Efficient Construction Of Variable-Stride Multibit Tries For IP Lookup *

Sartaj Sahni & Kun Suk Kim
{sahni, kskim}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

Srinivasan and Varghese [17] have proposed the use of multibit tries to represent routing tables used for Internet (IP) address lookups. They propose an $O(n * W^2 * k)$ dynamic programing algorithm to determine the strides for an optimal variable-stride trie that has at most k levels. Here, n is the number of prefixes in the routing table and W is the length of the longest prefix. We improve on this algorithm by providing an alternative dynamic programming formulation. The complexity of our algorithm is $O(n * W * k)$, on real router data sets. This is an improvement by a factor of W over the corresponding algorithm of [17]. Experiments conducted by us indicate that our variable-stride algorithm is between 2 and 17 times as fast for IPv4 routing table data.

Keywords*: Packet routing, longest matching prefix, controlled prefix expansion, multibit trie, dynamic programming.*

1 Introduction

With the doubling of Internet traffic every three months [18] and the tripling of Internet hosts every two years [6], the importance of high speed scalable network routers cannot be over emphasized. Fast networking "will play a key role in enabling future progress" [11]. Fast networking requires fast routers and fast routers require fast router table lookup.

An Internet router table is a set of tuples of the form (p, a), where p is a binary string whose length is at most W (W = 32 for IPv4 destination addresses and W = 128 for IPv6), and a is an output link (or next hop). When a packet with destination address A arrives at a router, we are to find the pair (p, a) in the router table for which p is a longest matching prefix of A (i.e., p is a prefix of A and there is no longer prefix q of A such that (q, b) is in the table). Once this pair is determined, the packet is sent to ouput link a. The speed at which the router can route packets is limited by the time it takes to perform this table lookup for each packet.

Longest prefix routing is used because this results in smaller and more manageable router tables. It is impractical for a router table to contain an entry for each of the possible destination addresses. Two of the reasons this is so are (1) the number of such entries would be almost one hundred million and would triple every three years, and (2) every time a new host comes online, all router tables will need to incorporate the new host's address. By using longest prefix routing, the size of router tables is contained to a reasonable quantity and information about host/router changes made in one part of the Internet need not be propagated throughout the Internet.

Several solutions for the IP lookup problem (i.e., finding the longest matching prefix) have been proposed. IP lookup in the BSD kernel is done using the Patricia data structure [16], which is a variant of a compressed binary trie [7]. This scheme requires $O(W)$ memory accesses per lookup. We note that the lookup complexity of longest prefix matching algorithms is generally measured by the number of accesses made to main memory (equivalently, the number of cache misses). Dynamic prefix tries, which are an extension of Patricia, and which also take $O(W)$ memory accesses for lookup, have been proposed by Doeringer et al. [5]. LC tries for longest prefix matching are developed in [13]. Degermark et al. [4] have proposed a three-level tree structure for the routing table. Using this structure, IPv4 lookups require at most 12 memory accesses. The data structure of [4], called the Lulea scheme, is essentially a three-level fixed-stride trie in which trie nodes are compressed using a bitmap. The multibit trie data structures of Srinivasan and Varghese [17] are, perhaps, the most flexible and effective trie structure for IP lookup. Using a technique called controlled prefix expansion, which is very similar to the technique used in [4], tries of a predetermined height (and hence with a predetermined number of memory accesses per lookup) may be constructed for any prefix set.

*This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

Srinivasan and Vargese [17] develop dynamic programming algorithms to obtain space optimal fixed-stride and variable-stride tries of a given height. Sahni and Kim [15] develop an alternative dynamic programming algorithm for fixed-stride tries. For IPv4 router databases, their algorithm is 2 to 4 times as fast as that of [17].

Waldvogel et al. [19] have proposed a scheme that performs a binary search on hash tables organized by prefix length. This binary search scheme has an expected complexity of $O(\log W)$. An alternative adaptation of binary search to longest prefix matching is developed in [8]. Using this adaptation, a lookup in a table that has n prefixes takes $O(W + \log n)$ time.

