of 7.

Figure 3 shows how the expanded database of Figure 1 is placed in a multibit trie. Since we have expanded to lengths 2, 5, and 7 the first level of the trie uses 2 bits, the second uses 3 bits (5-2), and the third uses 2 bits (7-5). If a trie level uses m bits, then each trie node at that level is an array of 2^m locations.

Search: To search, we break the destination address into chunks corresponding to the strides at each level of the trie (e.g., 2, 3, 2 in the above example) and use these chunks to follow a path through the trie until we reach a nil pointer. As we follow the path, we keep track of the last prefix that was alongside a pointer we followed. This last prefix encountered is the best matching prefix when we terminate.

We assume that each trie node N is a two-dimensional array where $N[bmp, i]$ contains any stored prefix associated with entry i and $N[ptr, i]$ contains any pointer associated with entry i. Absence of a valid prefix or pointer is signaled by the special character "nil". It is easy to see that the search time is $O(k)$, where k is the maximum path through the expanded trie, which is also the maximum number of expanded lengths. We also have at most two memory references per trie node; however, we can use cache line alignment (as these are adjacent locations) to reduce this to 1 memory reference per trie node.

Insertion and Deletion: Simple insertion and deletion algorithms exist for multibit tries.

Insertion involves following the trie path for the prefix to be inserted and expanding it in the trie node it ends up in; or in the addition of new trie nodes if the prefix is longer than the existing trie path. Deletion is similar to Insert. The complexity of insertion and deletion is the time to perform a search ($O(W)$) plus the time to completely reconstruct a trie node ($O(S)$ where S is the maximum size of a trie node.) For example, if we use 8 bit trie nodes, the latter cost will require scanning roughly $2^8 = 256$ entries. On the other hand, if we use 17 bit nodes, this will require potentially scanning 2^{17} entries. On a Pentium with a *Write Back* cache, while the first word in a cache line would cost 100 nsecs for the read, the writes would be only to cache and the entire cache line will take at most 200

nsecs to update. A single cache line holds 10 entries, so at 20 nsec/entry for 2^{17} entries we get 2.5 millisec.

The average times will be much better, as most of the prefixes are 24 bit prefixes. Since addition or deletion involves a search through the auxiliary structure (searching through the auxiliary structure has a worst case of O(W) memory accesses) and affects exactly one array element; this can take upto 3 usecs. *In our dynamic programs later in subsections 5.2 and 5.3, we constrain the nodes to be of maximum stride* 17*, so that the worst case criterion for insertion/delete times is met.*

Route Change: The search for the longest match in the datastructure gives an index into an 'intermediate' table. This table has one entry per prefix, and contains the output link corresponding to that prefix. This level of indirection is to support fast route changes. Route change involves searching for the prefix and then updating the corresponding entry in the intermediate table. This takes at most 3 usecs.

We present the time taken for insertion and for leaf pushing for the various databases for a 4 level trie in Table 3. Note that the average insertion time is $\frac{170 millisec}{38816} = 4$ usec. A 1-bit at a time trie is used as an auxiliary structure for supporting deletes is used to store the prefixes in each node. The memory required for this is given in Table 4.

Database	Number of nodes	Memory (KBytes)	Build Time (millisec)
MaeEast	85252	1998	170
MaeWest	41642	975	52
Pac	14172	332	19
Paix	4788	112	5

Table 4: Memory requirement and time to build 1-bit trie without path compression: auxiliary structure to support deletion.

5.2 Computing Optimal Fixed Strides

What are the degrees of freedom that we have? The obvious degree of freedom is to pick the expansion lengths $L_1 \ldots L_k$ which determine the strides through the trie. This assumes that all trie nodes at Level j have the same stride size. However, there is a further degree of freedom. We could make the stride size *variable* for each trie node. The root node must, of course, have a fixed starting stride size but each child of the root node could have a different stride size that is encoded along with the pointer that leads to child. We now describe how to pick optimum stride lengths.

