Fast Address Lookups Using Controlled Prefix Expansion

V. SRINIVASAN and G. VARGHESE
Washington University in St. Louis

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 and updates can be made faster using a set of transformation techniques. Our main technique, *controlled prefix expansion*, transforms a set of prefixes into an equivalent set with fewer prefix lengths. In addition, we use optimization techniques based on dynamic programming, and local transformations of data structures to improve cache behavior. When applied to trie search, our techniques provide a range of algorithms (*Expanded Tries*) whose performance can be tuned. For example, using a processor with 1MB of L2 cache, search of the MaeEast database containing 38000 prefixes can be done in 3 L2 cache accesses. On a 300MHz Pentium II which takes 4 cycles for accessing the first word of the L2 cacheline, this algorithm has a worst-case search time of 180 nsec., a worst-case insert/delete time of 2.5 msec., and an average insert/delete time of 4 usec. Expanded tries provide faster search *and* faster insert/delete times than earlier lookup algorithms. When applied to Binary Search on Levels, our techniques improve worst-case search times by nearly a factor of 2 (using twice as much storage) for the MaeEast database. Our approach to algorithm design is based on measurements using the VTune tool on a Pentium to obtain dynamic clock cycle counts. Our techniques also apply to similar address lookup problems in other network protocols.

Categories and Subject Descriptors: C.2.3 [**Computer-Communication Networks**]: Network Operations—*network monitoring*

General Terms: Performance

Additional Key Words and Phrases: Binary Search on Levels, controlled prefix expansion, expanded tries, Internet address lookup, longest-prefix match, multibit tries, router performance

George Varghese is supported by NSF Grant NCR-9405444 and an ONR Young Investigator Award. An earlier version of this article appeared in the *Proceedings of the ACM Sigmetrics'98/Performance'98 Joint International Conference on Measurement and Modeling of Computer Systems.*

Authors' address: Computer and Communications Research Center, Department of Computer Science, Washington University in St. Louis, One Brookings Drive, Campus Box 1045, St. Louis, MO 63130; email: cheenu@ccrc.wustl.edu; varghese@ccrc.wustl.edu.

1. INTRODUCTION

From the present intoxication with 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 triples approximately every two years [Gray 1996]. Also, Internet traffic is doubling every three months [Tammel 1997], 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,[1] and yet the fundamental limits of optical transmission have hardly been approached. Thus the key to improved Internet performance is faster routers. This market opportunity has led to a flurry of startups (e.g., Avici, Juniper, Torrent) that are targeting the gigabit and terabit router market.

What are the fundamental impediments to improved router performance? An Internet message (called a packet) arrives at an input link. A processor[2] examines the destination address of the packet in a Forwarding Database to determine the output link. In some cases, the processor may also perform "firewall" transit checks that require a lookup in a database of firewall rules. If the checks succeed, the processor instructs a switching system to switch the packet to the corresponding output link. Many routers also do a form of scheduling at the output link to ensure fairness among competing packet streams, and to provide delay bounds for time-critical traffic such as video.

Thus the three central bottlenecks in router forwarding are lookups, switching, and output scheduling. Switching is well studied, and good solutions like fast busses and crossbar switches (e.g., McKeown et al. [1997] and Turner [1997]) have been developed. Similarly, most vendors feel that full-scale fair queuing[3] is not required for a few years until video usage increases. In the interim, cheaper approximations such as weighted and deficit round robin [Shreedhar and Varghese 1996] ensure fairness and can easily be implemented. Thus a major remaining bottleneck is fast Internet lookups, the subject of this article.

1.1 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

[1]MCI and UUNET have upgraded their Internet backbone links to 622Mbits/sec.
[2]Some designs have a processor per input link; others use a processor per router. Some designs use a general purpose processor; others use dedicated hardware.
[3]Fair queuing [Demers et al. 1989] is a form of output scheduling that guarantees bandwidth fairness and tight delay bounds.

complex lookup called *longest matching prefix*. It also requires a more complex update procedure when prefixes are added and deleted.

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 a thousand entries to two prefix entries (USA* $--$> Boston; USA.CA.* $--$> LA). We use an asterisk 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 Internet address lookup problem is similar except that that we use bit strings instead of character strings. The currently deployed 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* $--$> L1; 0100* $--$> L2), the packet should be switched to link L2.

The Internet began with a simpler form of hierarchy in which 32-bit addresses were divided into a network address and a host number, so that routers could only store entries for networks. For flexible address allocation, the network addresses came in variable sizes: Class A (8 bits), Class B (16 bits), and Class C (24 bits). Organizations that required more than 256 hosts were given class B addresses; these organizations further structured their Class B addresses ("subnetting") for internal routing. Finally, Class B addresses began to run out. Thus larger organizations needed multiple Class C addresses. To reduce backbone router table size, the CIDR scheme [Rechter and Li 1993] now allocates larger organizations multiple *contiguous* Class C addresses that can be aggregated by a common prefix. In summary, the forces of subnetting and CIDR "supernetting" have led to the use of prefixes for the Internet. While the currently deployed IPv4 protocol uses 32-bit addresses, the next generation IPv6 protocol uses 128-bit addresses and continues to use longest matching prefix.

Best matching prefix is also used in the OSI Routing scheme [Perlman 1992]. Best matching prefix appears to be a fundamental problem for routers, and not just an artifact of the way the Internet has evolved.

1.2 Contributions

Our first contribution is a set of *generic* transformation techniques that can be applied to speed up *any* prefix matching lookup algorithm whose running time depends on the number of distinct prefix lengths in the database. Our major transformation technique, *controlled prefix expansion*, converts a set of prefixes with l distinct lengths to an equivalent set of prefixes with k distinct lengths for any $k < l$. Naive expansion can considerably increase storage. Thus, our second generic technique uses *dynamic programming* to solve the problem of picking optimal expansion

levels to minimize storage. Finally, our third generic technique, *local restructuring*, consists of three techniques that reorganize data structures to reduce storage and increase cache locality.

Our second contribution is to introduce two *specific* IP lookup algorithms, based on the transformation techniques, that are twice as fast as previous IP algorithms. Our first specific scheme, Expanded Tries, provides fast lookup times as well as fast update times, which may be crucial for backbone routers. The Expanded trie scheme can also be tuned to trade increased memory for reduced search times. Our second specific scheme, based on Binary Search on Levels, provides a scheme that scales well to IPv6 and yet has competitive search times for IPv4. Our schemes have been implemented by at least 3 companies and should be deployed in actual products by the end of 1998.

1.3 Outline of the Article

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 contribution by describing the three generic transformation techniques in Section 4. We describe how to apply our techniques to improve the performance of trie lookup in Section 5 to yield what we call expanded tries. We describe two versions of expanded tries, fixed stride and variable stride tries, and describe how to calculate optimal expanded tries that use the smallest possible storage for a given search speed. We also compare expanded tries more specifically to other multibit trie solutions of recent vintage. We then apply our techniques to improve the performance of Binary Search on Levels [Waldvogel et al. 1997] in Section 6. We briefly describe hardware implementation issues in Section 7. We 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 Databases

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 [Bradner 1997]: enterprise Routers (used in a campus or organization) and backbone routers (used by ISPs). The performance needs of these two types of routers are different.

Backbone routers today [Bradner 1997] 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. Because backbone routers typically run the Border Gateway Protocol [Rechter and Li 1995], and some implementations exhibit considerable instability, route changes can occur up to 100 times a second [Bradner 1997; Labovitz et al. 1997], requiring algorithms for handling route updates that take 10 msec. or less. Backbone routers may require frequent reprogramming as ISPs attempt to deal with customer requirements such as virus attacks. The distribution of packet sizes is bimodal, with peaks corresponding to either 64-byte 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 bytes.[4] However, large multicampus enterprise routers may look more like backbone routers.

Address space depletion has lead to 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 [Deering and Hinden 1995], 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 [Merit Network 1997] and are daily snapshots of the routing tables used at some major Network Access Points (NAPs). The largest of these, MaeEast (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 update times.

2.2 Implementation Model

In this article, we will compare lookup algorithms using a software platform. Software platforms are more flexible and have smaller initial design costs. For example, BBN [Internet-II 1997] uses DEC Alpha CPUs in each line card. However, hardware platforms are higher performance and cheaper after volume manufacturing. For example, Torrent [Internet-II 1997] uses a hardware forwarding engine. We will briefly discuss hardware platforms in Setion 7.

Thus for the majority of this article, we will consider software platforms using modern processors such as the Pentium [Intel 1998] and the Alpha [Sites and Witek 1995]. These CPUs execute simple instructions very fast

[4]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.

(few clock cycles) but take much longer (thirty to fifty clock cycles) to make a random access to main memory. The only exception is if the data is in either the Primary (L1) or secondary Cache (L2), 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 is expensive but fast Static Memory (SRAM, 10-20 nsec.). When a READ is done to memory of a single word, the 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 an approximate measure of the speed of any lookup algorithm, is the number of main memory (DRAM) accesses required, because these accesses often dominate search times. To do so, 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 chose a commodity 300MHz Pentium II running Windows NT that has a 8KB L1 data cache, a 512KB L2 cache, and a cache line size of 32 bytes. Since prefix databases are fairly large and the L1 cache is quite small, we (pessimistically) chose to ignore the effects of L1 caching. Thus our model assumes that *every* access leads to a L1 cache miss. When there is a L1 miss, the time to read in the first word in the cacheline from the L2 cache is 15 nsec. When there is a L2 miss, the total time to read in the word from memory (including the effects of the L2 miss) is 75 nsec. While many of our experiments use a 512KB L2 cache size, we also present a few projected results assuming an L2 cache size of 1024KB.

We chose the Pentium 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. We have implemented several previously known schemes on this platform. Further, we use a tool called Vtune [Intel 1997] that gives us access to dynamic instruction counts, cache performance, and clock cycles for short program segments. Thus for careful analytical worst-case comparisons, we use speed measurements given to us by Vtune. In addition, we also do a test of average speed using accesses to a million randomly chosen IP addresses.