Cheung and McCanne [3] develop "a model for table-driven route lookup and cast the table design problem as an optimization problem within this model." Their model accounts for the memory hierarchy of modern computers and they optimize average performance rather than worst-case performance.

Hardware solutions that involve the use of content addressable memory [9] as well as solutions that involve modifications to the Internet Protocol (i.e., the addition of information to each packet) have also been proposed [2, 12, 1].

In this paper, we focus on the controlled expansion technique of Srinivasan and Varghese [17]. In particular, we develop a new dynamic programming formulation for the construction of space optimal variable-stride tries of a predetermined height. The resulting algorithm is asymptotically faster, by a factor of W, (on real router data sets) than the corresponding algorithm of [17]. Experiments with IPv4 prefix sets indicate that our variable stride-algorithm is 2 to 17 times as fast as that of [17].

In Section 2, we develop our new dynamic programming formulation, and in Section 3, we present our experimental results.

2 Construction Of Multibit Tries

2.1 1-Bit Tries

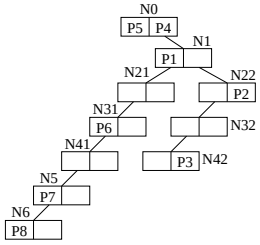
A *1-bit trie* is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level $l-1$ of the trie store prefixes whose length is l (the length of a prefix is the number of bits in that prefix; the terminating * (if present) does not count towards the prefix length). If the rightmost bit in a prefix whose length is l is 0, the prefix is stored in the left data field of a node that is at level $l-1$; otherwise, the prefix is stored in the right data field of a node that is at level $l-1$. At level i of a trie, branching is done by examining bit i (bits are numbered from left to right beginning with the number 0, and levels are numbered with the root being at level 0) of a prefix or destination address. When bit i is 0, we move into the left subtree; when the bit is 1, we move into the right subtree. Figure 1(a) gives the prefixes in the 8-prefix example of [17], and Figure 1(b) shows the corresponding 1-bit trie. The prefixes in Figure 1(a) are numbered and ordered as in [17]. Since the trie of Figure 1(b) has a height of 6, a search into this trie may make up to 7 memory accesses. The total memory required for the 1-bit trie of Figure 1(b) is 20 units (each node requires 2 units, one for each pair of (child, data) fields). The 1-bit tries described here are an extension of the 1-bit tries described in [7]. The primary difference being that the 1-bit tries of [7] are for the case when all keys (prefixes) have the same length.

Database	Number of prefixes	Number of 16-bit prefixes	Number of 24-bit prefixes
Paix	85682	3051	37413
Pb	35151	1705	15516
MaeWest	30599	1625	13137

Table 1. Prefix databases obtained from IPMA project[10] on Sep 13, 2000. The last column shows the number of nodes in the 1-bit trie representation of the prefix database. Note that the number of prefixes stored at level i of a 1-bit trie equals the number of prefixes whose length is $i+1$.

Original prefixes
P5=0*
P1=10*
P2=111*
P3=11001*
P4=1*
P6=1000*
P7=100000*
P8=1000000*

(a) 8-prefi x example of [17]



(b) Corresponding 1-bit trie

Figure 1. Prefixes and corresponding 1-bit trie

When 1-bit tries are used to represent IPv4 router tables, the trie height may be as much as 31. A lookup in such a trie takes up to 32 memory accesses. Table 1 gives the characteristics of three IPv4 backbone router prefix sets. For our three databases, the number of nodes in a 1-bit trie is between $2n$ and $3n$, where n is the number of prefixes in the database.