In this subsection, we restrict ourselves to fixed stride tries. They may be preferable because of their simplicity and the slightly faster search time. Assume a prefix database with L different prefix lengths. We wish to find the optimal set of k expansion levels that yields the smallest memory requirement for building a trie. We use the following dynamic program.

Define the *Cost* of a given set of k levels as *the number of computer words* needed by the trie that is built after expanding the database to these levels.

We build an auxiliary 1-bit trie. After building the 1-bit trie, the problem of finding the optimal levels for the trie can be found using the number of nodes in each level of the 1-bit trie. From now on we will distinguish the input levels (*trie levels*) from the output levels (*expansion levels*). Note that if all the 1 bit nodes at trie level $i+1$ are replaced by $j > i$ bit nodes, then all the 1 bit nodes in trie levels $i+1$ to $i+j$ can be ignored.

Thus if i is chosen as a level, placing the next expansion level at $i+j$ has a cost $nodes(i+1) * 2^j$, where $nodes(i)$ is the number of nodes at trie level i. Notice that we use $i+1$ instead of i. This is because all nodes at trie level i will be assigned to the expansion level; only nodes at Level $i+1$ need to be expanded. Finally, nodes at trie levels $i+2$ to j get a "free ride" from the expansion of trie level $i+1$.

With this prelude, consider the general problem of covering trie levels from 0 to say j using expansion levels 1 to r. Expansion level r is constrained to be at trie level j. We can reduce the problem to that of first placing expansion level $r-1$ at some trie level m in the range $[r-1, j-1]$. We are then left with the recursive problem of finding the optimal way to cover trie lengths from 1 to m using expansion levels 1 to $r-1$. This is illustrated in Figure 4.

Figure 4: Covering trie lengths from 0 to j using expansion levels 1 to r can be solved recursively by first placing Level r at length j, choosing Level r to be at some optimal length m. Finally, we recursively solving the covering problem for the upper portion of the tree using $r-1$ levels.

Using Figure 4 as a guide, we get the following equations that define a solution:

$$T[j, r] = min_{m \in \{r-1..j-1\}} T[m, r-1] + nodes(m+1) * 2^{j-m}$$

$$T[j, 1] = 2^{j+1}$$

The initial problem is $T[W-1, k]$ because when the strings are of maximum length W, the trie levels range from 0 to $W-1$. Recall that k is the target number of memory accesses we wish the trie to have after expansion. We can turn the equations above into a simple dynamic programming solution. Since we have two variables, one of which ranges from 0 to $W-1$, and the second of which ranges from 0 to k, we have $W * k$ subproblems. Because of the minimization, each subproblem takes $O(W)$ to compute; thus the complexity of our optimal fixed stride determination algorithm is $O(k * W^2)$. The results of using this optimization on real databases are reported in Section 5.4.

5.3 Optimal Trie using varying strides

We now consider designing storage-optimal tries that use varying strides. Assume as before that we have built an auxiliary 1-bit trie, and we wish to convert this into a variable stride trie that takes at most say k memory accesses.

We are given the maximum number of memory accesses k. Let R be the root of the 1-bit trie T built on the original prefix entries. We can walk through the trie in k accesses if we have a first stride s (for the new root node) and can walk through all the subtries rooted at trie level $s+1$ in $k-1$ accesses. This is illustrated in Figure 5. But each such subtrie can be recursively covered using the same algorithm. We omit further details of this algorithm except to say that it is fairly intricate, and requires depth first numbering of trie nodes to determine descendants of a node, and other algorithmic machinery.

Database	Number of nodes	Memory (Non-leaf-pushed)	Time (millisec) Build	Leaf Pushing	Memory After Leaf Push	Time for random search
MaeEast	2671	4006	170	55	2003	110 nsec
MaeWest	1832	2748	52	40	1374	100 nsec
Pac	803	1204	19	20	602	100 nsec
Paix	391	586	5	7	293	100 nsec

Table 3: Four Level with a stride length of 8 bits. The random search was done assuming all IP addresses are equally likely.

