Efficient IP Table Lookup via Adaptive Stratified Trees with Selective Reconstructions

MARCO PELLEGRINI and GIORDANO FUSCO
Istituto di Informatica e Telematica

IP address lookup is a critical operation for high-bandwidth routers in packet-switching networks, such as Internet. The lookup is a nontrivial operation, since it requires searching for the longest prefix, among those stored in a (large) given table, matching the IP address. Ever increasing routing table size, traffic volume, and links speed demand new and more efficient algorithms. Moreover, the imminent move to IPv6 128-bit addresses will soon require a rethinking of previous technical choices. This article describes a the new data structure for solving the IP table lookup problem christened the adaptive stratified tree (AST). The proposed solution is based on casting the problem in geometric terms and on repeated application of efficient local geometric optimization routines. Experiments with this approach have shown that in terms of storage, query time, and update time the AST is at a par with state of the art algorithms based on data compression or string manipulations (and often it is better on some of the measured quantities).

Categories and Subject Descriptors: C.2.6 [**Internetworking**]: Routers

General Terms: Algorithms

Additional Key Words and Phrases: IP table lookup; data structures

ACM Reference Format:

Pellegrini, M. and Fusco, G. 2007. Efficient IP table lookup via adaptive stratified trees with selective reconstructions. ACM J. Exp. Algor. 12, Article 1.4 (2007), 28 pages DOI 10.1145/1227161.1278376 http://doi.acm.org 10.1145/1227161.1278376

1. INTRODUCTION

1.1 Motivation for the Problem

The Internet is surely one of the great scientific, technological, and social successes of the last decade and an ever-growing range of services rely on the

Preliminary version in *Proceedings of the 12th European Symposium on Algorithms* (ESA). LNCS 3221, 2004.

Authors' address: Instituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche via Moruzzi 1, 56124 Pisa, Italy; email: {marco.pellegrini, giordano.fusco}@iit.cnr.it.

efficiency of the underlying switching infrastructure. Thus, improvements in the throughput of Internet routers are likely to have a major impact. The IP address-lookup mechanism is a critical component of an Internet packet switch (see McKeown [1999] for an overview). Briefly, a router within the network holds a lookup table with n entries, where each entry specifies a *prefix* (the maximum length w of a prefix is 32 bits in the IPv4 protocol, 128 bits in the soon-to-be-deployed IPv6 protocol) and a next-hop exit line. When a packet comes to the router, the destination address in the header of the packet is read, the longest prefix in the table matching the destination is sought, and the packet is sent to the corresponding next-hop exit line. How to solve this problem so to be able to handle millions of packets per second has been the topic of a large number of research papers in the last 10–15 years. A steady increase in the size of the lookup tables and relentless demand of traffic performance put pressure on the research community to come up with faster schemes.

1.2 Our Contribution

In the IP protocol, the address lookup operation is equivalent to the searching for the *longest prefix match (lpm)* of a fixed-length string among a stored set of prefixes. It is well known that the longest prefix match problem can also be mapped into the problem of finding the shortest segment on a line containing a query point. This is called the *geometric (or locus) view* as opposed to the previous *string view*. The geometric approach has been proposed in Warkhede et al. [2004] and Lampson et al. [1999]; it has been extended in Feldmann and Muthukrishnan [2000] and in several recent papers [Thorup 2003; Kaplan et al. 2003]. It has been used in Buchsbaum et al. [2003] to formally prove the equivalence among several different problems. Important asymptotic worst-case bounds have been found following the geometric point of view, but, currently, the algorithms for IP lookup with the best empirical performance, including the one in Buchsbaum et al. [2003], are based on the string view and use either optimized tries or data-compression techniques, or both. The question that we raise is whether the geometric view is valuable for the IP lookup problem only as a theoretical tool or it is also a viable tool from a practical point of view. We notice that for the multidimensional packet filtering problem, the multidimensional FIS tree [Feldmann and Muthukrishnan 2000], which follows the geometric view, has already established itself as a benchmark, not only for the asymptotic bounds, but also for empirical performance.[1] Since our objective is practical, we chose for the moment not to rely on the sophisticated techniques that have lead to the recent improvements in worst-case bounds (e.g., those in Feldmann and Muthukrishnan [2000], Thorup [2003], and Kaplan et al. [2003]).[2] Instead, we ask ourselves whether simpler search trees built with the help of local optimization routines could lead to the stated goal.

[1]See Geraci et al. [2005] for a an alternative efficient method for multidimensional packet filtering based on geometric concepts.

[2]In Section 8, we comment on the relative practical performance of the FIS tree and AST for IP lookup.

The experiments with the AST method lead us to give a strong positive hint on both accounts: (1) it is, indeed, possible to gain state of the art performance in IP lookup using the geometric view and (2) local adaptive optimization is a key ingredient for attaining this goal.

Since this is an experimental paper and the evaluation of the experimental data is critical, we stress the experimental methodology beforehand in Section 2. In Section 3, we discuss previous work with the intent of highlighting the key ideas and key differences with the AST, which is introduced in Section 4.1. In Section 4.2, we introduce some notation and preliminary storage reduction techniques, while in Section 5 the local optimization routines are described. Experiments and results for storage and query time are presented in Section 6, while dynamic operations are discussed in Section 7. In Sections 8 and 9, we extend the review of related work to more recent results and compare them with the AST.

2. METHODOLOGY

Performance is established by means of experiments, which are described in detail in Section 6. The experimental methodology we adopted follows quite closely the standards adopted in the papers describing the methods we compare to. Given the economic relevance of Internet technologies, such state of the art methods have been patented or are in the process of being patented, and, to the best of our knowledge, the corresponding codes (either as source or as executables) are not in the public domain. We refrained from reimplementing these methods, using as guidance the published algorithms, since we are aware that at this level of performance (few dozens of nanoseconds) overlooking seemingly simple implementation details could result in an grossly unfair comparisons. Given the above-mentioned difficulties, we settled for an admittedly weaker, but still instructive, comparison based on the fact that those papers present, either an explicit formula for mapping their performance onto different machines,[3] or such formulas could be derived almost mechanically from the description in words. A second issue was deciding the data sets to be used. Papers from the late 1990s used table snapshots usually dating from the mid 1990s, when a table of 40,000 entries was considered large. Although recovering such old data is probably feasible and using them certainly interesting, the evolution of Internet in recent years has been so rapid that the outcome of comparisons based on outdated test cases is certainly open to the criticism of not being relevant for the present (and the future) Internet. Therefore, the comparisons shown in Tables IV and V, (see later), where we use data sets of comparable size and the mapping onto a common architecture given by the formulas, should be considered just in a broad qualitative sense. The main general conclusion we feel we can safely draw is that the geometric view is as useful as insight drawn from string manipulation and data compression.

[3]Naturally such formulas cannot capture all the nuisances of modern CPU architectures: they are based on the choice of a broad family (e.g., RISC versus CISC), leaving as parameters the clock frequency, as well as size and access time of cache and RAM memory.

Table I. List of Input Lookup Tables[a]

	Routing Table	Date	Entries	Total Points	Active Points	Phantom Points	Duplicate Points
1	Paix	05/30/2001	17,766	35,532	17,807	9,605	8,120
2	AADS	05/30/2001	32,505	65,010	32,642	17,071	15,297
3	Funet	10/30/1997	41,328	82,656	24,895	35,391	22,370
4	Pac Bell	05/30/2001	45,184	90,368	34,334	33,750	22,284
5	Mae West	05/30/2001	71,319	142,638	53,618	50,217	38,803
6	Telstra	03/31/2001	104,096	208,192	64,656	79,429	64,107
7	Oregon	03/31/2001	118,190	236,380	45,299	116,897	74,184
8	AT&T US	07/10/2003	121,613	243,226	102,861	62,910	77,455
9	Telus	07/10/2003	126,282	252,564	85,564	86,738	80,262
10	AT&T East Canada	07/10/2003	127,157	254,314	78,625	93,795	81,894
11	AT&T West Canada	07/10/2003	127,248	254,496	78,708	93,824	81,964
12	Oregon	07/10/2003	142,475	284,950	108,780	84,165	92,005

[a]Input geometric statistics for active, duplicates, and phantom points.

Table II. Slim-AST Data Structure Performance[a]

	Routing Table	Entries	Memory Slim-AST	Memory Fat-AST	Worst Query (clock ticks)	Worst Class (Lev, Pts, Base)
1	Paix	17,766	374,314	440,436	35	2,2, 8
2	AADS	32,505	460,516	589,762	35	2,2, 8
3	Funet	41,328	453,034	679,568	30	2,1, 8
4	Pac Bell	45,184	464,063	716,548	35	2,2, 8
5	Mae West	71,319	610,587	980,182	35	2,2, 8
6	Telstra	104,096	780,452	1,314,990	35	2,2,16
7	Oregon	118,190	595,919	1,452,832	35	2,3, 8
8	AT&T US	121,613	955,702	1,488,806	30	2,1, 8
9	Telus (*)	126,282	878,179	1,610,716	40	2,4, 8
10	AT&T East Canada	127,157	878,155	1,542,682	40	2,4, 8
11	AT&T West Canada	127,248	878,491	1,543,525	40	2,4, 8
12	Oregon (*)	142,475	940,810	1,797,609	35	2,2,16

[a]Tests on a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 cache. L2 delay is D = 15 ns; L1 latency m = 4 ns. The parameters used are: $B = 16$, $C = 10^{-7}$, and $D = 61$ for all tables except (*) where $C = 1$.

We remark that the extensive testing with 12 lookup tables of size ranging from 17,000 to 142,000 (reported in Tables I, and II) confirms the reliability and robustness of AST.