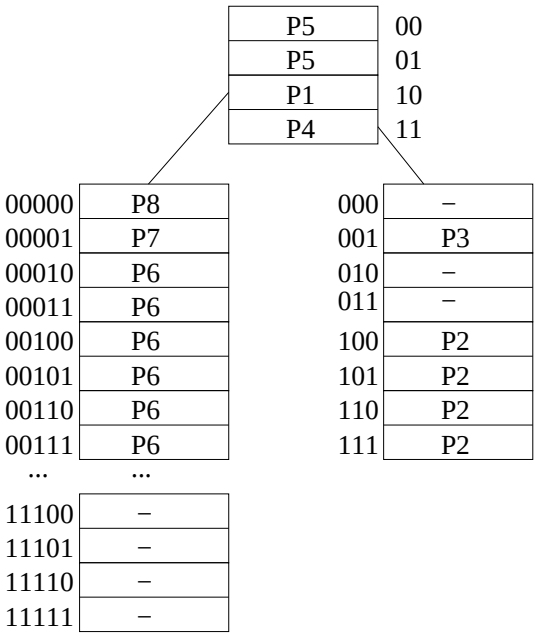


Figure 2. Two-level VST for prefixes of Figure 1(a)

2.2 Variable-Stride Tries

2.2.1 Definition and Construction

In a *variable-stride trie* (VST) [17], nodes at the same level may have different strides. Figure 2 shows a two-level VST for the 1-bit trie of Figure 1. The stride for the root is 2; that for the left child of the root is 5; and that for the root's right child is 3. The memory requirement of this VBT is 4 (root) + 32 (left child of root) + 8 (right child of root) = 44.

Let r-VST be a VST that has at most r levels. Let $Opt(N, r)$ be the cost (i.e., memory requirement) of the best r-VST for a 1-bit trie whose root is N. Srinivasan and Varghese [17], have obtained the following dynamic programming recurrence for $Opt(N, r)$.

$$Opt(N,r) = \min_{s\in\{1\ldots1+height(N)\}}\{2^s + \sum_{M\in D_s(N)} Opt(M, r-1)\},\ r>1 \quad (1)$$

where $D_s(N)$ is the set of all descendents of N that are at level s of N. For example, $D_1(N)$ is the set of children of N and $D_2(N)$ is the set of grandchildren of N. $height(N)$ is the maximum level at which the trie rooted at N has a node. For example, in Figure 1(b), the height of the trie rooted at N1 is 5. When $r = 1$,

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

The cost of covering all levels of N using at most one expansion level is $2^{1+height(N)}$. When more than one expansion level is permissible, the stride of the first expansion level may be any number s that is between 1 and $1 + height(N)$. For any such selection of s, the next expansion level is level s of the 1-bit trie whose root is N. The sum in Equation 1 gives the cost of the best way to cover all subtrees whose roots are at this next expansion level. Each such subtree is covered using at most $r-1$ expansion levels. It is easy to see that $Opt(R, k)$, where R is the root of the overall 1-bit trie for the given prefix set P, is the cost of the best k-VST for P. Srinivasan and Varghese [17], describe a way to determine $Opt(R, k)$ using Equations 1 and 2. The complexity of their algorithm is $O(n * W^2 * k)$, where n is the number of prefixes in P and W is the length of the longest prefix.

By modifying the equations of Srinivasan and Varghese [17] slightly, we are able to compute $Opt(R, k)$ in $O(mWk)$ time, where m is the number of nodes in the 1-bit trie. Since $m = O(n)$ for realistic router prefix sets, the complexity of our algorithm is $O(nWk)$. Let

$$Opt(N, s, r) = \sum_{M\in D_s(N)} Opt(M, r),\ s > 0,\ r > 1,$$

and let $Opt(N, 0, r) = Opt(N, r)$. From Equations 1 and 2, we obtain:

$$Opt(N,0,r) = min_{s\in\{1\ldots1+height(N)\}}\{2^s + Opt(N, s, r-1)\},\ r>1 \quad (3)$$

and

$$Opt(N, 0, 1) = 2^{1+height(N)}. \quad (4)$$

For $s > 0$ and $r > 1$, we get

$$\begin{aligned} Opt(N,s,r) &= \sum_{M\in D_s(N)} Opt(M, r) \\ &= Opt(LeftChild(N), s-1, r) \\ &+ Opt(RightChild(N), s-1, r). \end{aligned} \quad (5)$$