Levels	2	3	4	5	6	7	8	9	10	11	12
MaeEast	49168	1848	738	581	470	450	435	430	429	429	429
MaeWest	30324	1313	486	329	249	231	222	219	218	217	217
Pac	15378	732	237	132	99	87	79	76	75	75	75
Paix	7620	448	116	54	40	33	29	27	26	26	26

Table 5: Memory requirement(in KBytes) for different number of levels in the trie for the prefix database 1 using the fixed stride Dynamic Program (time taken for the dynamic program is 1 millisec for any database).

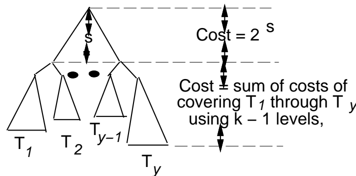

Figure 5: Covering a variable stride trie using k expansion levels. We first pick a stride s for the new root node and then recursively solve the covering problem for the subtrees rooted at Trie Level $s+1$ using $k-1$ levels.

5.4 Measured Performance of Expanded Tries

From Tables 5,6,7,8 we use appropriate values to construct Table 10. For example, the memory requirement for the MaeEast database for Non-leaf Pushed Variable stride trie for 3 levels is found from Table 6 to be $575 * 2 = 1150 KBytes$. To this we add the intermediate table size of 80 KBytes to get 1230 KBytes, which is the corresponding entry in Table 10.

Measurements were done using VTune, which does both static and dynamic analysis of the code for Pentium Processors. It takes into account pairing of instructions in the parallel pipeline, and stall cycles introduced due to dependencies between registers used in consecutive instructions. Using VTune, we find that for one level of the leaf pushed trie, the time taken is *1 Memory Access Time + 6 clock cycles*. The instructions at the beginning and end of the code take another 4 cycles. So for a k level trie, the worst case is $(10 + 6(k-1)) * clk + k * M_D$, which is $(6k+4) * clk + k * M_D$, where clk is the clock cycle time, which is $3.33 nsec$ for the 300 MHz Pentium and M_D is the memory access delay. Similarly we find time requirements for the various trie schemes.

M_D depends on the data required being in the L1 cache, L2 cache or in the main memory. With 512 KBytes of L2 cache available in the Pentium Pro, we can fit the entire 4 level trie and the intermediate table into the L2 cache. Access delay from the L2 cache is 15 nsecs. The delay for an access to the main memory is 60 nsecs. When there is a L2 miss, the total penalty is $60 + 15 = 75$ nsecs.

We now present a few tables based on the above timings, taking into account the size of the data structure. If the data structure is small enough to be fit in the L2 cache of 512 KBytes, then the memory access delay is counted as 15 nsecs. Otherwise it is counted as 75 nsecs. All numbers assume that we always miss the L1 cache. For the trie, the memory requirement is calculated by adding the size of the trie that is built by using the dynamic program and the size of the intermediate table (which is 2 bytes per prefix, and so 80 KBytes for the MaeEast database). When the total size is a little more than 512 KBytes, we assume that only the first level of the trie is in L2, cache while the others lead to a L2 miss.

5.5 Choosing a trie

From Table 10, we can decide which type of trie to use for the MaeEast database. We see that the minimum search time of 196 nsec is attained for the Leaf Pushed Variable Stride trie with 4 levels. Now we have to see if this meets the requirements for insertion and deletion. For a leaf pushed trie, a single insert or delete can cause nearly all the array elements in the entire trie to be scanned. For this 500 KByte structure, this could take upto 10 millisec. We find that the Non-leaf Pushed variable stride trie takes 226 nsec worst case for search, and can support prefix insert and delete operations with a worst case of 2.5 millisec (the maximum stride is constrained to 17 as mentioned earlier to guarantee this). Also note that in this case, the worst case of 226 nsec is calculated assuming that the entire structure is only in main memory (without any assumptions about part of structure being in L2 cache). Similar tables can be constructed for other databases and a choice made based on the requirements and the machine that is being used.

6 Faster Binary Search on Prefix Lengths

Binary search on levels presented in [WVTP97] involves organizing the prefixes in hash tables for each distinct length and doing a binary search among the lengths to locate the BMP. It also involves placing markers to enable binary search.