3. PREVIOUS WORK

The literature on the IP-lookup problem and its higher dimensional extension[4] is rather rich and a complete survey is beyond the scope of this article. Here we mention briefly the principles underlying some of the methods. At a high level we can distinguish three main approaches: (i) data structures and software

[4]Sometimes the IP lookup problem is treated as the one-dimensional (1-D) special case of the more general *packet-filtering* problem. Similarly, in geometric terms, the predecessor problem in one-dimension can be seen as a special case of the (hyper)-rectangle stabbing problem in dimension $d \geq 2$. When the packet filtering is solved via dimensionality reduction, the IP lookup can be thought of as the base case of the recursion. Sometimes the terminologies of the IP lookup and the packet-filtering problems get mixed up, which may generate confusion.

searching techniques, (ii) design of specialized hardware aiming at exploiting machine-level parallelism and (iii) avoiding the lookup process altogether by exploiting additional header information. Here we comment only on approach (i).

The classical data structure to solve the longest matching prefix problem is the binary trie [Knuth 1973]. Patricia tries [Morrison 1968] take advantage of common subsequences to compress certain paths in a binary trie, thus saving both in search time and storage. The first IP-Lookup algorithm in Sklower [1991] is based on the patricia trie idea. Nilsson and Karlsson [1999] add the concept of *level compression* by increasing the out-degree of each node of the trie as long, as at least a large fraction (called the filling factor) of the subtrees of a node are nonempty.

Waldvogel et al. [1997] organize the prefixes in groups of same length and each group is stored in a hash table for fast membership testing. The search is essentially a binary search by prefix length over the hash tables (a quite similar technique is found in Willard [1983]). The number of hash tables can be reduced by padding some of the prefixes [Srinivasan and Varghese 1999]. Good hashing strategies are studied in Broder and Mitzenmacher [2001a].

In Srinivasan and Varghese [1999], a method called *variable-stride trie* is described. Here a trie is built, but the number of bits (*stride*) used for the branching at a node is chosen separately for each trie node. The choices that optimize the memory occupation are found via dynamic programming. In Buchsbaum et al. [2003], the trie is augmented by compressing all leaves and internal nodes via shortest common superstring compression [Blum et al. 1994]. It should be noted that while in Srinivasan and Varghese [1999] the model states a 1 MB cap on storage and tries to attain the best worst-case time within the cap (which is also our model), the emphasis in Buchsbaum et al. [2003] is different: the target is a good average time for random traffic with no limit on storage; thus, up to 3 MB are used to store a table of 118 K entries.

Degemark et al. [1997] use data compression techniques to compactly store parts of the prefix tree representing the set of prefixes. At present, this technique achieves, in practice, the lowest use of storage. Crescenzi et al. [1999] instead start from a full table representation of the lookup function then applying a data-compression technique that reduces the storage to acceptable levels, in practice, while requiring only three memory accesses to answer a query.

The multiway range tree [Warkhede et al. 2004] and the multiway search tree [Lampson et al. 1999] are the two early geometric-based methods. The first one is akin to a classical segment tree with a root-to-leaf visit. The second one is akin to a fixed-stride k-ways trie with a very large branching factor at the root. The branching degree is uniform and is decided beforehand; no optimization, local or global, is done.

Ergun et al. [Sharp et al. 2001] and [Ergun et al. 2001] use the fast reconfiguration capabilities of skip lists [Pugh 1990] to adapt on-line the search data structure to the modifications of traffic patterns.

A number of recent papers have addressed the IP lookup problem with the aim of improving the worst-case asymptotic performance. Dynamic 1-D fat inverted segment trees [Feldmann and Muthukrishnan 2000] is an elaboration of traditional segment trees and, for a small constant l, achieves query

time $O(\log w + l)$ uses storage $O(n + ln^{1+1/l})$ and has amortized update cost $O(ln^{1/l} \log n)$ over a sequence of $O(n^{1/l})$ updates. Thorup [2003] improves the dynamic FIS tree: the storage becomes linear in n and any sequence of dynamic operations is supported efficiently (a detailed discussion of the FIS tree approach is postponed to Section 8). Kaplan et al. [2003] give new algorithms for maintaining sets of intervals on a line that can be used for solving the IP lookup problem efficiently in the pointer model.

4. THE AST

4.1 AST in a Nutshell

The AST construction algorithm is described in Section 5. Here we summarize the main new ideas. The best way to explain the gist of the algorithm is to visualize it geometrically. We first map the lookup problem into a predecessor search problem (see Section 4.2 for details). The equivalent input for the predecessor problem is a set of labeled points on the real line. We want to split this line into a grid of *equal size buckets* and then proceed recursively and separately on the points contained in each grid bucket. A uniform grid is completely specified by giving an anchor point a and the step s of the grid. During the query, finding the bucket containing the query point p is done easily in time $O(1)$ by evaluating the expression $\lfloor (p-a)/s \rfloor$, which gives the offset from the special bucket containing the anchor. We will take care of choosing s as a power of 2 so to reduce integer division to a right shift. If we choose the step s too short, we might end up with too many empty buckets, which implies a waste of storage. Thus, we choose s as follows: choose the smallest s for which the ratio of empty to occupied buckets is no more than a user-defined constant threshold. On the other hand, shifting the grid (i.e., moving its anchor) can have dramatic effects on the number of empty buckets, occupied buckets, and the maximum number of keys in a bucket. Thus, the search for the optimal step size includes an inner optimization loop on the choice of the anchor to minimize the maximum bucket occupancy. The construction of the (locally) optimal step and anchor can be done efficiently in time close to linear, up to polylog terms (see Section 5). The algorithm works in two main phases. In the first phase, we build a tree that aims at using small space at the expense of exhibiting a few long paths. In the second phase, the long paths are shortened by compressing them, increasing the storage used.

During the construction, a basic subtask is finding a grid optimizing a local objective function. This approach has to be contrasted with several techniques in literature where a global optimality criterion is sought usually via much more expensive dynamic programming techniques, such as in Cheung and McCanne [1999], Gupta et al. [2000], Srinivasan and Varghese [1999], and Buchsbaum et al. [2003]. Ioannidis et al. [2005] recently proposed a global reconstruction method for tries that rely on a reduction to the knapsack problem and, therefore, suggests the use of knapsack approximation methods. In Ioannidis et al. [2005], the target is to reduce the average query time, based on traffic statistics, not the worst-case query time.

A simple theoretical analysis of the asymptotic worst-case query time/storage for the AST gives asymptotic worst-case bounds $O(n)$ for storage and $O(\log n)$ for query time at the end of the first phase. We are not able to quantify asymptotically the improvements introduced in the second phase, since the computation is essentially data adaptive. However the query-time bound does not explain the observed performance and we leave it as an open problem to produce a more refined asymptotic analysis.

4.2 The Problem

The *longest prefix match* problem has an input that consists of a set T of 0/1 strings of length up to an integer w and a function mapping each prefix to a label indicating the next-hop. For any destination address q, we return the label associated with the longest among the strings in T that are prefixes of q. We can turn this problem into the *shortest stabbing segment* problem as follows. Each entry in a routing table has the form

$$netbase/netmask \rightarrow nexthop$$

where netbase is an 32-bits IP address, netmask denote the number of bits used for prefix, and next-hop is the destination rout of the packets whose destination address matches the prefix specified in this entry. From each table entry, we build a pair of values denoting the begin and the end of interval. The rule is:

$$\begin{aligned} begin &= netbase \wedge mask \\ end &= (netbase \vee (\neg mask)) + 1 \end{aligned}$$

where $\wedge \vee \neg$ are the bitwise operators AND, OR, and NOT. Moreover, we associate to *begin* and *end* values a label

$$\begin{aligned} label[begin] &= nexthop \\ label[end] &= nexthop\ of\ previous\ matching\ prefix \end{aligned}$$

Thus, we can turn the set T of labeled prefixes into a set S of labeled points (see Figure 1). For any query point q, its predecessor point in S is labeled correctly with the next-hop of the longest matching prefix for q. For a prefix $p \in T$, denote with $M(p)$ its corresponding segment (seen as a contiguous subset of the set $[0, \ldots, 2^w - 1]$).

4.2.1 *Semi-Closed Intervals.* In real tables it happens quite often that the segments are adjacent, meaning that for two prefixes p' and p'', $\max M(p')+1 = \min M(p'')$. In this case we can considerably reduce the number of required distinct points by considering our intervals closed on the left but open on the right. That is, the pair $[a, b[$ with $b \geq a$ represents the interval $[a, b-1]$. In such a way, the closed intervals $[a, b]$, $[b+1, c]$ are represented as $[a, b+1]$,

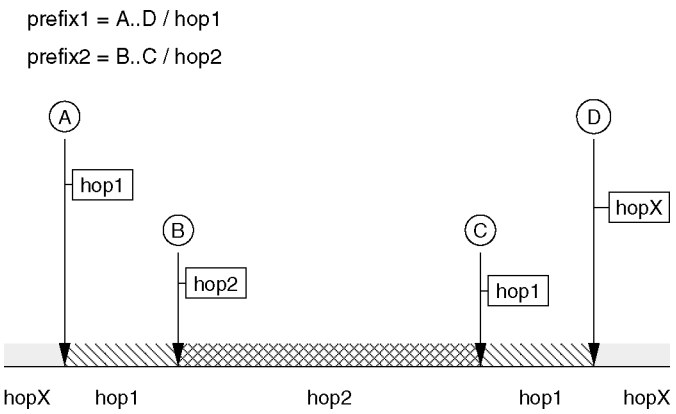

Fig. 1. Intervals and next-hops.

$[b+1, c+1]$. Thus, only three distinct points need to be recorded[5] instead of four. The left end-point of the second interval is called a *duplicate* point.

4.2.2 *Phantom Points.* If two consecutive points in S have the same label, then the right-most such point can be safely deleted from S for the purpose of solving for static predecessor queries. As seen in Table I, both duplicates and phantom represent a sizable portion of the points in actual tables.

5. CONSTRUCTION OF THE AST

5.1 Main Ideas

5.1.1 *Generic Recursive Bucketing Tree.* Here we describe the AST data structure by iteratively refining a general framework with specific choices. For simplicity, we describe in detail the data structure that solves the predecessor problem in a set of points on the line. Afterward, we will comment on the small modifications needed to solve the dynamic IP lookup problem.

