Efficient Construction Of Fixed-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 [16] have proposed the use of multibit tries to represent routing tables used for Internet (IP) address lookups. They propose an $O(k*W^2)$ time dynamic programming algorithm to determine the strides of an optimal k-level multibit fixed-stride trie when the longest prefix in the routing table has length W. We improve on this algorithm by providing an alternative dynamic programming formulation. While the asymptotic complexity of the resulting algorithm for fixed-stride tries is the same as that of the algorithm of [16], experiments using real IPv4 routing table data indicate that our algorithm runs 2 to 4 times as fast.

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

1 Introduction

With the doubling of Internet traffic every three months [17] 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 [15], 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 [16] 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

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

with a predetermined number of memory accesses per lookup) may be constructed for any prefix set. Srinivasan and Vargese [16] develop a dynamic programming algorithms to obtain space optimal fixed-stride and variable-stride tries of a given height.

Waldvogel et al. [18] 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 [16]. In particular, we develop a new dynamic programming formulation for the construction of space optimal fixed-stride tries of a predetermined height. Our algorithm has the same asymptotic complexity as does the corresponding algorithm of [16]. However, our algorithm runs about 2 to 4 times as fast on real IPv4 router prefix sets.

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

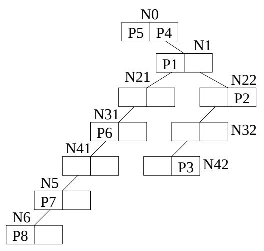
2 Construction Of Multibit Tries

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 [16], and Figure 1(b) shows the corresponding 1-bit trie. The prefixes in Figure 1(a) are numbered and ordered as in [16]. 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.

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

(a) 8-prefix example of [16]



(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 five IPv4 backbone router prefix sets. For our five 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.

2.2 Fixed-Stride Tries

2.2.1 Definition

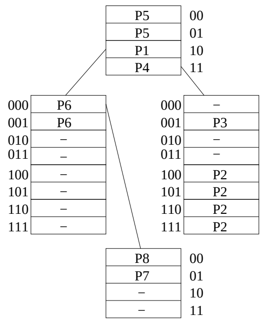
Srinivasan and Varghese [16] have proposed the use of fixed-stride tries to enable fast identification of the longest matching prefix in a router table. The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s child fields (corresponding to the 2^s possible values for the s bits that are used) and 2^s data fields. Such a node requires 2^s memory units. In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides.

Database	Number of prefixes	Number of nodes
Paix	85682	173012
Pb	35151	91718
MaeWest	30599	81104
Aads	26970	74290
MaeEast	22630	65862

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 $i + 1$.

Expanded prefixes (3 levels)
00* (P5a)
01* (P5b)
10* (P1)
11* (P4)
11100* (P2a)
11101* (P2b)
11110* (P2c)
11111* (P2d)
11001* (P3)
10000* (P6a)
10001* (P6b)
1000001* (P7)
1000000* (P8)

(a) Expanded prefixes



(b) Corresponding fixed stride trie

Figure 2. Prefix expansion and fixed stride trie

Suppose we wish to represent the prefixes of Figure 1(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level one nodes store prefixes whose length is 5 (2 + 3); and level three nodes store prefixes whose length is 7 (2 + 3 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length. For example, P5 = 0* is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, P4 = 1* is expanded to P4a = 10* and P4b = 11*. However, P1 = 10* is to be chosen over P4a = 10*, because P1 is a longer match than P4. So, P4a is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 2(a) shows the prefixes that result when we expand the prefixes of Figure 1 to lengths 2, 5, and 7. Figure 2(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.

Since the trie of Figure 2(b) can be searched with at most 3 memory references, it represents a time performance improvement over the 1-bit trie of Figure 1(b), which requires up to 7 memory references to perform a search. However, the space requirements of the FST of Figure 2(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 8 fields or 4 units; the two level 1 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units.

We may represent the prefixes of Figure 1(a) using a one-level trie whose root has a stride of 7. Using such a trie, searches could be performed making a single memory access. However, the one-level trie would require $2^7 = 128$ memory units.

2.2.2 Construction Of Optimal Fixed-Stride Tries

In the *fixed-stride trie optimization* (FSTO) problem, we are given a set P of prefixes and an integer k. We are to select the strides for a k-level FST in such a manner that the k-level FST for the given prefixes uses the smallest amount of memory.