6.1 Using Expansion

The complexity of binary search on prefix lengths is actually $\log_2 L$, where L is the number of distinct prefix lengths. For the real backbone databases we examined, L

	2 Levels		3 Levels		4 Levels		5 Levels		6 Levels		7 Levels	
	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
MaeEast	1559	130	575	871	423	1565	387	2363	377	2982	375	3811
MaeWest	1067	61	346	384	231	699	205	1009	196	1310	194	1618
Pac	612	16	163	124	93	125	77	311	71	384	69	494
Paix	362	2	83	10	40	29	30	60	27	70	26	91

Table 6: Memory requirement(in KBytes) using the variable stride Dynamic Program and the time taken for the dynamic program in milliseconds. Non-leaf pushed trie will have twice the memory requirement as the entries in this table.

	2 levels	3 levels	4 levels	5 levels	6 levels	7 levels
MaeEast	983	460	370	347	340	338
MaeWest	539	230	179	165	160	158
Pac	261	81	57	51	48	47
Paix	89	22	14	12	11	11

Table 7: Memory requirement(in KBytes) using the variable stride Dynamic Program, leaf pushed and allowing packed array nodes.

is 23. Thus the worst-case time of binary search for IPv4 is indeed 5 hashes. However, this immediately suggests the use of expansion to reduce the number of distinct lengths L. While the gain is only logarithmic in the reduced lengths, if we can reduce the worst case from 5 to 2 (or 3) we can double performance.

6.2 Applying Dynamic Programming

A prefix can cause markers to be placed in hash tables which contain prefixes of shorter lengths. Expansion causes the number of such hash tables to reduce, so there are fewer hash tables now in which markers need to be placed. So expansion can increase the number of prefixes but reduce the number of markers. These two effects can compensate up to a point, even in a worst case sense. When N prefixes are expanded so that only $W/2$ distinct lengths are left in the database the number of prefixes doubles to $2N$. The worst case number of markers changes from $N * (\log_2 W - 1)$ to $N * (\log_2 W/2 - 1) = N * (\log_2 W - 2)$. Thus the total storage remains unchanged at $N * \log_2 W$! Thus the worst case number of hashes can be reduced by 1 without changing the worst case storage requirement. We can pick k distinct lengths to expand to and then apply binary search on these levels to get a worst case of $\log_2 k$ hashes. We apply dynamic programming to chose the k levels that minimize the memory requirement.

6.3 Performance of Expansion + Binary Search on Prefix Lengths

We present the results obtained by expanding the prefixes to have only 3 distinct levels. We search for a hash function that gives at most 6 collisions; the time to find one such function is given in Table 11. The first 16-17 bit table is implemented as a full array, while the others are hash tables. The time taken when random IP addresses were searched, the time taken by the dynamic program, and the memory requirement when using levels given by the dynamic program, are all presented in Table 11.

7 IPv6 and Faster Lookups

For IPv6, an interesting combination is to do a hash based scheme on Levels 32, 64, and 96 and then do trie search on the lengths we are left with (0-32, 32-64, 64-96, 96-128). The hash tables should mostly contain markers and so should be fairly small and be able to fit in cache (*SRAM* in hardware).

A hardware estimate of this scheme assuming an initial hash plus a 4 level trie stored in *SRAM* would give a worst-case speed of around 120 nsec. This assumes 10 nsec *SRAM* and a 10 nsec clock, two memory accesses for the hash, and 4 memory accesses for the trie. If further speed increase is needed in hardware, we can pipeline. Any search tree, whether a trie or a binary search tree, can be pipelined by splitting its levels into pipeline stages. The basic idea is that each level of the tree corresponds to a pipeline stage. Pipelining allows 1 lookup per memory access at the cost of increased complexity.

