Efficient IP Table Lookup via Adaptive Stratified Trees with Selective Reconstructions

M. Pellegrini*† and G. Fusco*

Abstract

IP address lookup is a critical operation for high bandwidth routers in packet switching networks such as Internet. The lookup is a non-trivial operation since it requires searching for the longest prefix, among those stored in a (large) given table, matching the IP address. Ever increasing routing tables 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 look up 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).

1 Introduction

Motivation for the Problem. 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 efficiency of the underlying switching infrastructure. Thus improvements in the throughput of Internet routers are likely to have a large impact. The IP Address Lookup mechanism is a critical component of an Internet Packet Switch (see [McK99] 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. 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.

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 amongst 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 [SVW99] and [LSV99]; it has been extended in [FM00] and in several recent papers [Tho03, KMT03]. It has been used in [BFK+03] to prove formally 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 [BFK+03], are based on the string view and use either optimized tries, or data compression techniques, or both. The question that we rise 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.[1] Since our objective is practical we chose for the moment not to rely on the

*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, via Moruzzi 1, 56124 Pisa, Italy. E-mail: {marco.pellegrini, giordano.fusco}@iit.cnr.it

†Corresponding Author.

[1] We notice that for the packet filtering problem the multi-dimensional FIS tree [FM00], which follows the geometric view, has already established itself as a benchmark not only for the asymptotic bounds but also for empirical performance.

sophisticated techniques that have lead to the recent improvements in worst case bounds (e.g. those in [FM00, Tho03, KMT03]). Instead we ask ourselves whether simpler search trees built with the help of local optimization routines could lead to the stated goal. The experiments with the AST method lead us to give a strong positive hint on both account: (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 4.3 the local optimization routines are described. Experiments and results for storage and query time are in Section 5, while dynamic operations are discussed in Section 6.

2 Methodology

Performance is established by means of experiments which are described in detail in Section 5. 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, at the best of our knowledge, the corresponding codes (either as source or as executables) are not in the public domain. We refrained from re-implementing 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 resolved for an admittedly weaker but still instructive comparison via the fact that those papers present, either an explicit formula for mapping their performance onto different machines[2], or such formulae could be derived almost mechanically from the description in words. A second issue was deciding the data sets to be used. Papers from the the late nineties did use table snapshots usually dating from the mid nineties, when a table of 40,000 entries was considered a large one. 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) of Internet. Therefore the comparisons shown in Table 3, where we use data sets of comparable size and the mapping onto a common architecture given by the formulae, should be considered just in a broad qualitative sense. The main general conclusion we feel we can safely draw is that geometric view is as useful as insight drawn from string manipulation and data compression.

We remark that the extensive testing with 12 Lookup Tables of size ranging from 17,000 to 142,000 (reported in Tables (2), and (5)) confirms the reliability and robustness of the AST.

3 Previous work

The literature on the IP lookup problem and its higher dimensional extension[3] 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 searching techniques (ii) design of specialized hardware aiming at exploiting machine-level parallelism (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 [Knu73]. Patricia Tries [Mor68] take advantage of common sub-sequences to compress certain paths in a binary trie thus saving both in search time and storage. The first IP Lookup algorithm in [Skl91] is based on the Patricia Trie idea. Nilsson and Karlsson [NK99] 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 non-empty.

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

[3]Sometimes the IP lookup problem is treated as the 1-dimensional special case of the more general *packet filtering* problem. Similarly in geometric terms, the predecessor problem in 1-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 and this may generate some confusion.

Waldvogel et al. [WVTP97] organize the prefixes in groups of prefixes of same length and each group is stored in an 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 in [Wil83]). The number of hash tables can be reduced by padding some of the prefixes [SV99]. Good hashing strategies are studied in [BM01].

In [SV99] 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 [BFK$^+$03] the trie is augmented by compressing all leaves and internal nodes via shortest common super-string compression [BJL$^+$94]. It should be noticed that while in [SV99] the model states a 1 Mbyte cap on storage and tries to attain the best worst case time within the cap (which is also our model), the emphasis in [BFK$^+$03] is different: the target is a good average time for random traffic with no limit on storage, thus up to 3 Mbytes are used to store a table of 118K entries .