For Equation 5, we need the following initial condition:

$$Opt(null, *, *) = 0 \quad (6)$$

With the assumption that the number of nodes in the 1-bit trie is $O(n)$, we see that the number of $Opt(*, *, *)$ values is $O(nWk)$. Each $Opt(*, *, *)$ value may be computed in $O(1)$ time using Equations 3 through 6 provided the Opt

values are computed in postorder. Therefore, we may compute $Opt(R,k) = Opt(R,0,k)$ in $O(nWk)$ time. Our algorithm requires $O(W^2k)$ memory for the $Opt(*,*,*)$ values. To see this, notice that there can be at most $W+1$ nodes N whose $Opt(N,*,*)$ values must be retained at any given time, and for each of these at most $W+1$ nodes, $O(Wk)$ $Opt(N,*,*)$ values must be retained. To determine the optimal strides, each node of the 1-bit trie must store the stride s that minimizes the right side of Equation 3 for each value of r. For this purpose, each 1-bit trie node needs $O(k)$ space. Since the 1-bit trie has $O(n)$ nodes in practice, the memory requirements of the 1-bit trie are $O(nk)$. The total memory required is, therefore, $O(nk + W^2k)$.

In practice, we may prefer an implementation that uses considerably more memory. If we associate a cost array with each of the $O(n)$ nodes of the 1-bit trie, the memory requirement increases to $O(nWk)$. The advantage of this increased memory implementation is that the optimal strides can be recomputed in $O(W^2k)$ time (rather than $O(nWk)$) following each insert or delete of a prefix. This is so because, the $Opt(N,*,*)$ values need be recomputed only for nodes along the insert/delete path of the 1-bit trie. There are $O(W)$ such nodes.

2.2.2 Faster $k = 2$ Algorithm

The algorithm of Section 2.2.1 may be used to determine the optimal 2-VST for a set of n prefixes in $O(mW)$ (equal to $O(nW)$ for practical prefix sets) time, where m is the number of nodes in the 1-bit trie and W is the length of the longest prefix. In this section, we develop an $O(m)$ algorithm for this task.

From Equation 1, we see that the cost, $Opt(root,2)$ of the best 2-VST is

$$\begin{aligned} Opt(root,2) &= min_{s\in\{1\ldots1+height(root)\}}\{2^s \\ &\quad + \sum_{M\in D_s(root)} Opt(M,1)\} \\ &= min_{s\in\{1\ldots1+height(root)\}}\{2^s \\ &\quad + \sum_{M\in D_s(root)} 2^{1+height(M)}\} \\ &= min_{s\in\{1\ldots1+height(root)\}}\{2^s \\ &\quad + C(s)\} \end{aligned} \tag{7}$$

where

$$C(s) = \sum_{M\in D_s(root)} 2^{1+height(M)} \tag{8}$$

We may compute $C(s)$, $1 \le s \le 1 + height(root)$, in $O(m)$ time by performing a postorder traversal (see Figure 3) of the 1-bit trie rooted at $root$. Here, m is the number of nodes in the 1-bit trie. Since $m = O(n)$, where n is the number of prefixes, for practical data sets, the complexity of the algorithm of Figure 3 is $O(n)$ on practical data sets.

Algorithm ComputeC(t)
// Initial invocation is $ComputeC(root)$.
// The C array and $level$ are initialized to 0 prior
// to initial invocation.
// Return height of tree rooted at node t.
{
  if (t! = **null**) {
    $level++$;
    $leftHeight = ComputeC(t.leftChild)$;
    $rightHeight = ComputeC(t.rightChild)$;
    $level--$;
    $height = \max\{leftHeight, rightHeight\} + 1$;
    $C[level]$ += $2^{height+1}$;
    return $height$;
    }
  else return -1;
}

Figure 3. Algorithm to compute C using Equation 8.