The analytic worst-case measurements we use are much more conservative than what would be obtained by actually running the program on the Pentium II PC. This is because we assume that the L1 cache always misses, that branch prediction always fails, and that the worst possible branching sequence in the code is taken. A second reason for using Vtune is the difficulty of otherwise measuring the time taken for short code segments. A standard technique is to run C lookups to the same address D, measure (using the coarse-grained system clock) the time taken for these lookups, and then to divide by C to estimate the time taken for a single lookup.

Unfortunately, such measurements are optimistic because repeated lookups to the same address are likely to result in the few relevant parts of the data structure (i.e., the parts used to lookup D) entering the L1 cache.

3. PREVIOUS WORK

We describe previous schemes for IP lookup and compare them using our software 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 article, 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).

3.1 Classical Schemes

The most commonly available IP lookup implementation is found in the BSD kernel, and is a radix trie implementation [Sklower 1991]. If W is the length of an address, the worst-case time in the basic implementation can be shown to be $O(W)$. Thus the implementation can require up to 32 or 128 worst-case costly memory accesses for IPv4 and IPv6 respectively. Even in the best case, with binary branching and 40,000 prefixes, this trie implementation can take $\log_2(40{,}000) = 16$ memory accesses. A modified binary search technique is described in Lampson et al. [1998]. However, this method requires $O(\log_2 2n)$ steps, with n being the number of routing table entries. With 40,000 prefixes, the worst case would be 17 memory accesses. Using our crudest model, and 75 nsec. DRAM, a trie or binary search scheme will take at least 1.2 usec.

3.2 Hardware Solutions and Caching

Content-addressable memories (CAMs) that do exact matching can be used to implement best matching prefix. A scheme in McAuley et al. [1995] uses a separate CAM for each possible prefix length. This can require 32 CAMs for IPv4 and 128 CAMs for IPv6, which is expensive. It is possible to obtain CAMs that allow "don't care" bits in CAM entries to be masked out. Such designs only require a single CAM. However, the largest such CAMs today only allow around 8000 prefixes. While such a CAM may be perfectly adequate for an enterprise router, it is inadequate for a backbone router. Finally, CAM designs have not historically kept pace with improvements in RAM memory. Thus any CAM solution runs the risk of being made obsolete in a few years by software running on faster processors and memory.

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). This potentially dilutes the cache with hundreds of addresses that map to the same prefix. Also, typical backbone routers may expect to have hundreds of thousands of flows to different addresses. Some studies have shown cache hit ratios of around 50–70 percent [Newman et al. 1997]. Caching can help but does not avoid the need for fast lookups.

3.3 Protocol-Based Solutions

The main idea in Protocol Based solutions (IP and Tag Switching) is to replace best matching prefix by an exact match by having a previous hop router pass an index into the next router's forwarding table. This leads to a much faster lookup scheme (one memory access), but the cost is additional protocol and potential set up delays. IP switching [Newman et al. 1997; Parulkar et al. 1995] relies on switching long lived flows. This solution may be ineffective with short-lived flows such as web sessions. Tag switching [Chandranmenon and Varghese 1996; Rekhter et al. 1996] does not work at the boundaries of administrative domains. Both schemes require large parts of the network to make the required protocol changes before performance will improve. Both schemes also increase the vulnerability of an already fragile set of Internet routing protocols (see Labovitz et al. [1997]) by adding a new protocol that interacts with every other routing protocol. Also, neither completely avoids the BMP problem.

3.4 New Algorithms

Three new techniques [Degermark et al. 1997; Nilsson and Karlsson 1998; Waldvogel et al. 1997] for best matching prefix were discovered in the last year. The Lulea Scheme [Degermark et al. 1997] is based on implementing multibit 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. The LC trie scheme [Nilsson and Karlsson 1998] is also based on implementing multibit tries but compresses them using what the authors call level compression. We defer a detailed comparison of our multibit trie scheme to the LC trie and the Lulea scheme until Section 5.

The second Binary Search on Levels scheme [Waldvogel et al. 1997] is based on binary search of the possible prefix lengths and thus 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 [Internet-II 1997] and Torrent[5] use schemes based on hashing that claim good average performance but have poor worst-case times (16 memory accesses for the Torrent ASIK scheme).

3.5 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 using Patricia tries into our Pentium platform; we also implemented six-way search [Lampson et al. 1998] and binary search on hash tables [Waldvogel et al. 1997] using the largest (MaeEast) database. We project the worst-case evaluation presented in Degermark et al. [1997] and the average case

[5]http://www.torrent.com.

Table I. Prefix Databases as of September 12, 1997

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

Table II. Lookup Times for Various Schemes on a 300MHz 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. Note that for the Lulea scheme, the numbers have been recalculated with 300MHz clock and 15 nsec. latency L2 cache. For binary search on levels, memory is estimated assuming minimal perfect hash functions and $\log_2 = 5$ hashes.

	Average (24-Bit Prefix) (nsec.)	Worst Case (nsec.)	Memory Required for MaeEast Database (KB)
Patricia trie	1500	2500	3262
Six-way search on prefixes	490	490	950
Binary Search on Levels	250	650	1600
Lulea scheme	349	409	160
LC trie scheme	1000	—	700

numbers in Nilsson and Karlsson [1998] to the 300MHz Pentium II platform with 15 nsec. L2 cache (recall that 15 nsec. is the time taken for the first word in the cacheline to be read when there is a L1 cache miss). Projections were done using the numbers reported in the papers describing the schemes, and by scaling the earlier results to account for the (faster) clock cycle times in our platform. The results are shown in Table II. Throughout the article, for performance measurements we use the databases in Table I, which were obtained from Merit Network [1997].

3.6 Summary

The measurements in Table II 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 Levels scheme takes around 650 nsec. Thus using a software model and a commodity processor, the new 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 [Bradner 1997], 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 average packet size assumptions. If

traffic patterns change, and the average packet size goes down to say 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 considerable improvement in lookup times to around 2 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 Degermark et al. [1997] and Waldvogel et al. 1997] 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.

4. NEW TECHNIQUES

We first describe the three generic 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 a smaller number of distinct lengths. We then present our second technique which uses *dynamic programming* to solve the problem of picking optimal expansion levels to reduce storage. Finally, we present our third technique, *local restructuring*. We will apply these three techniques to tries in Section 5 and to Binary Search on Levels in Section 6.

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 [Perlman 1992].

Thus if we could restrict IP prefix lengths we could get faster search. But the backbone routers we examined [Merit Network 1997] 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.*

Suppose we have the IP database shown in the left of 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

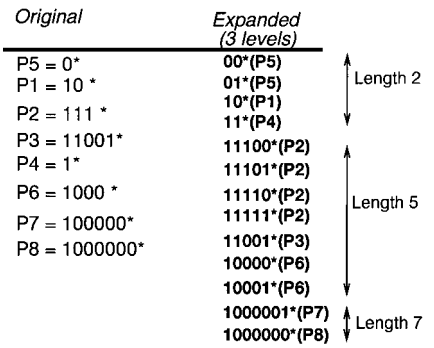

Fig. 1. Controlled Expansion of the Original Database shown on the Left (which has 7 prefix lengths from 1. . . 7) to an Expanded Database (which has only 3 prefix lengths 2, 5 and 7). Notice that the Expanded Database has more prefixes but fewer distinct lengths.

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 $r > m$. 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 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 in this terminology, we 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 $L_1, L_2, \ldots, L_k$. The complete expanded database is shown on the right of 1 together with the original prefix that each expanded prefix descends from. For example, 11101* descends from P2 = 111*.

Expansion can be performed in time proportional to the number of expanded prefixes. The algorithm uses an array A and maintains an invariant that $A[i]$ contains a pointer to the set of current prefixes of Length i. Initially, we load the original prefixes into A to preserve the invariant. We also take as input the sequence of expansion levels L_1, . . . L_k. The main loop scans the lengths in A in increasing order of length. If we are currently at length i and length i is in the target sequence of levels, we skip this length. Otherwise, we expand each prefix in set $A[i]$ to length $i + 1$ while respecting prefix capture.

4.2 Picking Optimal Expansion Levels

Expansion by itself is a simple idea. Some papers (e.g., Waldvogel et al. [1997]) have proposed using an initial Y bit ($Y = 16$ 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 Y are expanded to prefixes of length Y. 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 now briefly introduce the optimality problem.

In controlled expansion, we did not specify how we pick target prefix lengths (i.e., $L_1, \ldots L_k$). 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? In what follows, we will sometimes use the word "levels" in place of "target lengths."

The optimization criterion is to to pick the levels that minimize total storage. To introduce the problem, assume for now that storage is measured by the total number of expanded prefixes. A naive algorithm would be to search through all possible k target expansion lengths. But assuming IPv4 and $k = 6$, there are $32!/((26!)(6!))$ possible choices. The situation degenerates further when we go to 128-bit addresses. A second naive technique that works reasonably (but not optimally) is to use heuristics like picking the highest density lengths. However, when we experimented with real databases we often found (see results in Section 5.4 and Section 6.4) greatly reduced storage when the optimization programs picked "odd" lengths such as 18. Lengths like 18 appear to not slow down search appreciably despite alignment issues; in hardware, they do not matter at all, as long as the hardware has a barrel shifter.

Thus instead we pick optimal levels using dynamic programming [Cormen et al. 1990]. Figure 2 shows an example for IPv4, assuming we wish to minimize the total number of expanded prefixes using only k levels. In the first step, we use the input prefix database to compute a histogram of prefix lengths. The length of the black horizontal lines are proportional to the number of prefixes at a given length; note the density at 16 and 24. Now we know that the last level L_k must be at 32. In the second step, we reduce the problem to to placing level L_{k-1} (at say length x) and then covering the remaining lengths from 1 to $x - 1$ using $k - 1$ lengths. If we compute the storage required for each value of x, in the third step we can choose the minimal value of x and obtain the final levels.

Simple recursive solutions can lead to repeated subproblems and high running times. Instead, we use dynamic programming to build up problem instances and store the solutions in a table. It is not hard to see in the above example that we only have roughly Wk subproblems and roughly $W^2 \cdot k$ steps. This yields a fast algorithm.