Dagemark et al. [DBCP97] use data compression techniques to store compactly 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. [CDG99] instead start from a full table representation of the Lookup function then apply a data compression technique that reduces the storage to acceptable levels in practice while requiring only 3 memory accesses to answer a query.

The Multiway Range Tree [SVW99] and the Multiway seach tree [LSV99] 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. [SEM$^+$01] and [ESSS01] use the fast reconfiguration capabilities of skip list [Pug90] 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 [FM00] 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 [Tho03] improves the Dynamic FIS Tree: the storage becomes linear in n and any sequence of dynamic operations is supported efficiently. Kaplan, Molad and Tarjan [KMT03] 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 The AST in a nutshell.

The AST algorithm is described in Section (4.3). 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 for s a power of two 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. We choose thus 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. So the search for the optimal step size includes an inner optimization loop on the choice of the anchor to minimize the maximum bucket occupation. The construction of the (locally) optimal step and anchor can be done efficiently in time close to linear, up to polylog terms (see Section 4.3). 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 a few long paths. In the second phase the long paths are shortened by compressing them.

During the construction a basic sub-task is finding a grid optimizing a local measure. 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 [CM99], [GPB00], [SV99], and [BFK$^+$03]. Ioannidis, Grama and Atallah [IGA03] recently proposed a global reconstruction method for tries that relies on a reduction to the knapsack prob-

lem and therefore suggests the use of knapsack approximation methods. In [IGA03] 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 is shown in [PFV02] giving asymptotic worst case bounds $O(n)$ for storage and $O(\log n)$ for query time. However the query time bound does not explain the observed performance and we leave it as an open problem to produce a more refined analysis considering a variety of models.

4.2 The Problem

Except maybe for the notion of a *phantom point*, the concepts shown in this subsection are by now standard; they has been introduced for sake of completeness and to set up the background for the AST construction.
The string approach. Consider the binary alphabet $\Sigma = \{0, 1\}$. The set of strings of length $w \in \mathbb{N}$ with alphabet Σ is Σ^w. The empty string ϵ corresponds to Σ^0. A string $p \in \Sigma^u$ is a prefix of a string $s \in \Sigma^w$, for $u \leq w$, when there exists a string $q \in \Sigma^{w-u}$ such that $s = p \cdot q$ where $\cdot$ denotes the operation of concatenation of two strings. If p is a prefix of s then we say the predicate $prefix(p, s)$ is true. If $s \in \Sigma^w$ then $|s| = w$ is called the length of the string s. Consider the set P^w of prefixes of Σ^w, that is $P^w = \bigcup_{0 \leq v \leq w} \Sigma^v$, and a subset $P' \subseteq P^w$ of cardinality n. Consider a set H of labels for the next hop exit line.

Definition 1 *A Forwarding Table for a set of prefixes $P' \subseteq P^w$ is a mapping $T : P' \to H$.*

Given a string $s \in \Sigma^w$, the longest prefix match for s in P' is

$$\mathrm{lpm}(s, P') = argmax\{|p| : p \in P', prefix(p, s)\}.$$

If we assume that $\epsilon \in P'$ then $\mathrm{lpm}(s, P')$ is always defined and unique.

Definition 2 *Longest prefix match problem: Given a forwarding table $T : P' \to H$ we want to store T so that for every $s \in \Sigma^w$ we compute efficiently $T(\mathrm{lpm}(s, P'))$.*

The geometric (or locus) approach. Consider now the set Σ^w as a representation in binary notation of the sequence of natural numbers $U = [0, .., 2^w - 1]$ thus we have a natural 1-1 mapping $m : \Sigma^w \to U$ that takes a string to a number. The prefix $p \in P^w$ is associated with the set

$$M(p) = \{m(s) : s \in \Sigma^w, prefix(p, s)\}$$

The set $M(p)$ is contiguous is the sense that $M(p) = [\min M(p), \max M(p)]$. For example if $p = \epsilon$ then $M(\epsilon) = U$. The set is $M(p)$ uniquely identified by its extremal points. We thus introduce the set $Extr(P')$ of extreme points of segments from prefixes in P'.
Some useful properties:

Property 1 *If $prefix(p, s)$ then $M(s) \subseteq M(p)$.*

Property 2 *If $\neg(prefix(p, s) \bigvee prefix(s, p))$ then $M(s) \cap M(p) = \emptyset$.*

We can associate to each point in U a next hop $f : U \to H$ as follows:

$$f(a) = T(\mathrm{lpm}(m^{-1}(a), P'))$$

It is easy to show that, equivalently, for any $a \in Extr(P')$ we have $f(a) = T(p^*)$ where

$$p^* = argmax_{p \in P'}\{|p| : prefix(p, m^{-1}(a))\}.$$

Equivalently:

$$p^* = argmax_{p \in P'}\{|p| : a \in [\min M(p), \max M(p)]\}.$$

That is, the next hop for a is the next hop associated with the smallest set containing a among the sets $\{M(p) : p \in P'\}$. This set is uniquely defined. Thus we need to solve on-line smallest segment stabbing queries. Since these M-sets are defined by their extreme points, we can see them embedded in $\mathbb{R}$ and define the segment $s(p) = \{q \in \mathbb{R} : \min M(p) \leq q \leq \max M(p)\}$. Thus we have a problem of finding the next hop associated to the smallest segment intersected by a query point a.
Semi-closed intervals. In real tables it happens quite often that the M-sets are abutted meaning that for two prefixes p' and p'', $\max M(p') + 1 = \min M(p'')$. In this case we can reduce considerably the size of the set $Extr(P')$ by considering our intervals closed on the left by 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 $[a, b+1[, [b+1, c+1[$ thus only 3 distinct points need to be recorded[4] instead of 4.

[4]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.

Reduction to the predecessor problem. Consider the predecessor problem for $Extr(P')$. This consists in storing $Extr(P')$ in a data structure so that for each query $a \in U$ the point $pred(a) = \max\{q \in Extr(P') : q \leq a\}$ is found. Next we notice that if $pred(a') = pred(a'')$ then $f(a) = f(a'')$. That is, the next hop is the same for points having the same predecessor. So we can determine a mapping $Q : Extr(P') \cap U \to H$ so that for each $a \in U$: $f(a) = Q(Pred(a))$.
Note that in the dynamic case the removal of a prefix in T may require extensive reconstruction of the function Q, although the set $Extr(P')$ and the corresponding data structure for predecessor searching might be updated efficiently.

Definition 3 *If two intervals share an end-point than one of the two identical points is classified as a* duplicate *point.*

Definition 4 *If* $Q(Pred(a)) = Q(Pred(Pred(a)-1)$ *then the point* $Pred(a)$ *is called a* phantom.

A phantom is useless to discriminate among co-domain values and it can be eliminated from the set without altering the function $f()$. As seen in section 5 both duplicates and phantom represent a sizable portion of the points in actual tables.

4.3 Construction of the AST

Here we describe the AST data structure by iteratively refining a general framework with specific choices. Consider initially the set $U = [0, .., 2^w - 1]$ of all possible addresses, the set P of prefixes, and the set $S = S_P$ of end-points of the segments obtained by mapping each prefix $b \in P$ to the set of addresses of U matching b as described above. We will often think of U as embedded in the real line $\mathbb{R}$.
The main idea is to build recursively a tree by levels, each node x of the tree has an associated connected subset of the universe $U_x \subset U$, which is the set of all queries that visit node x, and a point data set $S_x \subset S$, which is $S \cap U_x$. For the root r we have $U_r = U$ and $S_r = S$.
Let x be a node and let k_x be the number of children on x. By $y[1], ..., y[k_x]$ we denote the children of x. A particular rule (to be described below) will tell us how to decide k_x and for each $i \in [1, .., k_x]$ how to compute $U_{y[i]}$. Afterwards simply $S_{y[i]} = U_{y[i]} \cap S_x$.
For each child $y[i]$, if $S_{y[i]}$ is empty we store at $y[i]$ the next-hop index of the shortest segment covering $y[i]$; and we stop the recursion. For each child $y[i]$, if $S_{y[i]}$ has one point we store at $y[i]$ the next-hop indices for the two intervals to the left and right of that point; and we stop the recursion.
For simplicity we explain the process at the root. Let S be a set of points on the real line, and $span(S)$ the smallest interval containing S. Consider an infinite grid $G(a, s)$ of anchor a and step s, $G(a, s) = \{a+ks : k \in N\}$. 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 $\alpha \in [0, 1]$. Consider an event a value α_i for which $S \cap f_G(\alpha_i) \neq \emptyset$.