8 Conclusions

We have described a new set of techniques based on controlled expansion and optimization that can be used to improve the performance of any scheme whose search times depend on the number of distinct prefix lengths in the database. Our trie schemes provide fast lookup times and have fast worst case Insert/Delete times. When compared to the Lulea scheme[DBCP97] we have a version (leaf-pushed, variable stride) that has faster lookup times (196 nsec versus 409 nsec) but somewhat more memory (500 kBytes versus 160 kBytes). More significantly, we have a trie variant that has fast lookup times (226 nsec) and reasonably fast worst-case insert times (2.5 msec). There are no reported insertion times for the Lulea scheme, because it inserts are supposed to be rare [DBCP97, WVTP97]. However, because BGP updates can add and withdraw prefixes at a rapid rate, prefix insertion *is* important.

In general, the Lulea scheme can potentially be cheaper in hardware because it uses a smaller amount of memory. However, it pays a penalty of four memory accesses per trie node for node compression in place of one access. Thus, using (expensive!) 10 nsec SRAM, we could easily build a trie lookup scheme that takes 60 nsec while the Lulea scheme would require 240 nsec. Our trie scheme would require more memory and hence cost more, but the cost will potentially be dwarfed by the cost of expensive optics needed for Gigabit and Terabit links.

With expansion, binary search on levels [WVTP97] can be made fairly competitive. Its main disadvantage is the

	2 levels	3 levels	4 levels	5 levels	6 levels	7 levels
Mae East	1744	728	580	543	534	531
Mae West	925	345	257	234	227	225
Pac	447	123	79	69	65	64
Paix	157	32	18	15	14	14

Table 8: Memory requirement(in KBytes) using the variable stride Dynamic Program, Non-leaf pushed and allowing packed array nodes.

Type of Trie	Time for search
Leaf Pushed Fixed Stride	$(6k+4)*clk+k*M_D$
Leaf Pushed Variable Stride	$(8k+2)*clk+k*M_D$
Non-leaf Pushed Fixed Stride	$((10k+1)*clk+k*M_D$
Non-leaf Pushed Variable Stride	$(12k-1)*clk+k*M_D$
Leaf Pushed Variable Stride with packed array nodes	$(8k+12)*clk+k*M_D$
Non-leaf Pushed Variable Stride with packed array nodes	$(12k+5)*clk+k*M_D$

Table 9: Time requirements for search when using a k level trie, found using Vtune for pentium. clk is the clock cycle time and M_D is the memory access delay.

time and memory required to find perfect hash functions, and its slow insert times; however, its average performance for IPv4 databases is competitive. Its real advantage is the potential scalability it offers for IPv6, either directly or using the combination scheme we have described.

While our approach is based on new algorithms we emphasize that it is also architecture and measurement driven. We rejected a number of approaches (such as compressing one-way trie branches) because the measurements indicated only small improvements.

We believe (with [WVTP97] and [DBCP97]) that IP lookup technology can be implemented in software. We also believe that fast lookup algorithms make the arguments for Tag and IP switching less compelling. For instance, neither Tag or IP switching appears to help implement fast firewalls. We have recently found a way to use lookup technology to do fast firewalls. If fast lookups are needed anyway for firewalls, it is not clear that other mechanisms are needed for forwarding in routers. Finally, we believe that routers of the future may be less vertically integrated than at present; instead they will be assembled from special chips for functions (e.g., lookups, switching, and scheduling) and commodity routing software, just as computers evolved from mainframes to PCs. We hope the lookup technology described by us and others will contribute to this vision of the future.

References

[Bra97] Scott Bradner. *Next Generation routers Overview.* Proceedings of Networld Interop 97, 1997.

[CLR90] T.H. Cormen, C.E. Liecerson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cor] Digital Electronics Corporation. *The DEC Alpha.* http://www.dec.com.

[CV96] Girish P. Chandranmenon and George Varghese. Trading Packet Headers for Packet Processing. *IEEE/ACM Transactions on Networking,* April 1996.

[DBCP97] M DegerMark, A Brodnik, S Carlsson, and S Pink. Small Forwarding Tables for Fast Routing Lookups. *Computer Communication Review,* October 1997.

[DH95] S Deering and R Hinden. *Internet Protocol, Version 6 (IPv6) Specification RFC 1883.* http://ds.internic.net/rfc/rfc1883.txt, 1995.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a fair queuing algorithm. *Proc. of Sigcomm'89,* September 1989.