We start by describing a *generic recursive bucketing tree* $T(U, S)$. Consider, initially, the set $U = [0, .., 2^w - 1]$ of all possible points and the input set $S \subset U$ of points. We recursively build a tree by levels. Each node x of the tree has an associated connected subset of the universe $U_x \subseteq U$ (i.e., an interval) and a point data set $S_x \subset S$, which is $S_x = S \cap U_x$. Initially at the root of the tree r, we have $U_r = U$ and $S_r = S$. Let x be a node of the tree and let $k(x)$ be the number of children on x. By $y[1], \ldots, y[k(x)]$, we denote the children of x. We partition U_x into a number $k(x)$ of disjoint intervals, $U_1, \ldots, U_{k(x)}$, each associated to a child of x, then, as a consequence, we associate to each child of x the set of points $S_{y[i]} = U_i \cap S_x$. The recursive construction stops for a node y

[5]Note that in case of singleton sets, such as $[a, a]$, we represent them as $[a, a-1]$, thus increasing the number of points. In actual tables, this increase happens rarely. If we use a machine with w-bit registers to store prefixes up to w bits, we might have to represent the number 2^w, which require $w+1$ bits. This, however, is not a problem, since for the purpose of a query, we do not need to store the points beyond the query range and for the purpose of updates, we just treat the last number as a special case.

when $|S_y| \leq c$ for a constant c. At each leaf y, we associate to y the set S_y and, moreover, the point in $S \setminus S_y$ that is the unique predecessor of any point in U_y.

5.1.2 *How to Query a Generic Recursive Bucketing Tree.* When a query point q is given, the generic recursive bucketing tree $T(U, S)$ is used to solve the predecessor problem, as follows. Trace a path from root to leaf as follows: when node x has been reached, visit the child y of x such that $q \in U_y$. When we reach a leaf l at the end of this path, the answer is computed by direct comparison of q with the $|S_l| + 1$ points stored at leaf l.

5.1.3 *From a Generic Recursive Bucketing Tree to an Adaptive Stratified Tree.* The only step that need to be specified is: how do we compute the partition of U_x into a number $k(x)$ of disjoint intervals? We use as split points the points of the infinite uniform grid $G(a, s) = \{a + ks : k \in N\}$ of anchor a and step s that fall in U_x (that is, $U_x \cap G(a, s)$). Note that by using these split points all the induced intervals (buckets) are of equal length (that is s), except maybe the first and the last interval. Note also that finding the index of the bucket containing a query point q amounts to just computing $\lfloor \frac{q-a}{s} \rfloor$, that is, an $O(1)$ operation, independent of the number of children of a node. Thus, we just need to specify rules for choosing the anchor a and the step s for a node x. To do so, we introduce two objective functions that we would like to simultaneously minimize. Since a multiobjective optimization is problematic, we will instead minimize one function subject to an upper bound on the other.

Let $E(a, s)$ be the number of buckets of the above construction containing no point in S_x and $F(a, s)$ the number of buckets containing at least one point of S_x. The ratio is $R(a, s) = \frac{E(a,s)}{F(a,s)}$. Define also the minimum of such ratio over all possible choices of the anchor a: $R_{min}(s) = \min_a R(a, s)$. The ratio $R_{min}(s)$ is the first objective function we are interested in. Such function has been defined with the purpose of controlling the number of empty buckets, since a proliferation of empty buckets is bad for memory consumption. Full buckets instead contain input points. Therefore, their number can be charged to the size of the input.

Let $G'(a, s)$ be the set of intervals partitioning U_x induced by $G(a, s) \cap U_x$. Consider $g(a, s)$, the maximum number of points of S in any such interval, formally:

$$g(a, s) = \max_{I \in G'(a,s)} |S \cap I|$$

Consider then the choice of anchor a that minimizes this maximum number, thus leading to the following function: $g_{min}(s) = \min_a g(a, s)$. The function $g_{min}(s)$ is the second objective function we are interested in. Such function has been introduced with the purpose of controlling the maximum depth of the search tree.

5.1.4 *Phase I: Saving Space.* In order to control memory consumption, we adopt the following action recursively and separately at each node: find smallest value s' such that $R_{min}(s') \leq C$, for a predefined constant threshold C. Then take an anchor a that realizes the value of $g_{min}(s')$, i.e., gives us the min–max occupancy (this shifting does not change the number of nodes at the level).

5.1.5 *Phase II: Selective Reconstruction.* The tree built in Phase I uses linear space (see proof below), but can have some root-to-leaf path of high length. The purpose of the second phase is to reduce the length of the long paths, without paying too much in storage. In order to do so, we visit the longest paths and we decrease the value s' used in the split so to increase the number of children and decrease the min–max bucket occupancy. This operation trades off storage against tree depth.

5.2 Further Details of the Construction

5.2.1 *How to Compute Efficiently $R_{min}(s)$ and $g_{min}(s)$.* In previous paragraphs, we have introduced two functionals that we want to minimize and have explained how they are used in the AST construction. Here we will show that such functions can be computed efficiently. Preprocessing time for data structures devoted to IP lookup is not one of the most important parameters. However, since occasionally one has to (partially) rebuild the data structures to maintain its properties, it is important that such operations are performed as efficiently as possible.

First of all we notice that by shifting the anchor by s to the right, the grid remains unchanged: $G(a, s) = G(a + s, s)$. Consider the continuous movement of a grid: $f_G(\alpha) = G(a + \alpha s, s)$ for $\alpha \in [0, 1]$. Call an *event* a value α_i for which $S \cap f_G(\alpha_i) \neq \emptyset$.

LEMMA 5.1. *The number of events is, at most, n.*

PROOF. Take a single fixed interval I of $G(a, s)$. We have an event when the moving left extreme meets a point in I. This can happen only once for each point in I. Therefore, overall, there are, at most, n events. □

Since we study bucket occupancy, extending the shift for $\alpha > 1$ is useless, since every distribution of points in the bucket has already been considered, given the periodic nature of the grid. Consider point $p \in S$ and the bucket I_p containing p. Point p produces an event when $\alpha_p = (p - \mathit{left}(I_p))/s$, that is, when the shift is equal to the distance from the left extreme of the interval containing p. Thus, we can generate the order of events by constructing a min-priority queue $Q(S, \alpha_p)$ on the set S using as priority the value of α_p. We can extract iteratively the minimum for the queue and update the counters $c(I)$ for the shifting interval I. Note that for our counters, an event consists in decreasing the counter for a bucket and increasing it for the neighbor bucket. Moreover, we keep the current maximum of the counters. To do so, we keep a second max-priority queue $Q(I, c(I))$. When a counter increases, we apply, the operation increase key; when it decreases, we apply the operation decrease key. Finally, we record changes in the root of the priority queue, recording the minimum value found during the lifetime of the algorithm. This value is the

$$g_{min}(s) = \min_{\alpha \in [0,1]} \max_{I \in G(\alpha s, s)} |S \cap I|$$

that is, we find the shift that for a given step s minimizes the maximal occupancy. Using standard priority queue implementations, the whole algorithm for computing $g_{min}(s)$ takes time $O(n \log n)$. Similarly, we can compute $R_{min}(s)$

within the same time bound. Next we show monotonicity properties of the functions $R_{min}(s)$ and $g_{min}(s)$ that will allow us to use binary search schemes in finding the value s'.

LEMMA 5.2. *For any two step values s and t, if $t = 2s$, then we have $g_{min}(t) \geq g_{min}(s)$.*

PROOF. Consider the grid $G_{min}(t)$ that attains min–max occupancy $K = g_{min}(t)$. Thus, every bucket in $G_{min}(t)$ has, at most, K elements. Now we consider the grid $G(s)$ that splits exactly in two every bucket in $G_{min}(t)$. In this grid $G(s)$, the maximum occupancy is, at most, K, so the value $g(s)$ that minimizes the maximum occupancy for a translate of $G(s)$ cannot attain a larger value than K, i.e., $g_{min}(s) \leq K = g_{min}(t)$. □

LEMMA 5.3. *For any two-step values s and t, if $t = 2s$, then we have $R_{min}(t) \leq R_{min}(s)$.*

PROOF. Take the grid $G_{min}(s)$, the grid of step s minimizing the ratio $R_{min}(s) = E_s/F_s$. Now make the grid $G(t)$ by pairing adjacent buckets. Call N_s, F_s, E_s the number of buckets, full buckets, and empty buckets in $G_{min}(s)$. Call N_t, F_t, E_t the number of buckets, full buckets, and empty buckets in $G(t)$. We have the relations: $N_s = N$, $N_t = N/2$, $F_s/2 \leq F_t \leq F_s$, and $(E_s - F_s)/2 \leq E_t \leq E_s/2$. Now we express $R_t = E_t/F_t$ as a function of N and F_t.

$$R_t = E_t/F_t = \left(\frac{1}{2}\right) N/F_t - 1$$

This is an arc of hyperbola (in the variable F_t) having maximum value for abscissa $F_t = F_s/2$. The value of the maximum is $E_s/F_s = R_{min}(s)$. Thus, we have shown that $R(t) \leq R_{min}(s)$. Naturally, also, $R_{min}(t) \leq R(t)$, so we have proved $R_{min}(t) \leq R_{min}(s)$. □

Thus the minimum ratio is monotonic increasing as grids get finer and finer and we can use binary search to find the largest step value satisfying the criterion of Phase I.

5.2.2 *Details of the Selective Reconstruction.* In current technology, access to the RAM memory is about ten times slower than one to L2 cache. Thus, it is fundamental to fit all the relevant data into the L2 cache. It is reasonable to suppose that the target machine has, say, 1 or 2 Mb of L2 cache and that it is completely devoted to the IP table lookup. With this hypothesis, the size of the routing table does not make a difference up to the L2 cache size. When the routing table is built, we can perform a selective reconstruction of the longest paths to flatten the AST in order to reduce the worst query time maintaining the total memory consumption below the L2 cache size. The following steps are repeated while the size of the routing table is below the L2 cache size and there are other reconstructions to perform:

1. Create a max-priority queue of internal nodes based on the maximum cost of their children (see below).

Table III. L2-Normalized Costs for Points of 8 Bits[a]

	Level				
Position	1	2	3	4	5
Empty	−3	20	42	66	116
1	12	30	65	93	121
2	12	35	67	94	125
3	16	35	64	100	128
4	22	40	69	101	129

[a]Measures in clock ticks. The first negative entry results from the inherent approximation in the measurement and normalization of the raw data, as an outlayer is not used in the construction of the interpolation formula.