Once we have determined the C values using Algorithm $ComputeC$ (Figure 3), we may determine $Opt(root,2)$ and the optimal stride for the root in an additional $O(height(root))$ time using Equation 7. If the optimal stride for the root is s, then the second expansion level is level s (unless, $s = 1 + height(root)$, in which case there isn't a second expansion level). The stride for each node at level s is one plus the height of the subtree rooted at that node. The height of the subtree rooted at each node was computed by Algorithm $ComputeC$, and so the strides for the nodes at the second expansion level are easily determined.

2.2.3 Faster $k = 3$ Algorithm

Using the algorithm of Section 2.2.1 may be determine the optimal 3-VST for a set of n prefixes in $O(mW)$ (equal to $O(nW)$ for practical prefix sets) time, where m is the number of nodes in the 1-bit trie and W is the length of the longest prefix. In this section, we develop a simpler and faster $O(mW)$ algorithm for this task. On practical prefix sets, the algorithm of this section runs in $O(m) = O(n)$ time.

From Equation 1, we see that the cost, $Opt(root,3)$ of the best 3-VST is

$$Opt(root,3) = min_{s\in\{1\ldots1+height(root)\}}\{2^s$$

$$+ \sum_{M \in D_s(root)} Opt(M,2)\}$$

$$= min_{s \in \{1 \ldots 1+height(root)\}}\{2^s + T(s)\} \quad (9)$$

where

$$T(s) = \sum_{M \in D_s(root)} Opt(M,2) \quad (10)$$

Figure 4 gives our algorithm to compute $T(s)$, $1 \leq s \leq 1 + height(root)$. The computation of $Opt(M,2)$ is done using Equations 7 and 8. In Algorithm $ComputeT$ (Figure 4), the method $allocate$ allocates a one-dimensional array that is to be used to compute the C values for a subtree. The allocated array is initialized to zeroes; it has positions 0 through W, where W is the length of the longest prefix (W also is $1 + height(root)$); and when computing the C values for a subtree whose root is at level j, only positions j through W of the allocated array may be modified. The method $deallocate$ frees a C array previously allocated.

The complexity of Algorithm $ComputeT$ is readily seen to be $O(mW)$. Once the T values have been computed using Algorithm $ComputeT$, we may determine $Opt(root, 3)$ and the stride of the root of the optimal 3-VST in an additional $O(W)$ time. The strides of the nodes at the remaining expansion levels of the optimal 3-VST may be determined from the $t.stride$ and subtree height values computed by Algorithm $ComputeT$ in $O(m)$ time. So the total time needed to determine the best 3-VST is $O(mW)$.

When the difference between the heights of the left and right subtrees of nodes in the 1-bit trie is bounded by some constant d, the complexity of Algorithm $ComputeT$ is $O(m)$. We use an amortization scheme to prove this. First, note that, exclusive of the recursive calls, the work done by Algorithm $ComputeT$ for each invocation is $O(height(t))$. For simplicity, assume that this work is exactly $height(t) + 1$ (the 1 is for the work done outside the **for** loop of $ComputeT$). Each active C array will maintain a credit that is at least equal to the height of the subtree it is associated with. When a C array is allocated, it has no credit associated with it. Each node in the 1-bit trie begins with a credit of 2. When $t = N$, 1 unit of the credits on N is used to pay for the work done outside of the **for** loop. The remaining unit is given to the C array $leftC$. The cost of the **for** loop is paid for by the credits associated with $rightC$. These credits may fall short by at most $d + 1$, because the height of the left subtree of N may be up to d more than the height of N's right subtree. Adding together the initial credits on the nodes and the maximum total shortfall, we see that $m(2 + d + 1)$ credits are enough to pay for all of the work. So, the complexity of $ComputeT$ is $O(md) = O(m)$ (because d is assumed to be a constant). In practice, we expect that the 1-bit tries for router prefixes will not be too skewed

```
Algorithm ComputeT(t)
// Initial invocation is ComputeT(root).
// The T array and level are initialized to 0
// prior to initial invocation.
// Return cost of best 2-VST for subtree rooted at node t
// and height of this subtree.
{
    if (t! = null) {
        level++;
        // compute C values and heights for left and right subtrees of t
        (leftC, leftHeight) = ComputeT(t.leftChild);
        (rightC, rightHeight) = ComputeT(t.rightChild);
        level--;
         // compute C values and height for t as well as
        // bestT = Opt(t, 2) and t.stride = stride of node t
        // in this best 2-VST rooted at t.
        height = max{leftHeight, rightHeight} + 1;
        bestT = leftC[level] = 2^(height+1);
        t.stride = height + 1;
        for (int i = 1; i <= height; i++) {
            leftC[level + i] += rightC[level + i];
            if (2^i + leftC[level + i] < t.bestT) {
                bestT = 2^i + leftC[level + i];
                t.stride = i;
            }}
        T[level] += bestT;
        deallocate(rightC);
        return (leftC, height);
        }
     else {//t is null
        allocate(C);
        return (C, -1);
        }
}
```