Lemma 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 meet 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 range beyond 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 $\alpha_p = (p - left(I_p))/s$ 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 count for a bucket and increasing it for the neighbour 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 life-time of the algorithm. This value is the

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

that is we find the shift that for a given step s minimizes the maximal occupancy. The whole algorithm takes time $O(n \log n)$.
In order to use a binary search scheme we need some monotonicity property which we are going to prove next.

Lemma 2 *Take two step values* s *and* t, *with* $t = 2s$ *then we have* $g(t) \geq g(s)$.

Proof. Consider the grid $G_{min}(t)$ that attains min-max occupancy $K = g(t)$. So every bucket in $G(t)$ 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. $g(s) \leq K = g(t)$. ∎

Now if we use only powers of two as possible values for a grid step the above monotonicity lemma implies that, for a given threshold x, we can use binary search to find the largest step $s = 2^k$ with $g(s) \leq x$ and the smallest step $u = 2^h$ with $g(u) \geq x$.
Call $R = E/F$ the ratio of the number empty to the full buckets.

Lemma 3 *Take two step values s and t, with $t = 2s$ 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 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. ∎

By the above lemmas we have two functions $g(.)$ and $R_{min}(.)$ that are both monotone (one increasing, one decreasing) in the step size that we can compute efficiently.

Phase I: Saving Space. In order to control memory consumption we adopt the following criterion recursively and separately at each node: find smallest s with $R(s) \leq C$, for a predefined constant threshold C then take the grid giving the occupancy $g(s)$, the min-max occupancy (this shifting does not change the number of nodes at the level). When the number of points in a bucket drops below a (small) threshold $2D$ we stop the recursion and we switch to standard sequential search[5].

Phase II: Selective Reconstruction. The tree built in Phase I uses linear space 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.

An access the RAM memory is about ten times slower than one to L2 cache. Thus it is fundamental to fit all the relevant data to into the L2 cache. It is reasonable to suppose that the target machine has 1 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 difference unless it is below 1 Mb. When the routing table is built, we can perform a selective reconstruction of the deepest paths to flatten the AST in order to reduce the worst query time maintaining the total memory consumption below 1 Mb.

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. Thus the cost of an empty leaf is the time to query a point inside the space of that leaf. 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.

The leaves can be subdivided into classes (l, p, a) depending on: their level in the AST, the number of points from the median to the farthest extreme and the number of bits 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 to perform a query in the minimum time measured in the test machine because the processor is not disturbed by 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] Since D in our experiments is very small, smaller than 3, we use to sequential search; for larger values binary search might be the method of choice

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

Selective Reconstruction.
The following steps are repeated while the size of the Routing Table is below 1 Mb and there are other reconstructions to perform: (i) Create a max-priority queue of internal nodes based on the maximum cost of their children. (ii) 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. (iii) 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 then split the *step* until the maximum number of points in a bucket becomes smaller than the current maximum; otherwise go 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 same cost are performed only after recomputing the priority queue.