2. Visit the AST and consider only internal nodes having only leaves as children; determine the cost of these nodes and insert them in the max-priority queue.
3. Extract from the queue the nodes with the same maximum cost and flatten them in the following way: if the maximum number of points in a child full leaf is greater than 1, split the *step* until the maximum number of points in a bucket becomes smaller than the current maximum; otherwise go to the parent and split its step until the maximum number of points in a bucket becomes smaller than maximum number of points a full leaf can contain (in this way, the level below is eliminated).

A few exceptions to the above rules: the root is not rebuilt, and, for simplicity, when a level is eliminated other reconstructions of the same cost are performed only after recomputing the priority queue.

5.2.3 *The Cost of a Node.* Each node of the AST has associated a cost, which represents the time of the slowest query among all the queries that visit that node. The cost of a full leaf is the time to reach the last point stored inside it (i.e., the time to visit all the points from the median to the farthest extreme). The cost of an internal node is the maximum of the costs of all its children. See Table III later for the costs in our experimental setting.

The leaves can be subdivided into classes (*Lev*, *Pts*, *Base*) depending on: their level *Lev* in the AST, the number of points *Pts* from the median to the farthest extreme, and the number of bits *Base* used to stored the points in the leaf (8, 16, or 32).[6]

To guide the selective reconstruction a clock-ticks table is build once for each CPU architecture. For all the possible classes of leaves, the table stores the corresponding query time. The query time of a particular class is measured in the following way: a simple AST containing nodes of the chosen class is built, several queries reaching nodes of that class are performed, the minimum time is taken, and it is normalized to L2 cache. The rationale is that a machine dedicated to the IP table lookup is capable of performing a query in the minimum time measured in the test machine, because the processor is not disturbed by

[6]As we search down the tree, the points can be represented with fewer bits. We take advantage of this fact to further decrease the storage consumption by roughly 10%.

other running processes. The measurements have been done using Pentium-specific low-level primitives to attain higher precision; nothing forbids to use operating system primitives for better portability.

5.2.4 *Modifications for Solving the IP-Lookup Problem.* In order to solve the static IP-lookup method, we just have to store labeled active points instead of all points. For the dynamic IP-lookup problem, we need to store all labeled points (active, phantom, and duplicates). Notice, however, that in all computations involving point/bucket occupancy only the active points will be considered, since duplicated and phantom points are relevant only for the update procedure. The root of the tree is treated in a special way, since we use 2^B buckets. The maximum number of active points in a leaf is set to a parameter D. Finally, the bound on the ratio $R_{min}(s)$ is a constant C. The three parameters B, C, D are specified by the user of the code at build time.

5.2.5 *Asymptotic Bounds for Storage and Worst-Case Query Time.* Since at each node, the number of empty buckets is bounded by the number of full buckets multiplied by a constant factor, it sufficed to analyze the number of full buckets. Since each full leaf has at least one point and each point is in, at most, one set S_l for some leaf l, the number of full leaves is $O(n)$. Since by positioning the anchor at the median point of the set S_x for a node x we obtain at least two full children from any internal node, we have that the brunching factor of the tree restricted to the full nodes is always at least two. Therefore, the number of internal full nodes is linear in the number of full leaves. Thus the overall memory consumption is $O(n)$. Since, at each level, a bucket receives, at most, one-half of the points of its father, the depth is, at most, $O(\log n)$.

6. EXPERIMENTS WITH AST

6.1 Fat-AST versus Slim-AST

We consider two variants of the basic AST data structure. The *Fat-AST* is built on all input points (active, duplicates, and phantom) in the data set. Thus, it can correctly answer any query and directly supports updates . However, all occupancy counts are done with respect to the active points only. The *Slim-AST* is built only on the active points in a data set. Thus, it can correctly answer any query but is unable to directly support updates . Updates on the Slim-AST can be done either (1) by a partial execution of the building algorithm, starting from the input data and portions of the data structure, or (2) by requiring a version of the Fat-AST for the input data on a different (usually slower and less expensive) memory bank and copying the portions of the tree that are changed back onto the Slim-Tree. For fairness, we compare the Slim-AST against data structures that do not support fast updates like Degermark et al. [1997] and Crescenzi et al. [1999] (see later Table IV), and we compare the Fat-AST against data structures also supporting fast updates like the variable-stride tries in Srinivasan and Varghese [1999] (see later Table V).

Table IV. Comparisons of Static IP Methods for a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache[a]

Ref	Method	Input size	Formula	Time (ns)	Memory (KB)
Degermark et al. [1997]	Lulea	38141	53T + 12D	254	160
Crescenzi et al. [1999]	Exp./Con.	44024	3T + 3D	49	1057
This paper	Slim-AST	45184	m + 3D	49	464

[a]L2 delay is D = 15 ns; L1 latency m = 4 ns.

Table V. Comparisons of Dynamic IP Methods for a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache

Ref	Method	Input size	Formula	Time (ns)	Memory (KB)
Srinivasan and Varghese [1999]	Variable stride	38816	26T + 3D	81	655
This paper	Fat-AST	45184	m + 3D	49	716

[a]L2 delay is D = 15 ns; L1 latency m = 4 ns.

6.2 Worst-Case Query Time

The testing of the (empirical) worst-case query time is done statically on the structure of the tree by finding out the longest (most expensive) path from root to leaf. The worst-case query time gives valuable information that is relevant for, but independent from, any actual query data stream.

Table I gives a synthetic view of the IP-lookup tables used for testing the AST data structure. For each table, we report a progressive number, the name of the organization or router holding the table, the date when the table was downloaded, and the number of prefixes in the tables. Each prefix generates two points. We next classify the points into active, phantom, and duplicate points, giving the count for each category.

Table II gives the relevant measure of storage, worst-case memory access, and worst-case time (in clock ticks) for the Slim-AST and the Fat-AST.

6.3 Normalization in L2

To test the AST data structure, we adopt the methodology in Degermark et al. [1997]. For completeness we describe the methodology. We visit each node of AST twice so to force data for the second invocation in L1 cache (This is known as the *hot cache* model). We measure number of clock ticks by reading the appropriate internal CPU registers relative to the second invocation. As in Degermark et al. [1997], we exclude from measurement the function invocation overhead and consider the search completed when we retrieve the index of the next-hop. We also record the number of L1 memory accesses of each query. This is important because, knowing the L1 latency, we decompose the measured time in CPU operation cost and memory access cost. Afterward, we scale the memory access cost to the cost of accessing data in L2 cache, unless by locality we can argue that the data must be in L1 cache, also on the target machine. We call this measurement model the *L2-normalized model*. Since the root is accessed in each query, after unrolling the search loop, we store step, anchor, and pointer to the first child at the root in registers so the access to the root is separately treated from accessing any other level of the AST. The results of the

L2-normalized measures are shown in Table III for 8-bit keys. Access times for 16 and 32 bit keys are almost identical.

6.4 Target Architecture

Here we derive a formula we will use to scale our results to other architectures. The target architecture is made of one processor and two caches: one L1 cache and one L2 cache. $P \leftrightarrow CL1 \leftrightarrow CL2$. We suppose every memory access is an L1 miss, except when we access consecutive cells. We need to determine by fitting the experimental measurements the constants a, b, and c in the formula for accessing a point at level k and distance s in the list stored at a leaf:

$$a + k(M + m + b) + (M + m) + (s - 1)(c + m)$$

where M is the time to access L2 cache and m the time to access $T1$ cache.

Since, in Table II, we have only results for classes from $(2, 1)$ to $(2, 4)$, in Table III we need accuracy of interpolation only in this range. Via easy calculations, the final interpolation formula for any $(2, s)$ class is: $3(M + m) + (s - 1)m$. We denote with $D = M + m$ the L2 latency.

6.5 Derivation of Formulas in Tables IV and V

From Crescenzi et al. [1999, p. 68]: "According to these measurements, on a Pentium II processor, each lookup requires $3 \cdot clk + 3 \cdot M_D$ nanoseconds, where clk denotes the clock cycle time and M_D is the memory access delay." Thus, in our notation, this time formula becomes $3T + 3D$, assuming the data structure fits completely in L2 memory.

From Srinivasan and Varghese [1999, pp. 27–29], several variants of the fixed and variable stride methods are compared on the 38,816 prefixes Mae East table for different sizes of the L2 memory (512 KB and 1 MB). Taking the data for the 1 MB L2 cache size in Srinivasan and Varghese [1999, Table X, p. 29], the authors comment: "Now we consider the same database using a machine with 1 MB of L2 cache (Table X). Here we see 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." Since we are interested in comparing, at this stage, only time versus storage, we do not consider the worst case update time as a drawback and we concentrate on the method attaining the smallest query time while keeping the overall memory consumption below 1 MB. The entry that satisfies these requirements is attained for the algorithm "leaf-pushed variable stride" with $k = 3$ levels. In Srinivasan and Varghese [1999, table VIII, p. 27] this algorithm is characterized by the formula $(8k + 2) \cdot clk + k \cdot M_D$, which, for $k = 3$, in our notation gives the formula $26T + 3D$.

From Degermark et al. [1997, p. 11]: "The difference in access time between the primary and the secondary caches is 20 ns (4 cycles). The lookup time on the Pentium Pro when two levels need to be examined is then at worst $69 + 8 \cdot 4 = 101$ cycles." Table II in Degermark et al. [1997, p. 10] reports 10 ns = 2 cycles L1 latency and 30 ns = 6 cycles for the Pentium Pro 200 MHz machine used

for the measurements. Since visiting each level of the data structure requires 4 memory accesses, two levels are accessed, and the latency of L2 access is 6 cycles, we can derive $x = 101 - 6 \cdot 8 = 53$ cycles for nonmemory access operations and $8D$ memory accesses, yielding formula $53T + 8D$. However, from Table I Degermark et al. [1997, p. 10] we see that a third level is often needed to handle all the queries in sufficiently large tables. Thus, a worst-case estimate on the 32,732 prefixes Mae East is given by adding 4 more memory access, yielding $53T + 12D$. In any case, the qualitative conclusion we draw from the comparison do not change when either formula is used.