[EIE+96] Andrew Walton Reading England, Una M. Quinlan Dublin Ireland, Stewart F. Bryant Redhill England, Michael J. Seaman San Jose Calif, John Rigby Reading England, Fearghal Morgan Moycullen, and Joseph O'Callaghan Glounthaune both of Ireland. Address recognition engine with look-up database for storing network information. U.S. Patent 5519858. Assignee Digital Equipment Corporation, Maynard, Mass., 1996.

[GLM98] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. *Proceedings of the IEEE Infocom 98,* April 1998.

[Gra96] Matthew Gray. *Internet Growth Summary.* http://www.mit.edu/people/ mkgray/net/internet-growth-summary.html, 1996.

[II97] Internet II. *Big Fast Routers: multi-megapacket forwarding engines for Internet II.* Proceedings of Networld Interop 97, 1997.

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

[Int] Intel. *The Pentium Processor.* http://www.pentium.com.

[Knu73] Donald Knuth. *Fundamental Algorithms vol 3: Sorting and Searching.* Addison-Wesley, 1973.

[LSV98] Butler Lampson, V. Srinivasan, and George Varghese. IP Lookups using Multi-way and Multicolumn Search. *Proceedings of the IEEE Infocom 98,* April 1998.

[MTW95] 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. Assignee Bell Communications research Inc Livingston NJ, January 1995.

	2 levels		3 levels		4 levels		5 levels		6 levels	
	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
Leaf Pushed fixed stride	49168	203	1930	298	820	393	660	428	550	523
Leaf Pushed Variable stride	1640	219	655	268	500	196	460	244	450	293
Non-leaf Pushed fixed	98275	209	3780	311	1560	413	1240	514	1020	616
Non-leaf Pushed variable	3200	226	1230	341	920	456	840	571	820	686
Leaf Pushed variable stride with packed array nodes	1063	243	540	284	450	206	427	248	420	289
Non-leaf Pushed variable with packed array nodes	1824	246	808	361	640	476	623	591	614	706

Table 10: Worst case lookup time(nsec) taken for various types of trie. With 512 KByte L2 cache, when the memory required is less than 512 KBytes, it can be fit into the L2 cache. The memory requirement in this table is for the MaeEast database, including the intermediate table. Notice how the search time drops when the entire structure fits into L2, even though the number of trie levels have increased.

Database	Time to find hash (sec)	Memory (KBytes) (16,24,32)	Time for dynamic program(ms)	Memory (KBytes) using levels from dynprog	Random Search (nsec)
MaeEast	750	4250	650	3250	190
MaeWest	150	2250	287	1250	190
Pac	8	512	55	512	180
Paix	0.1	256	15	256	180

Table 11: Binary search on hash tables after expanding to levels given by the dynamic program and time taken for the dynamic program. Random search is done assuming all IP addresses are equally likely.

[NK98] S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. *Proceedings of IEEE Broadband Communications 98*, April 1998.

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

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

[PST95] Guru Parulkar, Douglas C. Schmidt, and Jonathan Turner. IP/ATM: A Strategy for Integrating IP with ATM. *Computer Communication Review*, October 1995.

[RDK+97] Y. Rechter, B. Davie, D. Katz, E. Rosen, and G. Swallow. Cisco Systems' Tag Switching Architecture Overview. Technical Report RFC 2105, February 1997.

[RL95] Y Rechter and T Li. *A Border Gateway Protocol 4 (BGP-4) RFC 1771.* http://ds.internic.net/rfc/rfc1771.txt, 1995.

[SV95] M. Sreedhar and George Varghese. Efficient Fair Queuing using Deficit Round Robin. *Computer Communication Review*, October 1995.

[Tam97] Alan Tammel. *How to survive as an ISP.* Proceedings of Networld Interop 97, 1997.

[Tor] Torrent. *Torrent Systems, Inc.* http://www.torrent.com.

[WVTP97] M Waldvogel, G Varghese, J Turner, and B Plattner. Scalable High Speed IP Routing Lookups. *Computer Communication Review*, October 1997.