5 Experiments with AST

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 (2) 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, next we classify the points into active, phantom and duplicate points, giving the count for each category.
Table (5) gives the relevant measure for the Slim-AST. Slim-AST is built only on the active points (without duplicates and phantom points) and it can support the queries. Updates are supported with the help of an auxiliary data structure that records also the missing data (duplicates and phanthoms).
Normalization in L2. To test the AST data structure we adopt the methodology in [DBCP97]. 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 with the schema described above [7]. 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. Afterwards 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 treated separately from accessing any other level of the AST. The results of the L2-normalized measures are shown in table (1) for 8-bit keys, access times for 16 and 32 bit keys are almost identical.
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. Thus the general formula for accessing a point (k, s) is:

$$A + k(M + m + B) + (M + m) + (s - 1)(C + m)$$

Where we account for any initial constant, for accessing the intermediate nodes, for accessing the leaf and for accessing sequentially points stored at the leaf. Now taking the measurements on our machine and the known values of M and m we determine the constants A, B, C. Since in the slim table we have only results for classes from $(2,1)$ to $(2,4)$ in Table (1) we need accuracy in this range. Via easy calculations, the final interpolation formula for any $(2, s)$ class is: $3(M + m) + (s - 1)m$. In Table (3) we compare the AST (slim) method with three other methods for which analogous formulae have been derived by the respective authors (explicitly or implicitly), mapped onto a common architecture. The method in [DBCP97] is by far the most storage efficient, using roughly 4 bytes per entry; it is based on tree compression techniques, which make updates difficult. The query time in somewhat high. The Expansion/Compression method [CDG99] is very fast (requires only 3 memory access) using a fair amount of memory and requiring extensive reconstructions of

[7] As in [DBCP97] we exclude from measurement the function invocation overhead and consider the search completed when we retrieve the index of the next-hop.

the data structure at each update[8]. The method of [SV99] balances well query time, storage and update time. Its performance for tables of 40,000 entries is not far form the one of the AST. It is not clear however how the balance of resource utilization would scale on tables three times as large.

Table (4) is taken from [SV99] and is useful to get an idea of the gains in storage and time that recent research has brought forward. We list these methods separately since for them (except [DBCP97]) formulae have not been derived that allow a rigorous scaling.

Clearly as processors become faster the time is dominated by the memory access time. So we have our method and [CDG99] converge to roughly the same query time, however our use of storage seems more efficient.

6 Dynamic Operations

Description of the dynamic operations. We give a brief sketch of the dynamic operations. Inserting a new prefix s involves two phases: (1) inserting the end-points of the corresponding segment $M(s)$; (2) updating the next-hop index for all the leaves that are covered by the segment $M(s)$. 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 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 sub-trees with a span included into a segment that is included in $M(s)$ 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[9].

Deleting prefix s from the AST involves phases similar to insertion, however we need to determine first the segment in the AST including $M(s)$ in order to perform the leaf re-labeling. This is done by performing at most w searches, one for each prefix of s, and selecting the longest such prefix present in the AST. 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.

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[10]. Table (6) shows the number of memory accesses and the percentile time in μ-seconds for completing 95%, 99% and 100% of the updates (insertions) for the AST data structures built for the lookup tests. Experiments for deletion give almost identical 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.

As it is to be expected, the shortest prefixes, corresponding to the longest segments, require a visit of larger portions of the tree and are responsible for the slowest updates.

Comparison with known methods. In [SV99] an estimate of 2500 μ-sec 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 *nsec* 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 μ-seconds. Although our machine is faster than that used in [SV99] the L2 access time is quite similar, therefore the timings in [SV99] can be compared (although only on a qualitative basis) with results in table 6. We believe that the 2500 μ-sec worst case estimate for [SV99] is overly pessimistic: in our experiments (deletion and insertion of every entry of every table) we never had to rebuild nodes of very high degree.

Updates in [DBCP97] and [CDG99] are handled by rebuilding the data structure from scratch, thus requiring time of the order of several milliseconds for every update.

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

[9] This rough argument can be made precise in an amortized sense by using the model of (N,Δ)-sequences of updates of Mulmuley [Mul93].

[10] 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.

7 Acknowledgements

We thank Giogio Vecchiocattivi who contributed to testing an early version of the AST data structure [PFV02], and the partecipants of the mini-workshop on routing (held in Pisa in March 2003) for many interesting discussions.