6.6 Discussion of AST Performance

In Tables IV and V, we compare the Slim-AST and the Fat-AST methods with three other methods for which analogous formulas have been derived by the respective authors (explicitly or implicitly), mapped onto a common architecture. The method in Degermark et al. [1997] is, by far, the most storage efficient, using roughly four bytes per entry; it is based on tree compression techniques, which make updates difficult. The query time is much higher than that of the AST. The expansion/compression method [Crescenzi et al. 1999] is very fast (requires only three memory access) using a fair amount of memory and requiring extensive reconstructions of the data structure at each update.[7] The Slim-AST is as fast, but requires much less memory and thus, has better scaling properties.

The method of Srinivasan and Varghese [1999] balances query time, storage, and update time well. Its storage performance, for tables of roughly 40,000 entries, is not far form the one of the Fat-AST, however, the Fat-AST is remarkably faster on a similar data set. Recent improvements of the scheme in Srinivasan and Varghese [1999] are described and compared to the AST in Section 8.

7. DYNAMIC OPERATIONS

7.1 Relative Importance of Performance Measures

In this paper, we designed the AST experiments so to give highest importance to the worst case query time, second in importance comes storage consumption, and finally comes update times. In this section, we describe the dynamic operations on a Fat-AST. We describe some measurements on update time and make a comparison with those of a variable-stride tries in Srinivasan and Varghese [1999]. Other dynamic IP lookup results are mentioned in Section 8 and compared to the AST.

7.2 Description of the Dynamic Operations

We give a brief sketch of the dynamic operations. Inserting a new prefix p involves two phases: (1) inserting the end points of the corresponding segment $M(p)$; (2) updating the next-hop index for all the leaves that are covered by the segment $M(p)$. While phase (1) involves only two searches and local restructuring, phase (2) involves a DFS of the portion of the AST tree dominated by the

[7]However, recently the average query time, as well as the storage consumption, have been somewhat improved (R. Grossi, personal communication).

Table VI. Fat-AST Data Structure Performance of Insertion[a]

		100%			99%			95%		
		Accesses		Time	Accesses		Time	Accesses		Time
	Routing Table	L2	L1	(μs)	L2	L1	μs	L2	L1	(μs)
1	Paix	107	734	4.54	97	668	4.13	43	288	1.80
2	AADS	588	6	8.84	69	466	2.90	33	216	1.36
3	Funet	4285	156	64.90	79	538	3.34	59	398	2.48
4	PacBell	1296	154	20.06	71	480	2.98	35	222	1.41
5	MaeWest	1941	84	29.45	103	706	4.37	51	346	2.15
6	Telstra	1867	548	30.20	81	546	3.40	53	356	2.22
7	Oregon	4381	1598	72.11	457	3184	19.59	87	588	3.66
8	AT&T US	1675	76	25.43	67	460	2.85	43	282	1.77
9	Telus	1686	10	25.33	73	488	3.05	45	294	1.85
10	AT&T East Canada	1686	10	25.33	81	552	3.42	49	322	2.02
11	AT&T West Canada	1686	10	25.33	81	552	3.42	49	322	2.02
12	Oregon	2309	76	34.94	465	3244	19.95	67	446	2.79

[a]Number of memory access in L2 and L1 cache for the 100, 99, and 95 percentiles. Time estimate in the test machine. Tests on a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache. L2 delay is D = 15 ns; L1 latency m = 4 ns.

inserted/deleted segment. Such a DFS is potentially an expensive operation, requiring, in the worst case, the visit of the whole tree. However, we can trim the search by observing that it is not necessary to visit subtrees with a span included into a segment that is included in $M(p)$, since these nodes do not change the next-hop. There are n prefixes and $O(n)$ leaves and the span of each leaf is intersected without being enclosed by, at most, $2D + 1$ segments. Therefore, only $O(1)$ leaves need to be visited, on average, and eventually updated.[8]

Deleting prefix p from the AST involves phases similar to insertion. However, we need to determine first the segment in the AST including $M(p)$ in order to perform the leaf relabeling. This is done by performing, at most, w searches, one for each prefix of p, and selecting the longest such prefix present in the AST. In practice, fewer than w searches are necessary, in most cases. Although it is possible to check and enforce the other AST invariants at each update, we have noticed that the query performance remains stable over long sequences of random updates. Therefore, enforcing the AST invariant is best left to an off-line periodic restructuring process.

7.3 Experimental Data

We perform experiments on updating the AST using as update data stream the full content of each table. For deletions, this is natural, since we can delete only prefixes present in the table. For insertions, we argue that entries in the table have been inserted in the past; thus, it is reasonable to use them as representative also for the future updates.[9]

Tables VI and VII show upper estimates on the number of memory accesses and the percentile time (μsec) for completing 95, 99, and 100% of the updates

[8]This rough argument can be made precise in an amortized sense by using the model of (N, Δ)-sequences of updates of Mulmuley [1993].

[9]In public repositories of Internet data sets, it seems to be difficult to find both table snapshots and traffic streams for the same tables, which would form an even better test case.

Table VII. Fat-AST Data Structure Performance of Deletions[a]

		100%			99%			95%		
		Accesses		Time	Accesses		Time	Accesses		Time
	Routing Table	L2	L1	(μs)	L2	L1	(μs)	L2	L1	(μs)
1	Paix	102	22	1.62	57	390	2.42	31	196	1.25
2	AADS	588	6	8.84	57	390	2.42	29	192	1.20
3	Funet	4285	156	64.90	79	538	3.34	59	398	2.48
4	PacBell	1296	154	20.06	71	480	2.99	33	218	1.37
5	MaeWest	1941	84	29.45	89	610	3.78	49	320	2.02
6	Telstra	1948	28	29.33	81	546	3.40	50	316	2.01
7	Oregon	4381	1598	72.11	457	3184	19.59	87	588	3.66
8	AT&T US	1675	76	25.43	67	460	2.85	43	282	1.77
9	Telus	1686	10	25.33	73	488	3.05	45	292	1.84
10	AT&T East Canada	1686	10	25.33	81	552	3.42	49	322	2.02
11	AT&T West Canada	1686	10	25.33	81	552	3.42	49	322	2.02
12	Oregon	2309	76	34.94	465	3244	19.95	67	446	2.79

[a]Number of memory access in L2 and L1 cache for the 100, 99, and 95 percentiles. Tests on a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache. L2 delay is D = 15 ns; L1 latency m = 4 ns.

(insertions and deletions) for the Fat-AST data structures built for the lookup tests. Note that experiments for deletions and insertion give very similar counts.

During the update, we count the number of memory cells accessed distinguishing access in L2 and those in L1, assuming the data structure is stored completely in L2 cache. Since, in the counting, we adopt a conservative policy, i.e., we upper bound the number of cells accessed at each node visited during the update, the final count represents an upper bound on the actual number of memory access.

Tables VIII and IX show the insert and delete times (upper estimates) for a typical large table (West AT&T Canada), with results separated by prefix length. For the same prefix length in both tables, we obtain very similar counts. Note that the worst update time is attained by a few quite short prefixes (length from 8 to 12).

7.4 Comparison of Update Time with Srinivasan and Varghese [1999]

In Srinivasan and Varghese [1999] an estimate of 2500 μs for worst-case updates is given, counting the dominant cost of the memory accesses. The cost is incurred when a node in the trie with a large out degree (2^{17}) needs to be rebuilt and an access cost of 20 ns per entry is assumed. Since 24-bit prefixes are very frequent (about 50% of the entries), the variable-stride tries are forced so that updating for such length is faster, requiring roughly 3 μs. Although our machine is faster than that used in Srinivasan and Varghese [1999], the L2 access time is quite similar. Therefore, the timings in Srinivasan and Varghese [1999] can be compared (although only on a qualitative basis) with results in Table VI. We believe that the 2500 μs worst-case estimate for Srinivasan and Varghese [1999] is overly pessimistic: in our experiments (deletion and insertion of every entry of every table), except for the Oregon data sets, up to 95% of the updates are completed in about 4 μs. Thus, it is very close to the average update time in Srinivasan and Varghese [1999]. Globally the update performance of AST and variable-stride tries comparable.

Table VIII. Fat-AST Data Structure Performance of Insertion by Prefix Length for Data Set West AT&T Canada[a]

Prefix Length	# of Prefixes	100% Accesses L2	100% Accesses L1	100% Time (μs)	99% Accesses L2	99% Accesses L1	99% Time (μs)	95% Accesses L2	95% Accesses L1	95% Time (μs)
8	17	1686	14	25.35	100	0	1.50	100	0	1.50
9	5	49	0	0.73	39	2	0.59	39	2	0.59
10	8	1675	80	25.45	41	30	0.73	41	30	0.73
11	13	1273	258	20.13	1223	60	18.59	1223	60	18.59
12	54	841	122	13.10	744	140	11.72	479	142	7.75
13	97	494	260	8.45	473	16	7.16	245	192	4.44
14	259	251	382	5.29	204	286	4.20	124	174	2.56
15	473	176	764	5.70	112	8	1.71	64	222	1.85
16	7459	185	52	2.98	46	292	1.86	20	98	0.69
17	1601	92	612	3.83	49	334	2.07	30	184	1.19
18	2933	512	14	7.74	54	340	2.17	31	194	1.24
19	8477	316	38	4.89	53	356	2.22	31	198	1.26
20	8424	175	70	2.90	53	360	2.23	33	208	1.33
21	5972	110	740	4.61	63	432	2.67	41	276	1.72
22	8979	157	1080	6.67	65	436	2.72	49	326	2.04
23	10,088	157	1080	6.67	79	534	3.32	55	368	2.30
24	67,865	157	1080	6.67	87	588	3.66	57	386	2.40
25	151	157	1080	6.67	157	1080	6.67	157	1080	6.67
26	169	47	314	1.96	37	248	1.55	33	214	1.35
27	696	157	1080	6.67	157	1080	6.67	157	1080	6.67
28	691	48	306	1.94	47	314	1.96	33	214	1.35
29	2490	157	1080	6.67	157	1080	6.67	37	248	1.55
30	324	48	294	1.90	39	250	1.58	37	238	1.51
31	0									
32	3	39	250	1.58	29	180	1.16	29	180	1.16

[a]Number of memory access in L2 and L1 cache for the 100, 99, and 95 percentiles. Time estimate in the test machine. Tests on a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache. L2 delay is D = 15 ns; L1 latency m = 4 ns.