Figure 4. Algorithm to compute T using Equation 10.

k	Paix		Pb		MaeWest	
	No	Yes	No	Yes	No	Yes
2	2528	1722	1806	1041	1754	949
3	1080	907	677	496	619	443
4	845	749	489	397	441	351
5	780	706	440	370	393	327
6	763	695	426	361	379	319
7	759	692	422	358	376	316

Table 2. Memory required (in KBytes) by best k-VST

and that the difference between the heights of the left and right subtrees will, in fact, be quite small. Therefore, in practice, we expect $ComputeT$ to run in $O(m) = O(n)$ time.

3 Experimental Results

We programmed our dynamic programming algorithms in C and compared their performance against that of the C codes for the algorithms of Srinivasan and Varghese [17]. All codes were compiled using the `gcc` compiler and optimization level `O2`. The codes were run on a SUN Ultra Enterprise 4000/5000 computer. For test data, we used the five IPv4 prefix databases of Table 1.

Table 2 shows the memory required by the best k-level VST for the three databases of Table 1. The columns labeled "Yes" give the memory required when the VST is permitted to have Butler nodes [8]. This capability refers to the replacing of subtries with three or fewer prefixes by a single node that contains these prefixes [8]. The columns labeled "No" refer to the case when Butler nodes are not permitted (i.e., the case discussed in this paper). As can be seen, the Butler node provision has far more impact when k is small than when k is large. In fact, when $k = 2$ the Butler node provision reduces the memory required by the best VST by almost 50%. However, when $k = 7$, the reduction in memory resulting from the use of Butler nodes versus not using them results in less than a 20% reduction in memory requirement.

For the run time comparison of the VST algorithms, we implemented three versions of our VST algorithm of Section 2.2.1. None of these versions permitted the use of Butler nodes. The first version, called the $O(nk + W^2k)$ Static Memory Implementation, is the $O(nk + W^2k)$ memory implementation described in Section 2.2.1. The $O(W^2k)$ memory required by this implementation for the cost arrays is allocated at compile time. During execution, memory segements from this preallocated $O(W^2k)$ memory are allocated to nodes, as needed, for their cost arrays. The second version, called the $O(nWk)$ Dynamic Memory Implementation, dynamically allocates a cost array to each node of the 1-bit trie nodes using C's `malloc` method. Neither the first nor second implementations employ the fast algorithms of Sections 2.2.2 and 2.2.3. Table 3 gives the run time for these two implementations.

k	Paix		Pb		MaeWest	
	S	D	S	D	S	D
2	290	500	150	280	150	260
3	360	790	190	460	180	430
4	430	900	210	520	220	430
5	490	1140	260	610	240	570
6	530	1170	290	670	270	570
7	590	1390	330	780	300	690

S = $O(nk + W^2k)$ Static Memory Implementation
D = $O(nWk)$ Dynamic Memory Implementation

Table 3. Execution times (in msec) for first two implementations of our VST algorithm

k	Paix	Pb	MaeWest
2	70	30	30
3	210	100	90
4	550	290	270
5	640	350	370
6	740	430	390
7	920	530	450