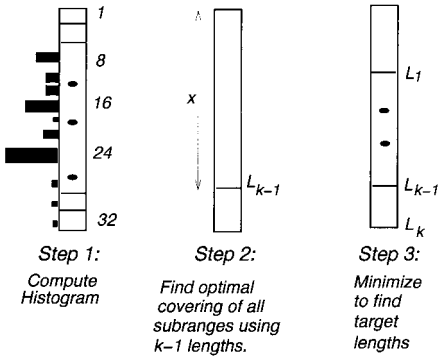

Fig. 2. Picking optimal levels using Dynamic Programming. Essentially, we have to cover all lengths from 1 to 32 using k target levels. This can be reduced to the problem of placing length L_{k-1} at length x and covering the remaining lengths $1 \ldots x - 1$ with $k - 1$ levels. We then minimize across all x to pick the optimal value.

As a first example above, we minimized the number of expanded prefixes. In what follows the real criterion is *to minimize the required data structure memory*. For example, with tries (Section 5), some prefixes can cause the addition of a large trie node while others can share trie nodes; when we use binary search on levels (Section 6), some prefixes can add auxiliary entries called markers. Because the minimization criterion will differ depending on the application, the specific dynamic programming technique will also vary, though with roughly the same structure.

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.

4.3.1 *Leaf-Pushing*. Consider a tree of nodes (see left side of Figure 3) 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 3, 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 (as in tries) 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 3 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 nonnull pointer, but not both. Thus we have reduced storage and improved locality. However, there is a cost for incremental rebuilding: information changes at a node close to the root can potentially change a large number of leaves.

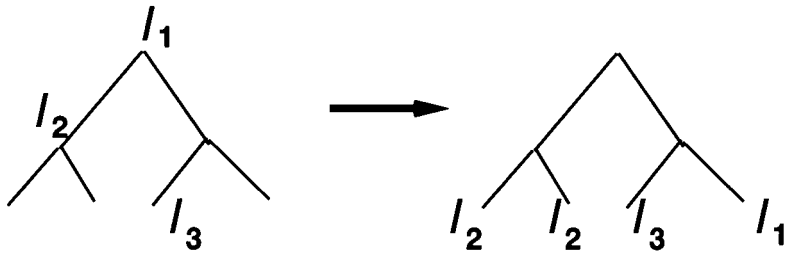

Fig. 3. Leaf-pushing pushes the final information associated with every path from a root to a leaf to the corresponding leaf. This saves storage because each nonleaf node need only have a pointer, while leaf nodes (that used to have nil pointers) can carry information.

4.3.2 *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 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 [Degermark et al. 1997]. We refer to such compressed arrays as *packed arrays*.

A second application is for perfect hashing [Cormen et al. 1990]. 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 *semiperfect 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

In this section we apply the techniques of Section 4 to tries. In Section 5.1, we first review the standard solution for one-bit tries, describe how expansion can be used to construct multibit tries, and finally show how to use leaf-pushing and packed nodes to reduce trie storage. In Sections 5.2 and 5.3 we show how to use dynamic programming to pick the number of bits to sample at each level of the trie in order to minimize storage. The difference between the two sections is that Section 5.2 makes the simpler assumption that all the trie nodes at the same level use the same number of bits (i.e., are *fixed stride*), while Section 5.3 considers *variable stride* tries that do not make this assumption.

Next, Section 5.4 describes measurements based on Vtune that quantify the trade-off between fixed and variable stride tries, as well as the trade-off between fast search times, fast update times, and small storage. Section 5.5 uses the results of Section 5.4 to pick an Expanded Trie structure that meets the desired criteria. Finally, Section 5.6 compares Expanded Tries in more detail with two other popular multibit trie schemes, Lulea tries [Degermark et al. 1997] and LC tries [Nilsson and Karlsson 1998].

5.1 Expanded Tries

A one-bit trie [Knuth 1973] 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 N is P, then the subtree rooted at N stores all prefixes that begin with P. Further, the 0-pointer at node N points to a subtree containing (if any exist) all prefixes that begin with the string $P0$; similarly, the 1-pointer at node N points to a subtree containing (if any exist) all prefixes that begin with the string $P0$.

Using our crude performance model, one-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 one bit. The central difficulty is that if we navigate in say strides of eight bits, we must ensure that we do not miss prefixes that have lengths that are not multiples of 8.

We have already seen the solution in Section 4. For example, suppose we want to traverse the trie two bits at a time in the first stride, three bits in the second stride, and two bits in the third stride. If we consider the database shown on the left of Figure 1, we have already expanded it to prefixes of lengths 2, 5, and 7 on the right. Since all prefixes have now been expanded to be on stride boundaries, we need not be concerned about missing prefixes.

Figure 4 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 two bits, the second uses three bits (5–2), and the third uses two bits (7–5). If a trie level uses m bits, then each trie node at that level is an array of 2^m locations. Each array element contains a pointer Q (possibly null) and an expanded Prefix P. Prefix P is stored along with pointer Q if the path from the root to Q (including Q itself) is equal to P.

Thus in comparing Figures 1 and 4 notice that the the root trie node contains all the expanded prefixes of length 2 (first four expanded prefixes in Figure 1). Now consider the first expanded prefix of length 5 in Figure 1, which is 11100* (P2). This is stored in a second level trie node pointed to by the pointer corresponding to the 11 entry in the first trie level, and stored at location 100.

5.1.1 *Search*. To search for destination address, 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

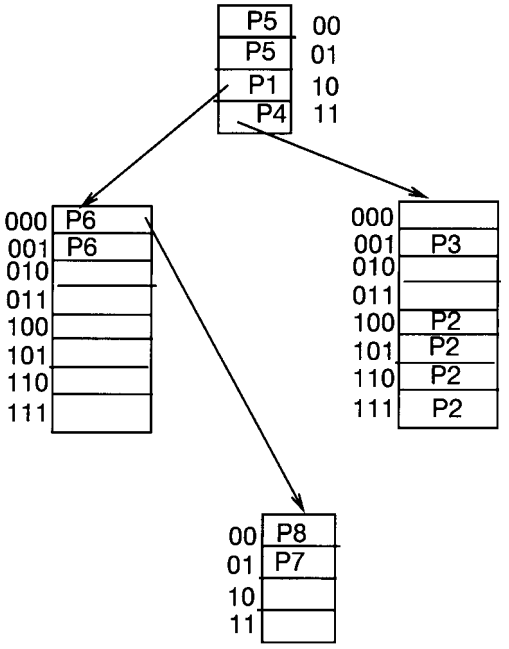

Fig. 4. Expanded trie corresponding to the database of 1. Note that the expanded trie only has a maximum path length of 3 compared to the one-bit trie that has a maximum path length of 7.

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 an 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*. We assume also that each node The search code is shown in Figure 5. 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 to layout the prefix and pointer fields associated with entry i in adjacent locations. In fact, in our implementation, both fields were packed within a single 32-bit word. This reduces search time to one memory reference per trie node.

5.1.2 *Insertion and Deletion.* Simple insertion and deletion algorithms exist for multibit tries. Consider Figure 4 and consider the addition of a prefix P9 = 1100*. We first simulate search on the string of bits in the new prefix up to and including the last complete chunk in the prefix. We will either terminate by ending with the last (possibly incomplete) chunk or by reaching a nil pointer. Thus for adding 1100* we follow the 11 pointer corresponding to chunk 11 and we terminate at the rightmost trie node at Level 2, say node N.

```
SEARCH(D) (* search for destination address D *)
  BMP := default; (* best matching prefix encountered on path so far *)
  N := root; (* current trie node we are visiting is root *)
  While (N ≠ nil) do
    C := chunk of D of length N.stride starting from bit N.startBit
    If N[bmp, C] ≠ nil then
      BMP := N[bmp, C]; (* found a longer prefix *)
    EndIf;
    N := N[ptr, C];
  Endwhile
  return(BMP);
```

Fig. 5. Code for Multibit Trie Search for best matching prefix of a destination address.

For the purposes of insertion and deletion, for each node N in the multibit trie, we maintain a corresponding one-bit trie with the prefixes that are stored in N. (This auxiliary structure need not be in L2 cache but can be placed in in main memory; this may, however, be hard to control.) Also, for each node array element, we store the length of its present best match. Now we expand the new prefix to be added and scan the array elements in the node that it affects. If we find that the present best match of an array element is from a prefix of smaller length, then we update it with the new prefix and new prefix length.

When we add 1100* and we end with the last incomplete chunk 00, we have to insert the possible expansions of this chunk into the trie node N. We then atomically link in the recomputed node into the fast trie memory.

If we end before reaching the last chunk, we have to insert new trie nodes. For example, if we add the prefix P10 = 110011*, we will find that we fail at node N when we find a nil pointer at the 001 entry. We then have to create a new pointer at this location which points to a new trie node that contains P10. We then expand P10 in this new trie node.

The insertion code Figure 6 uses the function FINDSTRIDE to determine the stride of a new node to be added. Any function can be used for this. Using a function that is dependent only on the trie level produces a fixed stride trie, while one that depends on the position in the trie produces a variable stride trie. The prefixes that end in a node N are stored as a set $N.prefixSet$, and this set of prefixes is organized as a one-bit trie. ADDTONODESET, REMOVEFROMSET, and FINDLONGESTMATCH operate on this one-bit trie. Once the node N in which the prefix P falls is determined, it has to be expanded to the *stride* of that node. Pseudocode for this is presented in Figure 7. For the purposes of expansion, there is an extra field $N[length, i]$ which contains the length of the prefix (if any) stored with entry i. This allows the expansion code to easily tell if a newly inserted prefix should capture (and hence overwrite) the stored prefix value corresponding to entry i. Note that $N[length, i]$ can be stored in auxiliary storage that is used only by the update code. The algorithm for deletion is similar. Once

```
INSERT(P) (* Insert a prefix P into the trie*)
  If root = nil initialize root;
  N := root;
  While (N.startBit + N.stride ≤ length(P) )
    C = N.stride bits of P starting from N.startBit;
    N' := N[ptr, C];
    If (N' = nil) (* have to create a new node to store P *)
      stride := FINDSTRIDE(N, C); (* find stride of new node *)
      N' := N[ptr, C] := ALLOCATENODE(stride); (* allocate memory for node *)
      N'.startBit := N.startBit + N.stride;
      N'.stride := stride; (* stride of newly created node *)
    EndIf;
  N := N';
  EndWhile;
  (* Now N is pointing to the last node in the path. The remainder *)
  (* of the prefix has to be expanded in node X *)
  INSERTINNODE(N,P); (* expand prefix P into node N *)
return;
```

Fig. 6. Code for insertion of a prefix P into a multibit trie. W is the length of a full address. Note that the code allows the stride of a node to be assigned using any function. For fixed stride tries, it will be a function of the trie level alone.

the trie node N in which the prefix P being deleted falls in is found, P has to be removed from that node and its expansions have to be changed to point to the next best match. So the best match P' of P in node N (after P is removed) is found, and the expansions of P that presently have P as the best matching prefix are updated to P'. Code for deletion is not presented for brevity. Deletion also deallocates memory once it finds that a trie node is not being used by any more prefixes.