Finally, we remark that updates in Degermark et al. [1997] and Crescenzi et al. [1999] are handled by rebuilding the data structure from scratch, thus requiring time of the order of several milliseconds for every update.

8. MORE RELATED WORK AND COMPARISONS WITH AST

8.1 Comparing the AST and the FIS Tree

Let S be the set of end points of the input intervals and q a query point. We want to report the shortest segment in the input set containing q. If $q \notin S$ then the answer for q is the same as the answer for the predecessor of q in S, which is a point of S. If $q \in S$ we just have to recognize this fact. Thus, a dynamic dictionary is needed and an auxiliary data structure is used for a restricted form of the problem in which the query set coincides with S. Usually a data structure for the predecessor problem is able to act as a dictionary. Thus, we can solve the stabbing problem by using two data structures:

1. A Data structure for *dynamic predecessor problem* (in Thorup [2003] also called *dynamic searching problem*).

Table IX. Fat-AST Data Structure Performance of Deletion by Prefix Length for Data Set West AT&T Canada[a]

Prefix Length	# of Prefixes	100% Accesses L2	100% Accesses L1	100% Time (μs)	99% Accesses L2	99% Accesses L1	99% Time (μs)	95% Accesses L2	95% Accesses L1	95% Time (μs)
8	17	1686	10	25.33	96	0	1.44	96	0	1.44
9	5	45	0	0.68	37	0	0.56	37	0	0.56
10	8	1675	76	25.43	41	26	0.72	41	26	0.72
11	13	1273	254	20.11	1223	56	18.57	1223	56	18.57
12	54	841	118	13.09	742	138	11.68	477	140	7.71
13	97	494	256	8.43	429	156	7.06	243	190	4.41
14	259	251	378	5.28	204	282	4.19	122	32	1.96
15	473	176	760	5.68	104	108	1.99	62	94	1.31
16	7459	160	1080	6.72	43	272	1.73	20	100	0.70
17	1601	90	610	3.79	47	310	1.95	30	172	1.14
18	2933	512	10	7.72	54	336	2.15	31	208	1.30
19	8477	316	34	4.88	53	360	2.23	31	198	1.26
20	8424	175	66	2.89	53	360	2.23	33	220	1.38
21	5972	110	736	4.59	63	428	2.66	41	272	1.70
22	8979	157	1076	6.66	67	460	2.85	49	322	2.02
23	10088	157	1076	6.66	79	530	3.31	55	368	2.30
24	67865	157	1076	6.66	85	586	3.62	55	366	2.29
25	151	157	1076	6.66	157	1076	6.66	157	1076	6.66
26	169	47	310	1.95	37	244	1.53	33	210	1.33
27	696	157	1076	6.66	157	1076	6.66	157	1076	6.66
28	691	48	302	1.93	47	310	1.95	33	210	1.33
29	2490	157	1076	6.66	157	1076	6.66	37	244	1.53
30	324	48	290	1.88	37	248	1.55	35	236	1.47
31	0									
32	3	37	248	1.55	27	178	1.12	27	178	1.12

[a]Number of memory access in L2 and L1 cache for the 100, 99, and 95 percentiles. Time estimate in the test machine. Tests on a 700 MHz Pentium III (T = 1.4 ns) with 1024 KB L2 Cache. L2 delay is D = 15 ns; L1 latency m = 4 ns.

2. A data structure for *dynamic list stabbing*, with the restriction that the set of possible queries coincide with the set of end points of the input segment.

The FIS tree (fat inverted segment tree) is an elaboration of the *segment tree* and is used to implement the data structure specified in (2). Since the FIS-tree has a tree structure and each end point of the input segments is stored at a leaf of the FIS tree, we can couple data structures (1) and (2) using an inverse mapping that maps the points returned by the data structure (1) to the corresponding leaves of the data structure (2). This mapping operation is essential, since the FIS tree can be visited efficiently at query time only when the visit follows a leaf-to-root path (for this reason the adjective "Inverted" is part of the acronym).

Data structure (1) can be implemented using a variety of solutions in literature. In Feldmann and Muthukrishnan [2000], it is suggested to use a dynamic van Emde Boas tree augmented with dynamic hashing so to keep the query time at $O(\log \log U)$ and reduce the storage to $O(n)$. A more practical solution also mentioned in Feldmann and Muthukrishnan [2000] is that of using B trees.

The coupling of the data structures (1) and (2) (e.g., van Emde Boas tree + FIS tree) is traversed as follows. First the data structure (1) is traversed in a root-to-leaf fashion so to identify the proper starting leaf in the data structure (2). Afterward the FIS tree is traversed from leaf-to-root and candidate answers are collected at the nodes on the path; among these the final answer is selected. If one were interested in a static solution, all this construction is useless and a data structure (1) with labeled points would be sufficient. Thus, the coupling of the data structures (1) and (2) is needed in order to have dynamic update procedures with provable asymptotic performance.

The AST is not a FIS tree. In particular, the AST is a tree that is traversed in a root-to-leaf fashion and finds the correct output as a label of the reached leaf. Thus, the AST sacrifices provable update performance bounds (and relies on empirically assessed measurements for this measure of quality), but gains in the fact that a single shallow tree has to be traversed. Even if a depth parameter $t = 2$ is chosen for the FIS tree, the twin data structure (1) when implemented as a van Emde Boas tree would visit $\log \log U$ nodes. Thus, for $U = 2^{32}$, an additional five nodes are visited, giving a total of seven visited nodes to answer any query. In our experiments with AST, we would solve IP lookup queries by visiting only two nodes of an AST tree and a total of three memory accesses. Since visiting a path in a tree would require at least a memory access per node, even without further experimental evidence, it is safe to conclude that any standard implementation of a FIS tree-based data structure for IP lookup would result in a slower query time than the AST (or other specialized methods, such as Srinivasan and Varghese [1999]).

8.2 Improved Construction of Fixed- and Variable-Stride Tries

Sahni and Kim [2001, 2002, 2003] in a series of papers propose new formulations of the dynamic programming underlying the construction of the fixed stride and variable stride tries in Srinivasan and Varghese [1999]. The emphasis is on improving the preprocessing time. Both in the fixed- and variable-stride cases, the user defines an input parameter k that is the depth of the tree. For a a given value of k, the searching algorithm and the searching performance is identical to that in Srinivasan and Varghese [1999]. Thus, the other quality measure that needs to be measured is the memory consumption. For $k = 3$ and variable-stride trie, for a table of 35 K prefixes, the reported storage is 677 KB [Sahni and Kim 2003, Table IV, p. 676], which is reasonably close to the storage bound reported in Srinivasan and Varghese [1999]. With the same data set, a value $k = 2$ would lower the search time, but push up the storage to 1.8 MB. Thus, we can conclude that the storage and search time tradeoffs are analogous to those in Srinivasan and Varghese [1999].

8.3 Data Structures Stressing Dynamic Operations

Sahni and Kim [2004] give a data structure that improves the update time with respect to the variable-stride tries in Srinivasan and Varghese [1999] and Sahni and Kim [2003], but has slower search time and larger storage requirements. In our setting, we compare data structures by giving higher priority to search

time (within reasonable storage), next in importance is storage and only the last criterion in update time. Thus, the AST and the result in Sahni and Kim [2004] are not comparable, because they imply a different ranking of relevant performance measures. Similarly, Lu and Sahni [2005] propose a variant of the B tree to efficiently handle updates. Experimental data reported cannot be compared directly with ours, since the raw timing reported in Lu and Sahni [2005] comprises accesses in RAM memory, while in our analysis we fit all data in L2-level cache. However, we can make use of a cache-miss analysis in Lu and Sahni [2005, p. 17] that estimates a worst-case number of cache misses for look up search at $0.5 \log_2 n$. This number would correspond to memory accesses if the data structure were completely in L2 cache. For $n = 32$ K prefixes this would imply a number of about seven memory accesses. This is far from the AST count of three for similar input sizes. This discrepancy is obviously as a result of the emphasis in improving dynamic operations at expenses of lookup efficiency. Lu and Sahni [2004b] explore the use of red-black trees and prioritysearch trees for solving the longest-prefix mach problem (as well as other related problems). Since these are trees with branching factor two, they put stress on low memory consumption and fast updates at the cost of having to visit paths of length $O(\log n)$ in order to solve IP-lookup queries. Sun et al. [2004] propose a general technique for exploiting independent sets in the input data on a multiprocessor parallel architecture to speed up update operations.

8.4 A Very Recent Data Structure

Sundström and Larzon [2005] give a data structure whose main feature is to be able to store large data sets requiring just four memory accesses to handle a lookup query. More precisely, a data set of 131,227 prefixes is stored in 950 KB and a search requires four memory access. The update operations on this data set have a worst-case cost of 752 memory accesses. The data sets 8, 9, 10, and 11 in our Table I get close to that used in Sundström and Larzon [2005] in terms of sheer size. We see that we can achieve for those data seta only three memory accesses. The Slim-AST also achieves similar storage bounds, but cannot handle updates directly. The Fat-AST would support dynamic operations, but uses roughly 50% more memory on such input data.