Table 4. Execution times (in msec) for third implementation of our VST algorithm

The third implementation of our VST algorithm uses the faster $k = 2$ and $k = 3$ algorithms of Section 2.2.2 and 2.2.3 and also uses $O(nWk)$ memory. The $O(nWk)$ memory is allocated in one large block making a single call to `malloc`. Following this, the large allocated block of memory is partitioned into cost arrays for the 1-bit trie nodes by our program. The run time for the third implementation is given in Table 4. The run times for all three of our implementations is plotted in Figure 5. Notice that this third implementation is significantly faster than our other $O(nWk)$ memory implementation. Note also that this third implementation is also faster than the $O(nk + W^2k)$ memory implementation for the cases $k = 2$ and $k = 3$ (this is because, in our third implementation, these cases use the faster algorithms of Sections 2.2.2 and 2.2.3).

To compare the run time performance of our algorithm with that of [17], we use the times for implementation 3 when $k = 2$ or $k = 3$ and the times for implementation 1

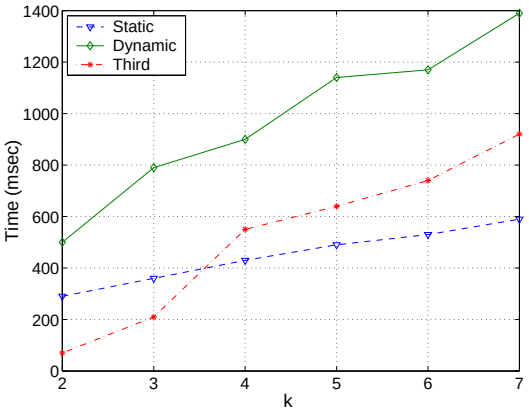


Figure 5. Execution times (in msec) for our three VST implementations

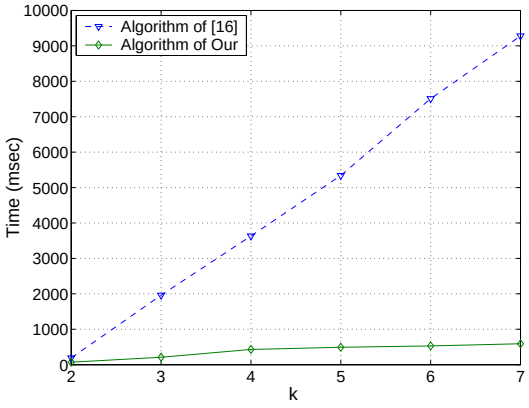


Figure 6. Execution times (in msec) for our best VST implementation and the VST algorithm of [17]

	Paix		Pb		MaeWest	
k	[17]	Our	[17]	Our	[17]	Our
2	190	70	130	30	50	30
3	1960	210	1230	100	360	90
4	3630	430	2330	210	700	220
5	5340	490	3440	260	1030	240
6	7510	530	4550	290	1340	270
7	9280	590	5650	330	1650	300

Table 5. Execution times (in msec) for our best VST implementation and the VST algorithm of [17]

k	Paix	Pb	MaeWest
2	290	170	100
3	340	210	170
4	410	250	220
5	510	290	230
6	470	270	280
7	500	300	270

Table 6. Time (in msec) to construct optimal VST from optimal stride data

when $k > 3$. That is, we compare our best times with the times for the algorithm of [17]. The times for the algorithm of [17] were obtained using their code and running it with the Butler node option off. The run times are shown in Table 5 and these times are plotted in Figure 6. For our largest database, Paix, our new algorithm takes less than half the time taken by the algorithm of [17] when $k = 2$ and less than one-fifteenth the time when $k = 7$. Speedups greater than 17 were observed for some database and k combinations.

The times reported in Tables 3–5 are only the times needed to determine the optimal strides for a given 1-bit trie. Once these strides have been determined, it is necessary to actually construct the optimal VST. Table 6 shows the time required to construct the optimal VST once the optimal strides are known. For our databases, when $k > 3$, the VST construction time is comparable to the time required to compute the optimal strides using our best optimal stride computation implementation. When $k \leq 3$, the VST construction time exceeds the time needed to determine the optimal strides.