The complexity of insertion and deletion is the time to perform a search ($O(W)$) plus the time to completely reconstruct a trie node ($O(S_{max})$ where S_{max} is the maximum size of a trie node.) For example, if we use eight-bit trie nodes, the latter cost will require scanning roughly $2^8 = 256$ trie node entries. On the other hand, if we use 17-bit nodes, this will require potentially scanning 2^{17} trie nodes entries. On a Pentium with a *Write Back* cache, while the first word in a cache line would cost 100 nsec. for the read, the writes would be only to cache and the entire cache line will take at most 200 nsec. to update. A single cache line holds 10 entries, so at 20 nsec./entry we require 2.5 msec. to write 2^{17} entries.

The average times will be much better, as most of the prefixes are 24-bit prefixes. Addition or deletion of such prefixes involves a search through the auxiliary one-bit trie (searching through the one-bit trie has a worst case of $O(W)$ memory accesses) and affects exactly one array element. (This is because all our databases contain a majority of 24-bit prefixes, and the dynamic programs in our experiments always choose a level at 24 bits.) Thus doing these W memory accesses and modifying one location takes

```
INSERTINNODE(N, P) (* Insert P into the node and expand*)
  ADDTONODESET(N.prefixSet, P);
  (* first find the smallest and largest expansions values of P *)
  startValue := P/(N.startBit, length(P)) << (N.startBit + N.stride − length(P) − 1); (* smallest *)
  endValue := startValue + 2^(N.startBit+N.stride−length(P)−1) − 1; (* largest expanded value *)
  j := startValue;
  While (j ≤ endValue) (* examine all expansions of P *)
    If (N[length, j] < length(P)) (* is prefix P longer? *)
      N[length, j] := length(P);
      N[bmp, j] := P; (* prefix P captures entry j *)
      j := j + 1;
    EndIf;
  EndFor;
return;
```

Fig. 7. Code for inserting a prefix P into trie node N. P is added to the list of prefixes that fall in N, and then expanded in N to overwrite those entries that currently map to a shorter prefix. The notation P / $(m1..m2)$ is used to denote the binary number formed by the bits $m1$ through $m2$ of P. The notation $X << Y$ is used to denote the number obtained by left shifting X by Y places.

approximately 3 usecs. Though the average update time is only 3 usecs., we still need to bound the worst-case update time. To get a worst-case update time of 2.5 msec., we saw in the previous paragraph that we have to limit the maximum stride to 17. *Thus in our dynamic programs in 5 and 5, we constrain the nodes to be of maximum stride* 17*, so that the worst-case criterion for insertion/deletion times is met.*

5.1.3 *Route Change.* Route change involves searching for the prefix and then updating the corresponding entry in the intermediate table. This takes at most 3 usecs.

As a baseline measurement, we present the time taken for insertion and for leaf-pushing for the various databases for a four-level (i.e., eight bits at a time) trie in Table III. Note that the average insertion time for MaeEast is 170 msec./38816 = 4 usec.

5.1.4 *Applying Local Restructuring.* Leaf-pushing (Section 4) can also be useful to reduce storage by a factor of two. If we leaf-push the multibit trie of Figure 4 we will push the *bmp* value (i.e., P1) in the 10 entry of the root node down to all the unfilled entries of the leftmost node at Level 2, say Y. Leaf-pushing can easily be accomplished by a recursive algorithm that pushes down *bmp* values along pointers, starting from the root downward. Unfortunately, it makes incremental insertion slower. We will discuss this trade-off in Section 5.4.

5.2 Optimal Expanded Trie with Fixed Strides

So far we have not described how to use dynamic programming to optimize trie levels. The obvious degree of freedom is to pick the expansion lengths

Table III. The Memory, Random-Search Times, and Insert Times for a Four-Level (i.e., a fixed stride length of eight bits) Trie Built on the Four Sample Databases. The random-search times were measured assuming all IP addresses are equally likely.

Database	Number of Nodes	Memory (KB) (Nonleaf-Pushed)	Time (msec.)		Memory after Leaf-Push	Time for Random Search
			Build	Leaf-Pushing		
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.

$L_1 \ldots L_k$ which determine the strides through the trie. This assumes that all trie nodes at Level j have the same stride size. We will restrict ourselves to fixed stride tries in this subsection. Fixed stride tries may be desirable because of their simplicity, fast optimization times, and slightly faster search times. 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 fixed stride 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. For example, for the trie in Figure 4 with 4 nodes, two of stride two bits and the other two of strides three bits each, the total number of array elements is $2^2 + 2^2 + 2^3 + 2^3 = 24$. We assume that each array element uses 1 word (4 bytes) of memory (this is true after leaf-pushing). Hence the cost of this trie is 24.

For finding the cost of a given set of levels, just using a histogram as in Figure 2 is insufficient. This is because one prefix can be an extension of a second prefix. For example, if P1 = 00* is a prefix and P2 = 000* is a second prefix that is an extension of P2, expanding both prefixes to length 5 would incur a cost of 8. This is only the cost of expanding 00*. This is because when we expand 00* to length 5 we already create entries for all the expansions of 000* to length 5. Thus if we added the cost of expanding P2 in isolation from length 3 to length 5 (cost equal to $2^2 = 4$) we would be double counting. On the other hand, if we had P1 = 00* and P3 = 111*, the true cost of expanding both to length 5 is indeed $8 + 4 = 12$.

Thus we build a auxiliary one-bit trie. This can help us determine prefix extensions and hence the correct costs. After building the one-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 one-bit trie. From now on we will distinguish the input levels (*trie levels*) from the output levels (*expansion levels*). Note that if all the one-bit nodes at trie level $i + 1$ are replaced by $j > i$ bit nodes, then all the one-bit nodes in trie levels $i + 1$ to $i + j$ can be ignored.

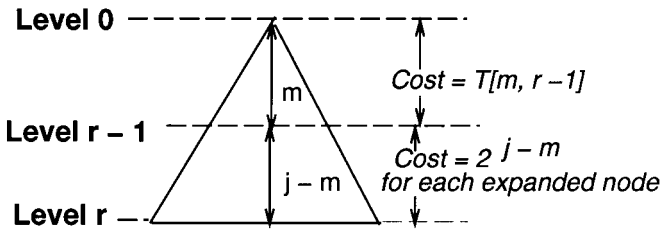

Fig. 8. Covering trie lengths from 0 to j using expansion levels 1 to r can be solved recursively by first placing Level r at length j and choosing Level $r - 1$ to be at some optimal length m. Finally, we recursively solving the covering problem for the upper portion of the tree using $r - 1$ levels.

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 determining the optimal cost $T[j, r]$ of covering levels from 0 to say j using expansion levels 1 to r. Expansion level r is constrained to be at trie level j. Recursively, as in 2 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 8.

Using Figure 8 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 worst-case memory accesses required to search the trie 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.

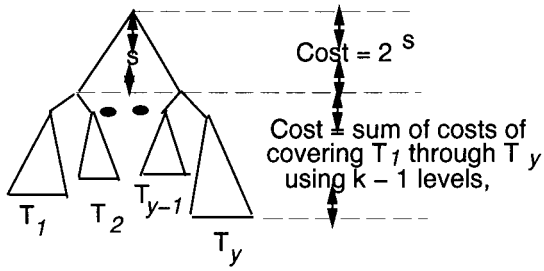

Fig. 9. Covering a variable stride trie using k expansion levels. We first pick a stride s for the new root node and then recursively solve the covering problem for the subtrees rooted at Trie Level $s + 1$ using $k - 1$ levels.

5.3 Optimal Expanded Trie Using Varying Strides

In the last section, we considered the expansion lengths to be a degree of freedom that could be optimized to minimize storage. However, there is a further degree of freedom. We can 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 can now have a different stride size that is encoded along with the pointer that leads to child. In this section, we describe how to construct storage-optimal tries that use varying strides.

Assume as before that we have built an auxiliary one-bit trie, and we wish to convert this into a variable stride trie that takes at most say k memory accesses. Generally k is determined by the software designer based on the required lookup speed. For example, if we are using 100 nsec. DRAM, and we wish a lookup speed of around 300 nsec., we would set $k = 3$. Note that it is easy to take all the dynamic programming problems we have and use them in a different way: given a fixed amount of memory, find the fastest trie (i.e., smallest value of k) that will fit into memory.

We are given the maximum number of memory accesses k. Let R be the root of the one-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 9. But each such subtrie can be recursively covered using the same algorithm.

Once again, we pick the value of s by taking the minimum over all values of s. Define $height(N)$ as the length of the longest path from N to any of its descendants. Thus $height(N) = 0$ if N is a leaf node. Let us denote the descendants of a node N in the one-bit trie at a height s below N by $D_s(N)$.

If we denote by $Opt(N, r)$, the optimal solution for a node N using r expansion Levels we get the following recursion:

$$Opt(N, r) = min_{s \in \{1. . . 1+height(R)\}}\{2^s + \sum_{M \in D_{s+1}(R)} Opt(M, r - 1)\}$$

$$Opt(N, 1) = 2^{(1+height(N))}$$

Table IV. Memory Requirement (in kilobytes) for Different Number of Levels in a Leaf-Pushed Trie Using the Fixed Stride Dynamic Programming Solution. The time taken for the dynamic program is 1 msec. for any database.