8 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 choosing properly 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 the AST good empirical performance is still an open issue for further research. A second open line of research is the integration of the AST techniques for empirical efficiency within the framework of the Fat Inverted Segment tree which presently yield good worst case asymptotic bounds

References

[BFK+03] Adam L. Buchsbaum, Glenn S. Fowler, Balachander Krishnamurthy, Kiem-Phong Vo, and Jia Wang. Fast prefix matching of bounded strings, 2003. In Proceedings of Alenex 2003.

[BJL+94] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, July 1994.

[BM01] Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *INFOCOM*, pages 1454–1463, 2001.

[CDG99] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi. IP address lookup made fast and simple. In *European Symposium on Algorithms*, pages 65–76, 1999.

[CM99] Gene Cheung and Steven McCanne. Optimal routing table design for IP address lookups under memory constraints. In *INFOCOM (3)*, pages 1437–1444, 1999.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.

[ESSS01] Ergun, Sahinalp, Sharp, and Sinha. Biased skip lists for highly skewed access patterns. In *ALENEX: International Workshop on Algorithm Engineering and Experimentation, LNCS 2153*, pages 216–229, 2001.

[FM00] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings of INFOCOM, volume 3*, pages 1193–1202. IEEE, March 2000.

[GPB00] Pankaj Gupta, Balaji Prabhakar, and Stephen P. Boyd. Near optimal routing lookups with bounded worst case performance. In *INFOCOM (3)*, pages 1184–1192, 2000.

[IGA03] Ioannis Ioannidis, Ananth Grama, and Mikhail Atallah. Adaptive data structures for ip lookups. In *INFOCOM '03*, 2003.

[KMT03] Haim Kaplan, Eyal Molad, and Robert E. Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 639–648. ACM Press, 2003.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol.3 — Sorting and Searching.* Addison-Wesley (Reading MA), 1973.

[LSV99] Butler W. Lampson, Venkatachary Srinivasan, and George Varghese. IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking*, 7(3):324–334, 1999.

[McK99] Nick McKeown. Hot interconnects tutorial slides. Stanford University, available at http://klamath.stanford.edu/talks/, 1999.

[Mor68] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of ACM*, 15(4):514–534, October 1968.

[Mul93] K. Mulmuley. *Computational Geometry, an Introduction through Randomized Algorithms.* Prentice Hall, 1993.

[NK99] S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communications*, 1999.

[PFV02] M. Pellegrini, G. Fusco, and G. Vecchiocattivi. Adaptive stratified search trees for ip table lookup. Technical Report TR IIT 22/2002, Istituto di Informatica e Telematica del CNR (IIT-CNR), Pisa,Italy, November 2002.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction.* Springer Verlag, 1985.

[Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[SEM+01] Jonathan Sharp, Funda Ergun, Suvo Mittra, Cenk Sahinalp, and Rakesh Sinha. 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)*, pages 1444–1453, Los Alamitos, CA, April 22–26 2001. IEEE Computer Society.

[Skl91] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter 1991*, pages 93–104, 1991.

[SV99] Venkatachary Srinivasan and George Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, pages 1–40, 1999.

[SVW99] S. Suri, G. Varghese, and P. Warkhede. Multiway range trees: Scalable ip lookup with fast updates. Technical Report 99-28, Washington University in St. Luis,Dept. of Computer Science, 1999.

Position	Level				
	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

Table 1: L2-normalized costs for points of 8 bits. Measures in clock ticks. The first negative entry is due to 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.

	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

Table 2: List of Input Lookup Tables. Input geometric statistics for active, duplicates and phantom points.

[Tho03] M. Thorup. 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, USA*, pages 649–658. ACM, 2003.

[vKO93] Marc J. van Kreveld and Mark H. Overmars. Union-copy structures and dynamic segment trees. *Journal of the ACM (JACM)*, 40(3):635–652, 1993.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983.