8.5 Partitioning Schemes

Lu et al. [2005] propose several schemes for an initial partition of the data set into disjoint subsets. The rationale is to be able to apply, for each subset, a more traditional data structure (e.g., variable-stride tries) whose parameters are optimized on the local input subset. This operation leads in experiments to improvement in storage and *average* search time. Note that this technique is "generic" and, as such, could also be applied directly to AST's.

8.6 Data Structures Stressing Storage Requirements

Elaborating on ideas in Degermark et al. [1997], Eatherton et al. [2004] propose the tree bitmap data structure that can be adapted to several memory

models. For the memory model adopted in this paper (L2 and L1 cache levels), they are able to remap the tables in Degermark et al. [1997] so that each tree node access requires only one memory access (while in Degermark et al. [1997]) four memory access per node are needed. However, the requirement of small memory occupancy results in a tree with higher depth. In experiments, a data set of 40,902 prefixes is stored in 312 KB, but requires seven memory accesses in worst case. Moreover, the technique in Eatherton et al. [2004] allows for fast average update time.

8.7 Different Matching Rules

Lu and Sahni [2004a] explore a slightly different problem matching rule. Here it is supposed each prefix that has an associated priority unrelated to the length and the problem is to return the highest priority prefix matching the destination address of a packet. The emphasis is on good worst-case asymptotic bounds on storage and update time. The number of memory accesses is equivalent to the height of the tree, which is $O(\log n)$. For $n = 32$ K this would imply about 15 memory accesses.

8.8 Hashing Based Schemes

The basic scheme of Waldvogel et al. [1997] (binary search on levels) is to expand certain prefixes so to obtain a small number of surviving prefix lengths and store all prefixes of the same length and markers for longer prefixes in a hash table. The IP-lookup problem is then solved by a binary search by prefix length. Kim and Sahni improve the basic approach in Waldvogel et al. [1997] by providing a new dynamic programming formulation, whose aim is to reduce storage consumption as well as reducing the number of prefix lengths. Broder and Mitzenmacher [2001b] give a technique for improving the performance of a single hash tables. The use of randomized schemes, based on hashing, can lead to good average case performance. However, this is paid for in the difficulty of controlling worst-case lookup time (which basically corresponds to the maximum bucket occupancy of one of the hash tables used). Moreover, dynamic updates becomes problematic, since the event of exceeding the maximum allowed bucket occupancy (for speed each bucket can occupy a single cache-line) requires the complete recomputation hash of functions and hash tables. In order to give a feeling of what can be accomplished with this technique, we report an experiment in Broder and Mitzenmacher [2001b] with a table of 38,816 prefixes showing that the scheme in Waldvogel et al. [1997] is able to solve queries with two memory accesses, using 4 MB, under the assumption that six prefixes can be stored in a single cache line. The scheme in Broder and Mitzenmacher [2001b] reduces the maximum bucket occupancy to four, uses one-half of the storage, but requires to visit two buckets in each hash table, thus, requiring, in total, four memory accesses. In contrast, the Fat-AST requires three memory accesses, thus is better than Broder and Mitzenmacher [2001b], for a much smaller memory occupancy.

8.9 Hw-Based Methods

There is a quite large portion of the recent research on the IP-lookup problem that uses specialized hw architectures different from the hw model adopted in this paper (e.g., Hasan and Vijaykumar [2005], Dharmapurikar et al. [2003], Taylor et al. [2003], Liu and Lea [2001], and Wuu and Pin [2001]). For this reason, we do not comment further on these results.

9. ON THE STATIC AND DYNAMIC PREDECESSOR PROBLEM

9.1 On the Predecessor Problem

Consider the following rather general formulation. Let $U = [0, 2^w - 1]$ be the set of positive integers expressible in binary notation with words of w bits[10] and let $S \subset U$ be a finite subset of cardinality n. The complement set $U \setminus S$ is composed of a collection I_S of disjoint open intervals. The universe U is thus partitioned into a collection of intervals I_S and points S. Let f be a function defined on U, with range values in a set H, such that f is *invariant within each interval* of I_S. The problem is to build, at preprocessing time, a data structure $D(U, S)$ so to efficiently retrieve $f(q)$ for a query $q \in U$ given on-line. For the static setting, the exact nature of the range set H and of the function f is immaterial and we concentrate on the task of searching, given a query $q \in U$, the node in the data structure $D(U, S)$ holding the associated value $f(q)$. Since we can associate the value for points in each interval I_S to the left end point of the interval, this query problem is equivalent to the *predecessor problem*: given U and S as above, for any $x \in U$ find $\max\{y \in S | y \leq x\}$.

Solutions to the (static) predecessor problem have performance bounds that are very sensitive to two conditions. The first is the relative size of n with respect to U; the second is the maximum amount of storage allowed. We may insist in having just linear storage $O(n)$, or storage polynomial in n or we may allow storage superpolynomial in n (i.e., polynomial in n *and* U). Note that, if n is not too small with respect to U, technically $n > U^{1/C}$ for some constant $C > 1$. Then, also, $U < n^C$ and, therefore, a polynomial in n and U is automatically a polynomial also in n only.

By using a k-ary tree with $k = U^\epsilon$, the resulting tree uses storage $O(nU^\epsilon)$ and the query time is constant. The height of the tree is $\frac{\log U}{\log U^\epsilon} = \frac{1}{\epsilon} = O(1)$, while the storage is bounded by $O(nk) = O(nU^\epsilon)$ (see e.g., Eppstein and Muthukrishnan [2001]). Naturally, with the observation above, when $n > U^{1/C}$, the storage is $O(n^{1+\epsilon})$ and the query remains $O(1)$.

The results we will survey try to reduce the storage requirement without increasing the query time too much. The methods we survey are, in general, dynamic, or can made so with some extra machinery. However, since we are interested in the query time versus storage performance, we will not dwell on such dynamic features.

Willard [1983, 1984] gives data structures (fast y-tries and fast q-tries) using storage $O(n)$ achieving query time $O(\min\{\log\log U, \sqrt{\log U}\})$. Stratified

[10] In typical current processors $w = 32$.

trees of van Emde Boas achieve query time $O(\log\log U)$ using storage $O(U)$ in the original paper [van Emde Boas et al. 1977]. Storage was reduced afterward to $O(nU^{\epsilon})$ in Johnson [1982] and to $O(n)$ in van Emde Boas [1977] when $n > |U|/\log^{c}|U|$. Willard [2000] lowers the lower limit for attaining linear storage to $n > |U|/(\log U)^{c\cdot\log\log|U|}$. Beame and Fich [1999] have shown that the predecessor problem among n input data from an integer universe U can be found using $O(n)$ storage and query time

$$O\left(\min\left\{\frac{(\log\log U)(\log\log n)}{\log\log\log U},\ \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$$

and, even more interestingly, they show that with storage limited to polynomial this is essentially the best that can be done for a certain range of relative sizes of n and U. They obtain the following static lower bounds: for every n such that $(\log U)^{\omega(1)} < n < U^{o(1)}$, using a number polynomial in n of memory cells of roughly $2^{(\log U)^{c}}$ bits each for $c < 1$, a query must use in the worst case $\Omega(\frac{\log\log U}{\log\log\log U})$ operations, while with cells of $O(\log U)^{C}$ bits each, $\Omega(\sqrt{\log n/\log\log n})$ operations are needed.

There are several results not as strong as those of Beame and Fich, but worth recalling. Raman [1995] gives a data structure using storage $O(n)$ and performing queries in time $O(\min\{\log\log U, 1+\log n/\log\log U\})$. A randomized method of Andersson [1995] uses storage $O(n)$ and has expected query time $O(\sqrt{\log n})$. Subsequently, in Andersson [1996], still using linear storage, the method has been made deterministic and the query time improved to

$$O\left(\min\left\{\sqrt{\log n},\ \log\log U\log\log n,\ \log\log n+\frac{\log n}{\log\log U}\right\}\right)$$

The problem of the optimality of the known bounds on the query time for the predecessor search problem is studied in Patrascu and Thorup [2006].

Knuth [1973] surveys several bucketing schemes (tries, Patricia trees, and digital search tree), which are all variants of the basic k-ary tree approach. All these schemes produce search trees of height $\log_k n$, while the expected size of the tree is $O(nk/\ln k)$. A variant discussed in Knuth [1973] uses an hybrid approach, when a bucket holds less than s data items, a standard binary search tree is used. The expected height is reduced to $(1+1/s)\log_k n+\log s$, while the storage is $O(nk/s\ln k+n)$.

9.2 Previous Results with Distribution-Dependent Guarantees

N-trees defined by Ehrlich [1981] capture, in a formal way, the probably older idea of recursive bucketing, where the number of buckets depends on the number of data items in each bucket. Results in Ehrlich [1981] show that the expected construction time is $O(n)$, when the data are drawn uniformly at random from the universe. Later, Tamminen [1983] showed that for any data set drawn from a distribution with a bounded and Lebesgue integrable probability density function, the height of the N-tree is bounded by a small constant. This result is interesting since it introduces distribution-dependent guarantees

beyond the standard uniform distribution. Willard [1985] gives an $O(\log\log n)$ expected time algorithm to search on data drawn from a smooth distribution.

9.3 Comparison of AST and Predecessor Search Data Structures

The AST does not have provable worst-case bounds that are competitive with the above-mentioned structures for predecessor search. However, experiments in Pellegrini and Vecchiocattivi [2001] have shown that the AST trees have storage and depth tradeoffs (measured in number of nodes of the tree and in the longest path) that are significantly superior to k, n, and van Emde Boas trees.

10. CONCLUSIONS AND OPEN PROBLEMS

Preliminary experiments with AST give encouraging results when compared with published results for state of the art methods. However, fine tuning of the AST requires properly choosing some user-defined parameters and we do not have, at the moment, general rules for this choice. How to find a theoretical underpinning to explain the AST good empirical performance is still an open issue for further research.

ACKNOWLEDGMENTS

We thank Giogio Vecchiocattivi who contributed to testing an early version of the AST data structure [Pellegrini et al. 2002], and the participants of the mini-workshop on routing (held in Pisa in March 2003) for many interesting discussions.