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

To decipher the above equation, notice from Figure 9 that the cost of the first level is 2^s. The remaining cost is the cost of covering the subtrees T_1 through T_y in Figure 9 using $r - 1$ levels. Finally, the roots of the subtrees T_1 though T_y are the members of the set D_{s+1}.

Once again, we can use the above equation to generate a dynamic programming solution. We will construct $Opt(N, l)$ for all the nodes in the original trie in increasing values of l and store the results in a table. The original problem we must solve is the problem $Opt(R, k)$. It only remains to describe how we efficiently compute $D_h(N)$ the set of descendants of a node at height h.

To do so we first preprocess the trie to assign Depth First Search (DFS) numbers. This is a standard algorithmic technique [Cormen et al. 1990] where we walk the trie in DFS order, keeping a counter that increments for each visit. The advantage of DFS numbering is that it allows us to easily enumerate the descendants of a node.

The final complexity of the algorithm is $O(n*W^2*k)$ where n is the number of original prefixes in the original database, W is the width of the destination address, and k is the number of expansion levels desired. This can be explained as follows. We do one pass for each value of the number of levels from 1 to k. In a single pass, we fix one level, while the other levels have to be varied among W levels. Thus there are $O(W^2)$ such pairs. When two levels are fixed, the list of nodes in both levels has to be traversed once. The number of nodes in each level is at most $O(n)$. So the overall complexity (considerably overestimated) is $O(n*W^2*k)$. Notice that the complexity has grown considerably over the fixed stride dynamic program.

5.4 Measured Performance of Expanded Tries

We now present results obtained by using the various dynamic programs. We use the prefix databases shown in Table I for all our experiments.

5.4.1 *Performance of Fixed Stride Dynamic Program.* Table IV shows the variation in memory for the four databases as we use the fixed stride dynamic program and vary the maximum number of levels in the final multibit trie. The time taken for the dynamic program to run itself is very

Table V. Memory Requirement (in kilobytes) Using the Variable Stride Dynamic Programming Solution for Leaf-Pushed Tries, and the Time Taken for the Dynamic Program to Run in Milliseconds. A non-leaf-pushed trie will have twice the memory requirements as the entries in this table.

	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

modest: in all cases it takes 1 msec. to run.[6] As we would expect, the memory decreases as we increase the number of levels (of course, the search time will also increase).

The most interesting thing to note is that even the simplest dynamic program pays off very well. When compared to the straightforward solution of using equal lengths at every level, picking an optimal stride length saves us roughly a factor of 3 in storage. For example with four levels without using the dynamic program, we see from Table III that the memory required for the leaf-pushed trie is 2003KB; using optimal fixed stride lengths calculated by the dynamic program, we see in Table IV that the memory for four levels has decreased to 738KB. Given that an extra 1 msec. is easily affordable for insertion and deletion, this is a good trade-off.

5.4.2 *Performance of Variable Stride Dynamic Program.* Next, Table V shows the variation in memory for the four databases as we use the *variable* stride dynamic program and vary the maximum number of levels in the final multibit trie. The time taken for the dynamic program to run itself is included in the main table this time because the time increases roughly linearly with the size of the database: notice for instance that the computation time for MaeEast and two levels is roughly 2.2 times the time required for MaeWest, while the size of MaeEast is 2.75 times that of MaeWest. Recall that the complexity of the variable stride algorithm is $O(n*W^2*k)$. (The small discrepancy is probably because there is a portion of the computation time that does not depend on n, the number of prefixes.)

As we expect from the complexity estimates, the computation time grows with the number of levels. The computation time for MaeEast and two levels is fairly small (130 msec.) and yields a reasonable amount of memory (1559KB for the leaf-pushed trie). The computation time for MaeEast and four levels is much larger (almost 1.6 seconds). On the other hand, when compared to the fixed stride dynamic program, the variable stride program saves roughly a factor of two in storage. For example, the four-level leaf-pushed trie version of MaeEast now takes 423KB compared to 737KB (fixed stride program) and 2003KB (unoptimized version).

[6]This is because the complexity of the dynamic programming solution only depends on the number of target levels and the address width, and not on the size of the database.

Table VI. Memory Requirement (in kilobytes) Using the Variable Stride Dynamic Program, Leaf-Pushed and Allowing Packed Array Nodes

	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

Are variable stride tries worth the much larger table rebuilding times? First, they should work well for enterprise routers because route changes happen rarely and the storage reduction makes it more likely that the route table will fit in cache. Second, we have said that BGP instabilities may require prefix addition or deletion in the order of 10 msec. Does that mean that taking 1.6 seconds to calculate optimal variable stride tries makes variable stride tries infeasible in backbone routers?

However, note that many of the BGP insertions and deletions are because of pathologies (the same prefix being inserted and deleted by a router that is temporarily swamped [Labovitz et al. 1997]), and others are because of route changes [Labovitz et al. 1997]. On the other hand, the rate at which new prefixes get added (or deleted) by managers seems more likely to be in the order of days.

Suppose these assumptions are true and the dynamic program is run every day using the set S of *all* the prefixes that were present in the database during any period during the day (as opposed to using the prefixes in a snapshot of the database at some instant). Then the dynamic program will calculate the storage-optimal trie to store set S. Now during the next day, first suppose that managers do not add or delete new prefixes. Thus routing pathologies could cause prefixes to be added or deleted to result (at any instant) in some new set $S' \subseteq S$. But the storage required to store S' can only be less than that of S because S' can be obtained from S by deletions, and deletions can only decrease the storage required by a trie.

Next, if managers add a small set of prefixes R during the next day, this can result in a new set $S' \subseteq S \cup R$ at any instant. But if R is small, the resulting storage taken by S' can only be slightly more than the storage calculated for S. Thus despite the fact that the optimal strides have been calculated for S and not for S', the resulting loss of accuracy should be small. Thus running a dynamic program that takes several seconds to run every day (the programs can be run by an external agent, and the strides given to the table building process) appears to be feasible without affecting worst-case insertion and deletion times.

5.4.3 *Performance of Variable Stride Dynamic Program with Packed Nodes*. Table VI shows the effect of using packed nodes in addition to the variable stride program. Recall from Section 4.3 that a packed node is a trie node that is very sparsely populated. We chose to define a trie node with only 3 valid pointers as being sparse. In many of our multibit trie

Table VII. Memory Requirement (in kilobytes) Using the Variable Stride Dynamic Program, Non-Leaf-Pushed, and Allowing Packed Array Nodes

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

implementations, we have observed that the leaf trie nodes (and there are a large number of such leaf nodes) are very sparse. Thus replacing the sparse array with a packed array of 3 pointers (and using linear or binary search [Lampson et al. 1998] to locate the appropriate location) can save storage. Clearly this can slow down search. We chose a size of 3 so that the packed node could fit into a cache line on the Pentium; thus accesses to all the pointers within the packed node are cache accesses once the packed node has been initially referenced.

Notice from Table VI that we have some gain (roughly a factor of 2) over the variable stride trie (Table V) for two levels (983 versus 1559KB for MaeWest), but there are diminishing returns at higher levels (370 versus 423KB for four levels). This is because a variable stride trie can choose smaller strides for leaf nodes. The gains for using packed nodes with a fixed stride program (not shown) are more substantial.

5.4.4 *Variable Stride without Leaf-Pushing*. We have seen that leaf-pushing can reduce storage by roughly a factor of two but can increase insertion times. Thus Table VII describes the memory requirement if we limit ourselves to *not* using leaf-pushing, but using variable strides and packed nodes. This time we see that packed nodes provide more benefit. For example, from Table V we know that if we do not use leaf-pushing for MaeEast and four levels we should take twice the storage shown (i.e., two times 423 which is 846). However, with packed nodes the storage is 580, roughly a factor of 1.5 reduction.

5.4.5 *Variation of Search Times with Trie Level and Trie Type*. So far, all our results have only described the storage requirements (and time for the dynamic programs to run) for various types of tries. Intuitively, the smaller the number of levels the faster the search time. However, each type of trie will require different computation time per level; for example, variable stride tries require slightly more computation time because of the need to extract a stride as well as a next pointer.

We measured the worst-case paths of each of our trie programs using a tool called VTune [Intel 1997], 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. We show the Vtune results in Table VIII.

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

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	$(12k + 5)*clk + k*M_D$

As an example, we find that for one level of the leaf-pushed trie, the time taken is *One 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 time is $(6k + 4)*clk + k*M_D$, where *clk* is the clock cycle time (3.33 nsec. for the 300MHz Pentium), and M_D is the memory access delay. Similarly worst-case times were computed for each of the trie schemes.

Notice that leaf-pushed trie nodes take a smaller amount of time to process (because each location has either a pointer or information but not both) than non-leaf-pushed nodes, fixed stride nodes take a smaller amount of time to process than variable stride nodes (because of the need to extract the stride length of the next node in variable stride nodes), and ordinary nodes take longer to process than packed nodes. However, at a large clock rate (e.g., 300MHz) the difference between 12 clock cycles and 6 clock cycles (20 nsec.) is not as significant as the memory access times (75 nsec.). In hardware, the extra processing time will completely be hidden. Thus the search time differences between the various trie schemes do not appear to be very significant.

M_D depends on the data required being in the L1 cache, L2 cache or in the main memory. With the 512KB of L2 cache available in the Pentium Pro, we can sometimes fit an entire four-level trie and the intermediate table into the L2 cache. Thus our tables will sometimes show that the search times decrease as we increase the number of levels because the storage reduces and the trie fits completely into the L2 cache. The access delay from the L2 cache is 15 nsec. The delay for an access to the main memory is 60 nsec. When there is a L2 miss, the total time needed for an access is $60 + 15 = 75$ nsecs.

5.4.6 *Putting It Together*. From Tables IV–VII we use appropriate values to construct Tables IX and X which illustrate storage and search time trade-offs for the various species of tries we have discussed. Table IX assumes 512KB of L2 cache as in the Pentium Pro, while Table X is a projection to a Pentium that has 1MB of L2 cache. The values in these two tables are constructed from the earlier tables.

Note that instead of storing the next hop pointers directly in the trie, we use a level of indirection: the longest matching prefix in the trie points to

Table IX. Memory and Worst-Case Lookup Times for Various Types of Trie. With a 512KB L2 cache, when the memory required is less than 512KB, the table 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.