[WS95] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated Volume 2 - The Implementation.* Addison Wesley, 1995.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. In *SIGCOMM*, pages 25–36, 1997.

Ref	Method	Input size	Formula	Time (ns)	Memory (KB)
[DBCP97]	Lulea	38141	53T +12D	254	160
[SV99]	Variable stride	38816	26T + 3D	81	655
[CDG99]	Exp./Con.	44024	3T +3D	49	1057
This paper	Slim AST	45184	m + 3D	49	464
This paper	Slim AST	142475	m + 3D	49	941

Table 3: Comparisons of IP methods for a 700 MHz Pentium III (T=1.4ns) with 1024 KB L2 Cache L2 delay is D=15 nsec, L1 latency m= 4 nsec

Ref	Method	Average (nsec)	Worst (nsec)	Size (kB)	Notes
[Mor68, WS95]	Patricia trie	1500	2500	3262	BSD Implementation
[LSV99]	6-way search	490	490	950	Implemented
[WVTP97]	Binary search on levels	250	650	1600	Implemented
[DBCP97]	Lulea scheme	349	409	160	Projected
[NK99]	LC Tries	1000	-	700	Projected

Table 4: Comparisons of some IP methods for a 300 MHz Pentium II with 512 KB L2 Cache for Table Mae East of September 12, 1997 holding 38816 prefixes. L2 latency is 15 nsec. Source [SV99].

	Routing Table	Entries	Target Cache	Total Memory	Worst query (clock-ticks)	Corresponding Class (Lev,Pts,Base)
1	Paix	17,766	512	374,314	35	2,2, 8
2	AADS	32,505	512	460,516	35	2,2, 8
3	Funet	41,328	512	453,034	30	2,1, 8
4	Pac Bell	45,184	512	464,063	35	2,2, 8
5	Mae West	71,319	1024	610,587	35	2,2, 8
6	Telstra	104,096	1024	780,452	35	2,2,16
7	Oregon	118,190	1024	595,919	35	2,3, 8
8	AT&T US	121,613	1024	955,702	30	2,1, 8
9	Telus (*)	126,282	1024	878,179	40	2,4, 8
10	AT&T East Canada	127,157	1024	878,155	40	2,4, 8
11	AT&T West Canada	127,248	1024	878,491	40	2,4, 8
12	Oregon (*)	142,475	1024	940,810	35	2,2,16

Table 5: Slim-AST data structure performance. Tests on a 700 MHz Pentium III (T=1.4ns) with 1024 KB L2 Cache L2 delay is D=15 nsec, L1 latency m= 4 nsec. The parameters used are: $B = 16$ $C = 10^{-7}$ $D = 61$ for all tables except (*) where $C = 1$.

	Routing Table	100%			99%			95%		
		Accesses		Time	Accesses		Time	Accesses		Time
		L2	L1	μ-sec	L2	L1	μ-sec	L2	L1	μ-sec
1	Paix	202	26	3.13	59	392	2.45	31	200	1.27
2	AADS	590	8	8.88	59	392	2.45	31	200	1.27
3	Funet	4285	160	64.92	79	492	3.15	61	406	2.54
4	PacBell	1298	156	20.09	71	484	3.00	35	230	1.45
5	MaeWest	1943	86	29.49	89	614	3.79	49	324	2.03
6	Telstra	1948	32	29.35	81	550	3.42	49	326	2.04
7	Oregon	4381	1602	72.12	457	3188	19.61	87	592	3.67
8	AT&T US	1675	80	25.45	69	462	2.88	42	266	1.69
9	Telus	1686	14	25.35	73	492	3.06	45	294	1.85
10	AT&T East Canada	1686	14	25.35	81	544	3.39	49	332	2.06
11	AT&T West Canada	1686	14	25.35	81	544	3.39	49	332	2.06
12	Oregon	2309	80	34.96	467	3246	19.99	66	434	2.73

Table 6: Slim-AST data structure performance of insertion. 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=1.4ns) with 1024 KB L2 Cache L2 delay is D=15 nsec, L1 latency m= 4 nsec.