REFERENCES

ANDERSSON, A. 1995. Sublogarithmic searching without multiplications. In *36th Annual Symposium on Foundations of Computer Science: October 23–25, 1995, Milwaukee, Wisconsin*, IEEE, Ed. IEEE Computer Society Press, Los Alamitos, CA. 655–663.

ANDERSSON, A. 1996. Faster deterministic sorting and searching in linear space. In *37th Annual Symposium on Foundations of Computer Science*. IEEE, Burlington, Vermont. 135–141.

BEAME, P. AND FICH, F. E. 1999. Optimal bounds for the predecessor problem. In *STOC: The ACM Symposium on Theory of Computing*. ACM Press, New York. 295–304.

BLUM, A., JIANG, T., LI, M., TROMP, J., AND YANNAKAKIS, M. 1994. Linear approximation of shortest superstrings. *Journal of the ACM 41*, 4 (July), 630–647.

BRODER, A. Z. AND MITZENMACHER, M. 2001a. Using multiple hash functions to improve IP lookups. In *INFOCOM*. 1454–1463.

BRODER, A. Z. AND MITZENMACHER, M. 2001b. Using multiple hash functions to improve IP lookups. In *IEEE Computer and Communications Societies (INFOCOM)*. Vol. 3. 1454–1463.

BUCHSBAUM, A. L., FOWLER, G. S., KRISHNAMURTHY, B., VO, K.-P., AND WANG, J. 2003. Fast prefix matching of bounded strings. In *Proceedings of Alenex 2003*.

CHEUNG, G. AND MCCANNE, S. 1999. Optimal routing table design for IP address lookups under memory constraints. In *INFOCOM (3)*. 1437–1444.

CRESCENZI, P., DARDINI, L., AND GROSSI, R. 1999. IP address lookup made fast and simple. In *European Symposium on Algorithms*. 65–76.

DEGERMARK, M., BRODNIK, A., CARLSSON, S., AND PINK, S. 1997. Small forwarding tables for fast routing lookups. In *SIGCOMM*. 3–14.

DHARMAPURIKAR, S., KRISHNAMURTHY, P., AND TAYLOR, D. E. 2003. Longest prefix matching using bloom filters. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. ACM Press, New York. 201–212.

EATHERTON, W., VARGHESE, G., AND DITTIA, Z. 2004. Tree bitmap: Hardware/software IP lookups with incremental updates. *Computer Communication Review 34*, 2, 97–122.

EHRLICH, G. 1981. Searching and sorting real numbers. *Journal of Algorithms 2*, 1 (Mar.), 1–12.

EPPSTEIN, D. AND MUTHUKRISHNAN, S. 2001. Internet packet fileter management and rectangle geometry. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*. ACM Press, New York. 827–835.

ERGUN, F., SAHINALP, S. C., SHARP, J., AND SINHA, R. K. 2001. Biased skip lists for highly skewed access patterns. In *ALENEX: International Workshop on Algorithm Engineering and Experimentation, LNCS 2153*. 216–229.

FELDMANN, A. AND MUTHUKRISHNAN, S. 2000. Tradeoffs for packet classification. In *Proceedings of INFOCOM*, Vol. 3. IEEE, Los Alamitos. 1193–1202.

GERACI, F., PELLEGRINI, M., PISATI, P., AND RIZZO, L. 2005. Packet classification via improved space decomposition techniques. In *Proceedings of INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies* (March 13–17). Vol. 1. 304–312.

GUPTA, P., PRABHAKAR, B., AND BOYD, S. P. 2000. Near optimal routing lookups with bounded worst case performance. In *INFOCOM (3)*. 1184–1192.

HASAN, J. AND VIJAYKUMAR, T. N. 2005. Dynamic pipelining: Making IP-lookup truly scalable. *ACM SIGCOMM Computer Communication Review 35,* 4, 205–216.

IOANNIDIS, I., GRAMA, A., AND ATALLAH, M. J. 2005. Adaptive data structures for IP lookups. *ACM Journal of Experimental Algorithms 10*.

JOHNSON, D. B. 1982. A priority queue in which initialization and queue operations take $O(\log(\log(D))$ time. *Mathematical Systems Theory 15*, 295–309.

KAPLAN, H., MOLAD, E., AND TARJAN, R. E. 2003. Dynamic rectangular intersection with priorities. In *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing*. ACM Press, New York. 639–648.

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

LAMPSON, B. W., SRINIVASAN, V., AND VARGHESE, G. 1999. IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking 7*, 3, 324–334.

LIU, Y.-C. AND LEA, C.-T. 2001. Fast IP table lookup and memory reduction. In *IEEE Workshop on High Performance Switching and Routing*. Dallas, TX. 228–232.

LU, H. AND SAHNI, S. 2004a. Dynamic IP router-tables using highest-priority matching. In *IEEE Symposium on Computers and Communications*. 858–863.

LU, H. AND SAHNI, S. 2004b. $O(\log n)$ dynamic router-tables for prefixes and ranges. *IEEE Transactions on Computers 53*, 10, 1217–1230.

LU, H. AND SAHNI, S. 2005. A b-tree dynamic router-table design. *IEEE Transactions on Computers 54*, 7, 813–824.

LU, H., KIM, K. S., AND SAHNI, S. 2005. Prefix- and interval-partitioned dynamic IP router-tables. *IEEE Transactions on Computers 54*, 5, 545–557.

MCKEOWN, N. 1999. Hot interconnects tutorial slides. Stanford University, available at http://klamath.stanford.edu/talks/.

MORRISON, D. R. 1968. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of ACM 15*, 4 (Oct.), 514–534.

MULMULEY, K. 1993. *Computational Geometry, an Introduction through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

NILSSON, S. AND KARLSSON, G. 1999. IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communications*.

PATRASCU, M. AND THORUP, M. 2006. Time-space trade-offs for predecessor search. In *STOC '06: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*. ACM Press, New York. 232–240.

PELLEGRINI, M. AND VECCHIOCATTIVI, G. 2001. Empirical study of search trees for ip addressing look up. Tech. Rep. TR IMC B4-01-13, Istituto di Matematica Computazionale del CNR, Area della Ricerca, Pisa, Italy. November. Available at www.iit.cnr.it/staff/marco.pellegrini.

PELLEGRINI, M., FUSCO, G., AND VECCHIOCATTIVI, G. 2002. Adaptive stratified search trees for ip table lookup. Tech. Rep. TR IIT 22/2002, Istituto di Informatica e Telematica del CNR (IIT-CNR), Pisa, Italy (Nov.)

PUGH, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (June), 668–676.

RAMAN, R. 1995. Improved data structures for predecessor queries in integer sets.

SAHNI, S. AND KIM, K. S. 2001. Efficient construction of fixed-stride multibit tries for IP lookup. In *8th IEEE Workshop on Future Trends of Distributed Computer Systems (FTDCS)*. IEEE Computer Society, Bologna, Italy. 178–184.

SAHNI, S. AND KIM, K. S. 2002. Efficient construction of variable-stride multibit tries for IP lookup. In *Symposium on Applications and the Internet (SAINT)*. IEEE Computer Society, Los Alamitos, CA. 220–229.

SAHNI, S. AND KIM, K. S. 2003. Efficient construction of multibit tries for IP lookup. *IEEE/ACM Transactions on Networking 11*, 4, 650–662.

SAHNI, S. AND KIM, K. S. 2004. Efficient dynamic lookup for bursty access patterns. *International Journal on Foundations of Computer Science 15*, 4, 567–591.

SHARP, J., ERGUN, F., MITTRA, S., SAHINALP, C., AND SINHA, R. 2001. A dynamic lookup scheme for bursty access patterns. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*. IEEE Computer Society, Los Alamitos, CA. 1444–1453.

SKLOWER, K. 1991. A tree-based packet routing table for berkeley unix. In *USENIX Winter 1991*. 93–104.

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

SUN, X., SAHNI, S. K., AND ZHAO, Y. Q. 2004. Fast update algorithm for IP forwarding table using independent sets. In *High Speed Networks and Multimedia Communications (HSNMC 2004)*, Z. Mammeri and P. Lorenz, Eds. Lecture Notes in Computer Science, Toulouse, France. vol. 3079. Springer, New York. 324–335.

SUNDSTRÖM, M. AND LARZON, L.-Ä. 2005. High-performance longest prefix matching supporting high-speed incremental updates and guaranteed compression. In *IEEE Computer and Communications Societies (INFOCOM)*. Vol. 3. 1641–1652.

TAMMINEN, M. 1983. Analysis of N-trees. *Information Processing Letters 16*, 3 (Apr.), 131–137.

TAYLOR, D. E., LOCKWOOD, J. W., SPROULL, T. S., TURNER, J. S., AND PARLOUR, D. 2003. Scalable IP lookup for internet routers. *IEEE Journal on Selected Areas in Communications 21*, 4.

THORUP, M. 2003. Space efficient dynamic stabbing with fast queries. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9–11, 2003*, San Diego, CA. ACM, New York. 649–658.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time. *Information Processing Letters 6*, 80–82.

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

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing lookups. In *SIGCOMM*. 25–36.

WARKHEDE, P. R., SURI, S., AND VARGHESE, G. 2004. Multiway range trees: Scalable IP lookup with fast updates. *Computer Networks 44*, 3, 289–303.

WILLARD, D. E. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters 17*, 2 (24 Aug.), 81–84.

WILLARD, D. E. 1984. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences 28*, 3 (June), 379–394.

WILLARD, D. E. 1985. Searching unindexed and nonuniformly generated files in $\log \log N$ time. *SIAM Journal on Computing 14*, 4 (Nov.), 1013–1029.

WILLARD, D. E. 2000. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing 29*, 3 (June), 1030–1049.

WUU, L.-C. AND PIN, S.-Y. 2001. A fast IP lookup scheme for longest-matching prefix. In *International Conference on Computer Networks and Mobile Computing (ICCNMC)*. IEEE Computer Society, Los Alamitos, CA. 407–412.

Received March 2005; revised April 2007 and June 2007; accepted June 2007