The primary operation performed on an optimal VST is a lookup or search in which we begin with a destination address and find the longest prefix that matches this destination address. To determine the average lookup/search time, we searched for as many addresses as there are prefixes in a database. The search addresses were obtained by using the 32-bit expansion available in the database for all prefixes in the database. Table 7 and Figure 7 show the average time to perform a lookup/search. As expected, the average search time increases monotonically with k. The search time for a 2-VST is about 60% that for a 7-VST.

4 Conclusions

We have developed a faster algorithm, to compute the optimal strides for variable stride tries, than the one proposed in [17]. Our algorithm is faster by a factor of between 2 and 17. We expect these speedup factors will be larger for

IPv6 databases.

References

[1] A. Bremler-Barr, Y. Afek, and S. Har-Peled, Routing with a clue, *ACM SIGCOMM* 1999, 203-214.

[2] G. Chandranmenon and G. Varghese, Trading packet headers for packet processing, *IEEE Transactions on Networking*, 1996.

[3] G. Cheung and S. McCanne, Optimal routing table design for IP address lookups under memory constraints, *IEEE INFOCOMM*, 1999.

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, ACM SIGCOMM, 1997, 3-14.

[5] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 1996, 86-97.

[6] M. Gray, Internet growth summary, http://www.mit.edu/people/mkgray/net/internet-growth-sumary.html, 1996.

[7] E. Horowitz, S.Sahni, and D. Mehta, Fundamentals of Data Structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[8] B. Lampson, V. Srinivasan, and G. Varghese, IP Lookup using Multi-way and Multicolumn Search, *IEEE Infocom 98*, 1998.

[9] A. McAuley and P. Francis, Fast routing table lookups using CAMs, IEEE INFOCOM, 1382-1391, 1993.

[10] Merit, Ipma statistics, http://nic.merit.edu/ipma, (snapshot on Sep. 13, 2000), 2000.

[11] D. Milojicic, Trend Wars: Internet Technology, http://www.computer.org/concurrency/articles/trendwars_200_1.htm, 2000.

[12] P. Newman, G. Minshall, and L. Huston, IP switching and gigabit routers, *IEEE Communications Magazine*, Jan., 1997.

[13] S. Nilsson and G. Karlsson, Fast address look-up for Internet routers, *IEEE Broadband Communications*, 1998.

[14] S. Sahni, Data Structures, Algorithms, and Applications in Java, McGraw-Hill, 2000.

k	Paix	Pb	MaeWest
2	486	424	426
3	598	467	494
4	659	569	566
5	757	628	638
6	841	665	699
7	918	730	719

Table 7. Search time (in nsec) in optimal VST

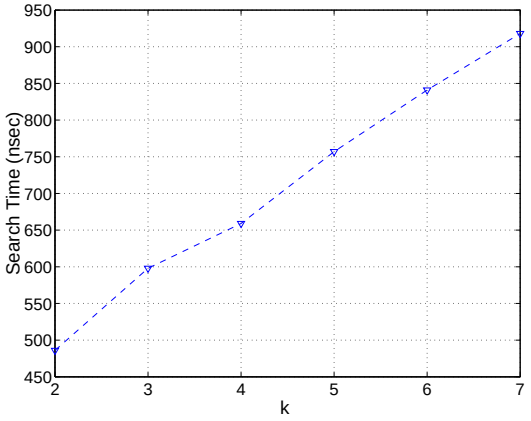


Figure 7. Search time (in nsec) in optimal VST

[15] S. Sahni and K. Kim, Efficient Construction Of Fixed-Stride Multibit Tries For IP Lookup. *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2001.

[16] K. Sklower, A tree-based routing table for Berkeley Unix, Technical Report, University of California, Berkeley, 1993.

[17] V. Srinivasan and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion", ACM Transactions on Computer Systems, Feb:1-40, 1999.

[18] A. Tammel, How to survive as an ISP, *Networld Interop*, 1997.

[19] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, 25-36, 1997.