	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

an entry in an *intermediate table* that stores the actual next hop for forwarding. This helps in speeding up route changes, which are much more frequent than insert/deletes. This is because only one entry (corresponding to the prefix whose route is changing) has to be changed, and no expansion needs to be done. Thus we need to account for the intermediate table size (which is 2 bytes per prefix, and is thus 80KB for the MaeEast database) in computing memory requirements.

For example, the memory requirement for the MaeEast database for Non-Leaf-Pushed Variable stride trie for three levels is found from Table V to be 575*2 = 1150KB. We add the intermediate table size of 80KB to get 1230KB, which is the corresponding entry in Tables IX and X.

Tables IX and X estimate the memory access delay as follows. If the data structure is small enough to be fit in the L2 cache of 512KB, then the memory access delay M_D is taken to be 15 nsec.; if not, it is taken to be 75 nsec. All numbers assume that all accesses miss the L1 cache. The total memory requirement for each trie is calculated by adding the size of the trie that is built using the dynamic program, with the size of the intermediate table. When the total size is a little more than 512KB, we assume that only the first level of the trie is in the L2 cache, while the other levels lead to a L2 miss.

For a machine which is to be used as a router, we might be able to afford a larger L2 cache. For a machine with 1MB of L2 cache, the projected search times would be as in Table X.

5.5 Choosing a Trie

As an example, from Table IX, we can decide which type of trie to use for the MaeEast database. We see that the minimum search time of 196 nsec.

Table X. Memory and Time Taken for Various Types of Trie with 1MB L2 Cache

	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	153	660	188	550	223
Leaf-Pushed Variable stride	1640	219	655	148	500	196	460	244	450	293
Non-Leaf-Pushed fixed	98275	209	3780	311	1560	413	1240	514	1020	256
Non-Leaf-Pushed variable	3200	226	1230	341	920	216	840	271	820	326
Leaf-Pushed variable stride with packed array nodes	1063	183	540	164	450	206	427	248	420	289
Non-Leaf-Pushed variable with packed array nodes	1824	246	808	181	640	236	623	291	614	346

is attained for the leaf-pushed Variable Stride trie with four levels. Next, 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 500KB structure, this could take at least 10 msec. 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 msec. (the maximum stride is constrained to be 17 as described earlier to guarantee worst-case insert/delete times). 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 the structure being in the L2 cache).

Now we consider the same database using a machine with 1MB of L2 cache (Table X). Here we see that the minimum time is 148 nsec.; however this does not meet the worst-case deletion time requirements. In practice, it might work quite well because the average insertion/deletion times are a few microseconds compared to the worst-case requirement of 2.5 msec. However, we see that the Non-Leaf-Pushed variable stride trie with packed array nodes and three levels gives a worst-case bound of 181 nsec. and supports insert/delete with a worst-case bound of 2.5 msec. Similar tables can be constructed for other databases and a choice made based on the requirements and the machine that is being used.

5.6 Comparing Multibit Trie Schemes

The Lulea scheme [Degermark et al. 1997] uses a clever scheme to compress multibit trie nodes using a bitmap. While we have used compression on a limited scale (for sparse nodes with no more than three pointers), our compression scheme will not work well with large sparse trie nodes. The Lulea scheme uses a trie with fixed strides of 16,8,8; they are able to compress large trie nodes without using (slow) linear search using a fast

method of counting the number of bits set in a large bitmap. It appears that the Lulea scheme uses leaf-pushing; thus insertion and deletion will be slow. The Lulea scheme also optimizes the information associated with prefixes by noting that there are (typically) only a small number of next hops. Both the Lulea node compression and next-hop optimizations are orthogonal to and can be used together with our schemes.

The LC Trie scheme [Nilsson and Karlsson 1998] essentially constructs a variable stride trie[7] but with the special property *that all trie nodes are completely packed*. The LC trie implementation begins with a one-bit trie and forms the root node of the LC trie as the largest multibit trie node that is full (i.e., contains no null entries). Removing the root node breaks up the original one-bit trie into multiple subtries, and the process is repeated with each subtrie. Finally, the variable stride trie nodes are laid out contiguously in an array. Once again, this makes insertion and deletion slow.

When compared to these other schemes, our multibit trie schemes have greater tunability and faster insertion/deletion times. We have paid a great deal of attention to insertion and deletion times. But perhaps more importantly, our schemes are *tunable*: we can generate a collection of trie schemes with various trade-offs in terms of search, insertion times, and storage. Since the requirements of hardware and software designers on different platforms and in different environments can be so different, we believe this tunability is an important advantage. For example, the LC-trie scheme is limited to building a single variable stride trie (of depth 6 or 7) for each database of prefixes. By contrast, our schemes allow an implementor to add more memory to reduce search time.

6. FASTER BINARY SEARCH ON LEVELS

So far, we have applied the general ideas of expansion and optimal choice of levels to trie schemes. To show that our ideas are general, we apply our ideas to a completely different type of lookup algorithm based on repeated hashing. We start by reviewing the existing idea of binary search on levels [Waldvogel et al. 1997] in Section 6.1. We then describe in Section 6.2 how our new expansion ideas can reduce the worst-case time. Next, in Section 6.3 we describe a dynamic programming solution for picking optimum levels. Finally, in Section 6.4, we describe implementation and performance measurements for binary search on hash tables with expansion.

6.1 The Base Idea

We briefly review the main idea of binary search on levels [Waldvogel et al. 1997]. Binary search on levels should not be confused with binary search on prefixes described in Lampson et al. [1998]. The first step is to separate prefixes by lengths; more precisely, we keep an array $A[i]$ such that $A[i]$ points to all prefixes of length i. A simple way to search for a longest matching prefix for say address D is as follows. We start with the longest

[7]Our schemes were invented concurrently.

length nonempty set $A[max]$. We then extract the first max bits of D and search for an exact match with any element in set $A[max]$. This can be done efficiently if each set $A[i]$ is organized as a hash table (or a CAM in hardware). If we find a match, we find the longest matching prefix; otherwise, we backup to the next longest length $A[l]$ that has a nonempty set and continue the process until we run out of lengths. This scheme will take a worst case of 32 hashes for IPv4.

The main idea in Waldvogel et al. [1997] is to use binary search on levels. Rather than start searching for an exact match in the largest length table the scheme searches for an exact match in the *median* length table. If we find a match, we recursively search the right half (i.e., all lengths strictly greater than the median); if we do not find a match we recursively search the left half (i.e., all lengths strictly less than the median). This will result in a worst-case search time of $\log_2 32 = 5$ hashes for IPv4. However, the basic scheme needs additional machinery for correctness. The main mechanism that concerns us here is the addition of *markers*.

Consider the data structure shown in Figure 10 corresponding to the unexpanded database of Figure 1. Each prefix P1 to P7 is placed in a separate hash table (shown vertically using a dashed line) by length. Thus the length 1 hash table contains P4 and P5, and the length two database contains P1, etc. The bolded entries in the hash tables are what we call markers.

To see why markers are required, consider a search in the for an address D whose first five bits are 11001. Assume the bolded markers nodes do not exist in 10. Thus if we do a search in the length 4 table for the string 1100 (first four bits of D), we will not find a match (because there is no prefix 1100*). According to our binary search rules, we have to backup to the left half. But the correct answer, P3 = 11001*, is in the right half! To fix this problem, we add a bolded marker corresponding to the first four bits of 11001* (see Figure 10) to the Length 4 table. This will guide binary search to the right half if the destination address has first four bits equal to 1100.

It may appear that one has to have a marker for each prefix P of length L at all smaller lengths from $1 \ldots L - 1$. Actually, one needs only markers at lengths where binary search will search for P, and that are smaller than L. Thus if the largest length is 7, the prefix P8 = 1000000* only needs a marker 1000* in the Length 4 table and a marker 100000* in the Length 6 table (see Figure 10).[8] This leads to at most $\log_2 W$ hash table entries per input prefix, where W is the maximum prefix length.

This algorithm is attractive because it offers a very scalable solution ($O(\log_2 W)$ time and $O(n \log_2 W)$ memory), where n is the number of

[8]Note that some nodes such as 1000* and 100000* are markers as well as prefixes. This is shown in Figure 10 by making the entries bolded but also having them contain a prefix entry. Every entry also contain a precomputed *bmp* value that is the best matching prefix of the entry itself. These *bmp* values are used in Waldvogel et al. [1997] to prevent backtracking. However, they are not relevant to the rest of this article.

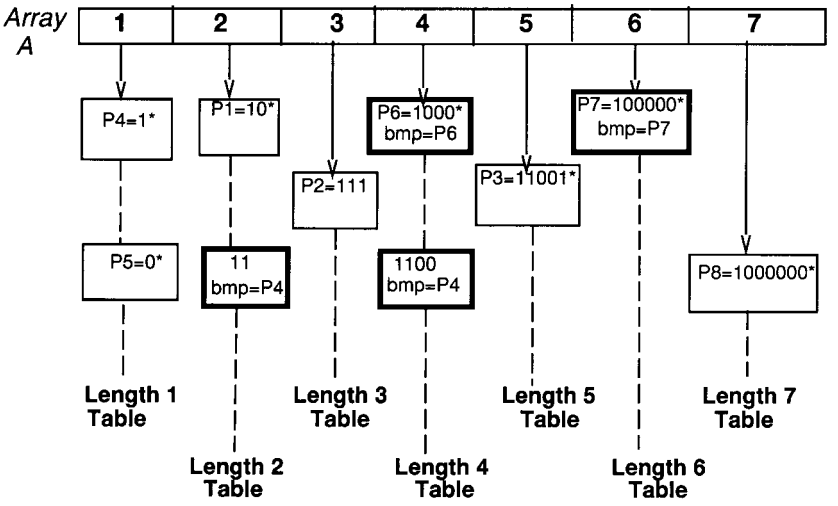

Fig. 10. The binary search database corresponding to the unexpanded prefix set of Figure 1. The marker entries are shown bolded.

prefixes. For IPv4, this algorithm takes a worst case of 5 hashes; for IPv6 with $W = 128$, the worst case only increases to 7 hashes. Waldvogel et al. [1997] describe another enhancement called Rope Search, and the idea of having an initial array; however, these improvements improve the average performance and not the worst case. There are no measurements in Waldvogel et al. [1997] except some estimates based on estimating the cost of hashing.

6.2 Faster Binary Search through Expansion

The speed of binary search on levels is actually $\log_2 l$, where l is the number of distinct prefix lengths. For the real backbone databases we examined, l 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 to $k < l$. 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.

Figure 11 shows the binary search database for the expanded set of prefixes shown on the right of Figure 1. Because the length of the set pointed to by $A[i]$ is not i, we have to store the length of the prefixes in the set along with the pointer to the set. The search will start in the median length set (Length 5 table). If we find a match we try the Length 7 set; otherwise we try the Length 3 set. When we compare with the unexpanded binary search database (Figure 10) we find that the number of distinct lengths has reduced from 7 to 3. Thus the worst-time number of hashes decreases from $\log_2(7 + 1) = 3$ to $\log_2(3 + 1) = 2$. On the other hand, the number of nodes has increased from to 10 to 13. The number of markers, however, actually *decreases.*

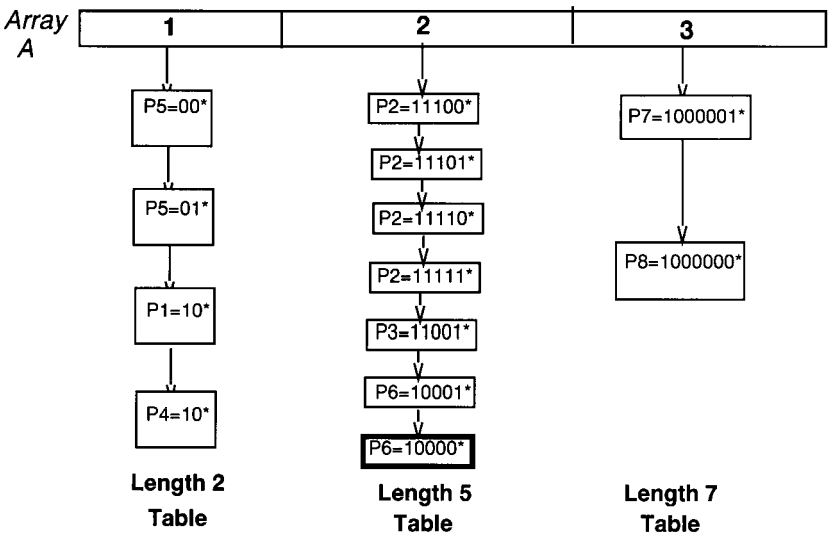

Fig. 11. The binary search database corresponding to the expanded prefix set of 1. The single marker node (which points to the presence of P7 and P8 at higher lengths) is shown bolded. Compare with the database of 10 and observe that the number of hash tables has decreased.

6.3 Applying Dynamic Programming

We have already seen that 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.

A further expansion to $W/4$ lengths however will improve the worst-case number of hashes to $\log_2 W - 2$ but increase the maximum number of hash table entries to $8^*n + n^*(\log_2 W/4 - 1) = n^*(5 + \log_2 W)$. Thus further expansion can potentially increase the worst-case storage by a factor of 5.

In practice, we have found that expansion from 23 distinct lengths to 8 lengths—and even 3 lengths—does not increase the storage by such dramatic factors. The key to this improvement is to use dynamic programming to pick the levels so as to balance the costs of expansion and the (secondary) cost of markers. For instance, a natural heuristic would be to pick the k expansion levels at the k most densely populated prefix lengths (e.g., 24, 16 etc.). However, suppose we had only two expansion levels and we tried to pick these levels as 24 and 16. If there are several 17- and 18-bit prefixes, the cost of expanding these to 24 bits may be quite high. Thus we have found that a dynamic programming solution produces much better results than the natural heuristics.

To this end, we define *ExpansionCost*$[i, j]$ as the number of prefixes obtained by expanding all prefixes length l such that $i \leq l < j$ to length j, with *ExpansionCost*$[j, j] = 0$ for all j.

We begin by filling the table *ExpansionCost*$[i, j]$. Let $n(i)$ denote the number of prefixes of length i. The number of prefixes of length j that would result from expanding all prefixes of length i to length j is given by $n(i)*2^{j-i}$. However, to find the number of prefixes that would end up in level j by expanding all prefixes of length $i \ldots j - 1$ to length j, we cannot simply compute $\Sigma_{m=i..j-1} n(m)*2^{j-m}$. This is because this does not take into account common prefixes. For example, while 0* and 00* expanded to length 3 would yield only 4 prefixes of length 3, the above sum would yield 6. (This is identical to the double counting problem described when we described the trie dynamic programming solution.)

ExpansionCost can be computed correctly by actually simulating the placement of hash entries for each possible expansion. However, this would be an exponential time algorithm.

Instead, we use our old workhorse, the auxiliary one-bit trie. When we are calculating *ExpansionCost*$[i, j]$ assume that *ExpansionCost*$[i + 1, j]$ has been calculated. Now we examine the nodes in trie level $i - 1$. For each node x, calculate the number of prefixes of length j that it would contribute. This can be done as follows. Assume each node has a 0-pointer and a 0-info field, and similarly a 1-pointer and 1-info field. (Recall that trie node locations without leaf-pushing contain information about prefixes alongside the pointers.) We find the expansion cost contributed by the 0 and 1 pointers separately and add them up.

If the 0-info value is nonnull, then this represents a valid prefix, and would expand to level j, giving 2^{j-i} prefixes of length j. If the 0-info value is null, then that does not correspond to a valid prefix. In this case, if the 0-pointer points to a nonnull value y, then we count the value contributed by y when expanded to length j. If y is null, there is no cost contribution. We do the same computation for the 1-pointer and then add up the total to find the total contributed by node x.

It is interesting to see why counting the cost of expansion is more complicated than for tries (see Figure 8). This is because every trie node must be accounted for in trie solutions, whether the trie node contains a prefix or not. In the binary search case, a trie node contributes to the final database only if it contains a valid stored prefix or if it contributes a marker. Since markers do not need to be expanded, we count markers separately.

The complexity for computing *ExpansionCost* can be shown to be $O(n*W^2)$). This is because for each of the $O(W^2)$ pairs of possible values of i, j we must examine potentially $O(n)$ nodes at a trielevel. We also need to bound the storage contributed by markers.

Let $Entries[j]$ be the maximum number of markers and entries that would be induced by prefixes of length j when a level is placed at length j. We can estimate this quantity by examining the nodes in the trie at trie level $j - 1$[9] and counting the number of pointers that go from level $j - 1$ to level j, and the number of *info* values stored at level $j - 1$.

If there is a prefix of length j, then must be an *info* value at trie level $j - 1$. On the other hand, a prefix of length greater than j, would contribute a nonnull pointer at trie level $j - 1$. Thus each *info* value corresponds to a prefix entry stored at level j; each nonnull pointer corresponds to a possible marker. This is a considerable over-estimate of the number of markers because it essentially assumes that each prefix of length L will contribute a marker at each level less than L. However, it seems very hard to perfectly account for the number of markers contributed by a given set of prefixes for arbitrary expansion levels. The approximation also does produce good results.

If we denote the cost of covering lengths $1 \ldots j$ using levels $1 \ldots r$ by $T[j, r]$, we have:

$$T[j, r] = Entries[j] + min_{m \in \{r-1 \ldots j-1\}}\{T[m, r - 1] + ExpansionCost[m + 1, j]$$

$$T[j, 1] = Entries[j] + ExpansionCost[1, j]$$

There are $k*W$ elements in the array; computing each finding each element takes $O(W)$ time, since we take a minimum over at most W quantities. Thus the overall complexity of this algorithm is $O(k*W^2)$.

6.4 Performance of Expansion + Binary Search on Hash Tables

We present the results obtained by expanding the prefixes to have only three distinct levels. For each database, our program searches for a semiperfect hash function that gives at most six collisions; the time to find one such function is also given in Table XI. The 16-bit table is implemented as a full array, while the other tables 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 XI.

When compared to the worst-case figure for 5 levels for the unexpanded form of MaeEast, we have roughly twice the memory requirement (3220KB versus 1600) but a factor of 1.66 reduction in the worst-case number of memory accesses (3 versus 5). This appears to be an attractive trade-off. The only fly in the ointment is that searching for a good semiperfect hash

[9]Although this might seem as if it should be j instead of $j - 1$, the statement is correct because level numbers start from 0.

Table XI. Binary Search on Hash Tables after Expanding to Levels Given by the Dynamic Program and Time Taken for the Dynamic Program. Random-search time is found by computing the average measured search time to a million randomly chosen IP addresses.

Database	Time to Find Hash (sec.)	Memory (KB) (16, 24, 32)	Time for Dynamic Program (msec.)	Memory (KB) 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

function (in order to guarantee worst-case times) can take as long as 13 minutes! Once again, we could make the argument that such a search could be relegated to a management station to be done once every hour because the rate at which managers add prefixes to the Internet (a manual process) is on the order of hours, if not days. Note that if we calculate a semiperfect hash function for the set S of all prefixes that a router sees in a day, deletions will cause no problems because the same semiperfect hash function will work for any subset of S.

7. IP LOOKUPS IN HARDWARE

While this article has focused on software comparisons, it is also important to consider how IP lookup schemes could be implemented in hardware. Many vendors today are using hardware IP lookup engines because of the cheaper cost and the higher speeds offered by such engines. Thus in this section, we first present a hardware model and the relevant metrics for hardware. We then briefly consider the schemes described in this article from a hardware implementation perspective.

7.1 Hardware Model

We assume a cheap forwarding engine (say US $50 assuming high volumes) operating at a clock rate of 2–10 nsec. We assume the chip can place its data structure in SRAM (with say 10 nsec. access times) and/or DRAM (60–100 nsec. access times). In memory comparisons, it is important to observe that SRAM is several times more expensive than DRAM. For example, following current prices, SRAM costs six times as much as DRAM per byte. Finally, some forwarding engines may have a few megabits of on-chip memory that is extremely fast (say 5 nsec. access times).

Also, modern memory technologies like SDRAM and RAMBUS[10] provide a way to hide the long access times of DRAMs. They do so by providing a single chip with multiple DRAM banks to which accesses can be interleaved. Thus an access for bank $B1$ can be started while the results of a previous access to bank $B0$ are still forthcoming. Such technology lends

[10]http://www.rambus.com/

itself naturally to pipelining, with the memory for each pipeline stage in a separate bank. At the same time, the forwarding engine only needs a single set of address pins to drive the multiple memory banks of a single RAMBUS or SDRAM.

Besides the search time measured in memory accesses, two other crucial measures for a hardware implementation are *worst-case* memory usage and update times. While our article has described the memory usage of trie algorithms on "typical databases," it would be better to have a hardware implementation that can specify the worst-case number of prefixes it can support, given a bounded amount of fast memory. For single chip forwarding engines, fast update times are also crucial. Besides the problem of backbone instability alluded to earlier, a chip that can do database updates entirely in hardware is much more attractive to customers. For example, consider an IP lookup chip that can do wire speed forwarding at OC-48 rates (one lookup every 166 nsec.), can handle 50,000 arbitrary prefixes, and can do an update every msec. Such a chip would be extremely valuable today compared to CAM solutions that can only handle around 8000 prefixes.

7.2 Comparing Schemes in Hardware

None of the schemes described in this article have good worst-case memory usage. However, it is easy to add worst-case memory bounds for multibit tries using a technique called path compression, first described in Wilkinson et al. [1998]. In this technique, trie nodes with only one pointer are removed and substituted with an equivalent bit string. Path compression differs slightly from the more standard technique for compressing one-way branches using a skip count, is described in Knuth [1973] and used in Sklower [1991]. An important advantage of path compression over skip count compression is that path compression does not require backtracking [Sklower 1991; Stevens 1994] during search.

Using path compression, the number of trie nodes used to store n prefixes can be shown to be no more than $2n$. Path compression [Wilkinson et al. 1998] can be applied to all the multibit trie schemes described in this article to provide worst-case memory bounds.

Given this modification, the expanded trie schemes described in our article are best applicable to hardware implementations that use external DRAM. This is because DRAM is cheap and inexpensive, and the increased memory needs of expanded tries are not a significant issue. However, since even our fastest trie schemes take 3–4 memory accesses, implementations (e.g., routers that support OC-48 links) that require lookup times faster than say 300 nsec. will require pipelining.

A simple observation is that any search tree, whether a trie or a binary search tree, can be pipelined by splitting its levels into pipeline stages. The idea is that each level of the tree corresponds to a pipeline stage. The address is fed to the first stage which contains the root. The comparison at the first stage stage tells which pointer to follow (e.g., to node $N1$ or $N2$).

The address is then passed to the second stage *along* with the pointer to say node $N2$. The second stage contains both nodes $N1$ and $N2$. Since the passed pointer says $N2$, the address is compared with $N2$ and the resulting pointer is passed to the third stage, and so on. Pipelining allows a speed of one lookup per memory access at the cost of possibly increased complexity. As we have seen, memory technologies such as SDRAM and RAMBUS lend themselves naturally to pipelining, by providing a single set of pins that drive multiple memory banks. In summary, *all* the schemes described in this article can be pipelined to provide one lookup per memory access.

As link speeds continue to increase (e.g., terabit forwarding), DRAM memory access times will become a bottleneck. At such speeds, the entire IP lookup database will need to be stored in SRAM or on-chip memory. In such environments, where memory is again limited, schemes that compress trie nodes such as Degermark et al. [1997] and Eatherton et al. [1999] will do better than multibit expanded tries.

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 [Degermark et al. 1997] we have a version (leaf-pushed, variable stride) that has faster lookup times (196 nsec. versus 409 nsec.) but more memory (500KB versus 160KB). 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 Inserts are supposed to be rare [Degermark et al. 1997; Waldvogel et al. 1997]. However, because BGP updates can add and withdraw prefixes at a rapid rate, prefix insertion *is* important. Our schemes can also be tuned to a wide range of software and hardware environments. Our schemes have been implemented by at least three companies and will, hopefully, appear in forthcoming products.

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. Our optimization ideas can also be applied to the Lulea scheme to decrease memory further.

With expansion, binary search on levels [Waldvogel et al. 1997] can be also made fairly competitive. Its main disadvantage is the time and memory required to find semiperfect 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 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 in our software implementations) because the measurements on real databases indicated only small improvements.

We believe with Waldvogel et al. [1997] and Degermark et al. [1997] that IP lookup technology can be implemented in software, at least up to gigabit speeds. We also believe that fast lookup algorithms make the arguments for Tag and IP switching less compelling. 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.

ACKNOWLEDGMENTS

We would like to thank Sanjay Cherian (Ascend), Zubin Dittia (Washington University), Will Eatherton (Washington University), Chaitanya Kodeboyina (Microsoft), Geoff Ladwig (Bay Labs), Craig Partridge (BBN), Marcel Waldvogel (ETH) and the anonymous referees for invaluable feedback.

REFERENCES

BRADNER, S. 1997. Next generation routers overview. In *Proceedings of Networld* (Interop '97).

CHANDRANMENON, G. AND VARGHESE, G. 1996. Trading packet headers for packet processing. *IEEE/ACM Trans. Netw. 4*, 2 (Apr.), 141–152.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

DEERING, S. AND HINDEN, R. 1995. Internet protocol, version 6 (IPv6) specification. RFC 1883. Internic.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *Proceedings of the ACM Conference on Communications, Architecture, and Protocols* (SIGCOMM '97). ACM Press, New York, NY.

DEMERS, A., KESHAV, S., AND SHENKER, S. 1989. Analysis and simulation of a fair queuing algorithm. In *Proceedings of the ACM Conference on Communications Architectures and Protocols* (SIGCOMM '89, Austin, TX, Sept. 19–22), L. H. Landweber, Ed. ACM Press, New York, NY.

EATHERTON, W., DITTIA, Z., AND VARGHESE, G. 1999. Full tree bit map: Hardware/software IP lookup algorithms with incremental updates. In *Proceedings of the IEEE Conference on Computer Communications* (Infocom '99). IEEE Press, Piscataway, NJ.

GRAY, M. 1996. Internet growth summary. Available via http://www.mit.edu/people/mkgray/net/internet-growth-summary.html.

INTEL. 1997. VTune Performance Enhancement Environment. Intel Corp., Santa Clara, CA.

INTEL. 1998. Pentium Pro and Pentium II processors and related products. Intel Corp., Santa Clara, CA.

INTERNET-II. 1997. Big fast routers: Multi-megapacket forwarding engines for Internet II. In *Proceedings of Networld* (Interop '97).

KNUTH, D. E. 1973. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

LABOVITZ, C., MALAN, G., AND JAHANIAN, F. 1997. Internet routing instability. In *Proceedings of the ACM Conference on Communications, Architecture, and Protocols* (SIGCOMM '97). ACM Press, New York, NY.

LAMPSON, B., SRINIVASAN, V., AND VARGHESE, G. 1998. IP lookups using multi-way and multicolumn search. In *Proceedings of the IEEE Conference on Computer Communications* (Infocom '98, San Francisco, CA, Mar. 29–Apr. 2). IEEE Press, Piscataway, NJ.

MCAULEY, A., TSUCHIYA, P., AND WILSON, D. 1995. Fast multilevel heirarchical routing table using content-addressable memory. U.S. Patent serial number 034444. United States Patent Office, Washington, D.C.

MCKEOWN, N., IZZARD, M., MEKKITTIKUL, A., ELLERSICK, B., AND HOROWITZ, M. 1997. The tiny tera: A packet switch core. *IEEE Micro* (Jan.).

MERIT NETWORK. 1997. Ipma statistics. Merit Network, Inc., Ann Arbor, MI. Available via http://nic.merit.edu/ipma. (Snapshot on 12 September 97).

NEWMAN, P., MINSHALL, G., AND HUSTON, L. 1997. IP switching and gigabit routers. *IEEE Commun. Mag.* (Jan.).

NILSSON, S. AND KARLSSON, G. 1998. Fast address look-up for Internet routers. In *Proceedings of the IEEE Conference on Broadband Communications* (Stuttgart, Germany, Apr.). IEEE Press, Piscataway, NJ.

PARULKAR, G., SCHMIDT, D., AND TURNER, J. 1995. IP/ATM: A strategy for integrating IP with ATM. In *Proceedings of the ACM Conference on Computers Communications Technology* (SIGCOMM '95). ACM Press, New York, NY.

PERLMAN, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Co., Inc., Redwood City, CA.

RECHTER, Y. AND LI, T. 1993. An architecture for IP address allocation with CIDR. RFC 1518.

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

REKHTER, Y., DAVIE, B., KATZ, D., ROSEN, E., SWALLOW, G., AND FARINACCI, D. 1996. Tag switching architecture overview. Internet draft.

SHREEDHAR, M. AND VARGHESE, G. 1996. Efficient fair queuing using deficit round robin. *IEEE/ACM Trans. Netw. 4*, 3 (June), 376–385.

SITES, R. L. AND WITEK, R. T. 1995. *Alpha AXP Architecture Reference Manual*. 2nd ed. Digital Press, Newton, MA.

SKLOWER, K. 1991. A tree-based routing table for Berkeley Unix. In *Proceedings of the 1991 Winter USENIX Conference*. USENIX Assoc., Berkeley, CA.

STEVENS, W. R. 1994. *TCP/IP Illustrated*. Vol. 1, *The Protocols*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

TAMMEL, A. 1997. How to survive as an ISP. In *Proceedings of Networld* (Interop '97).

TURNER, J. 1997. Design of a gigabit ATM switch. In *Proceedings of the IEEE Conference on Computer Communications* (Infocom '97). IEEE Press, Piscataway, NJ.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing lookups. In *Proceedings of the ACM Conference on Communications, Architecture, and Protocols* (SIGCOMM '97). ACM Press, New York, NY.

WILKINSON, H., VARGHESE, G., AND POOLE, N. 1998. Compressed prefix matching database searching. United States Patent Office, Washington, D.C. U.S. Patent 5781772.

Received: May 1998; revised: November 1998; accepted: November 1998