For some P, a k-level FST may actually require more space than a $(k-1)$-level FST. For example, when P = 00*, 01*, 10*, 11*, the unique 1-level FST for P requires 4 memory units while the unique 2-level FST (which is actually the 1-bit trie for P) requires 6 memory units. Since the search time for a $(k-1)$-level FST is less than that for a k-level tree, we would actually prefer $(k-1)$-level FSTs that take less (or even equal) memory over k-level FSTs. Therefore, in practice, we are really interested in determining the best FST that uses at most k levels (rather than exactly k levels). The *modified* MSTO problem (MFSTO) is to determine the best FST that uses at most k levels for the given prefix set P.

Let O be the 1-bit trie for the given set of prefixes, and let F be any k-level FST for this prefix set. Let s_0, ..., s_{k-1} be the strides for F. We shall say that level 0 of F covers levels $0, ..., s_0 - 1$ of O, and that level j, $0 < j < k$ of F covers levels $a, ..., b$ of O, where $a = \sum_0^{j-1} s_q$ and $b = \sum_0^j s_q - 1$. So, level 0 of the FST of Figure 2(b) covers levels 0 and 1 of the 1-bit trie of

Figure 1(b). Level 1 of this FST covers levels 2, 3, and 4 of the 1-bit trie of Figure 1(b); and level 2 of this FST covers levels 5 and 6 of the 1-bit trie. We shall refer to levels $e_u = \sum_0^u s_q$, $0 \le u < k$ as the *expansion levels* of O. The expansion levels defined by the FST of Figure 2(b) are 0, 2, and 5.

Let $nodes(i)$ be the number of nodes at level i of the 1-bit trie O. For the 1-bit trie of Figure 1(a), $nodes(0:6) = [1,1,2,2,2,1,1]$. The memory required by F is $\sum_0^{k-1} nodes(e_q) * 2^{s_q}$. For example, the memory required by the FST of Figure 2(b) is $nodes(0) * 2^2 + nodes(2) * 2^3 + nodes(5) * 2^2 = 24$.

Let $T(j,r)$, $r \le j+1$, be the cost (i.e., memory requirement) of the best way to cover levels 0 through j of O using exactly r expansion levels. When the maximum prefix length is W, $T(W-1,k)$ is the cost of the best k-level FST for the given set of prefixes. Srinivasan and Varghese [16] have obtained the following dynamic programming recurrence for T:

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

$$T(j,1) = 2^{j+1} \quad (2)$$

The rationale for Equation 1 is that the best way to cover levels 0 through j of O using exactly r expansion levels, $r > 1$, must have its last expansion level at level $m+1$ of O, where m must be at least $r-2$ (as otherwise, we do not have enough levels between levels 0 and m of O to select the remaining $r-1$ expansion levels) and at most $j-1$ (because the last expansion level is $\le j$). When the last expansion level is level $m+1$, the stride for this level is $j-m$, and the number of nodes at this expansion level is $nodes(m+1)$. For optimality, levels 0 through m of O must be covered in the best possible way using exactly $r-1$ expansion levels.

As noted by Srinivasan and Varghese [16], using the above recurrence, we may determine $T(W-1,k)$ in $O(kW^2)$ time (excluding the time needed to compute O from the given prefix set and determine $nodes()$). The strides for the optimal k-level FST can be obtained in an additional $O(k)$ time. Since, Equation 1 also may be used to compute $T(W-1,q)$ for all $q \le k$ in $O(kW^2)$ time, we can actually solve the MFSTO problem in the same asymptotic complexity as required for the FSTO problem.

We can reduce the time needed to solve the MFSTO problem by modifying the definition of T. The modified function is C, where $C(j,r)$ is the cost of the best FST that uses *at most* r expansion levels. It is easy to see that $C(j,r) \le C(j,r-1)$, $r > 1$. A simple dynamic programming recurrence for C is:

$$C(j,r) = \min_{m \in \{-1..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\}, j \ge 0, r > 1 \quad (3)$$

$$C(-1,r) = 0 \text{ and } C(j,1) = 2^{j+1}, j \ge 0 \quad (4)$$

To see the correctness of Equations 3 and 4, note that when $j \ge 0$, there must be at least one expansion level. If $r = 1$, then there is eactly one expansion level and the cost is 2^{j+1}. If $r > 1$, the last expansion level in the best FST could be at any of the levels 0 through j. Let $m+1$ be this last expansion level. The cost of the covering is $C(m,r-1) + nodes(m+1) * 2^{j-m}$. When $j = -1$, no levels of the 1-bit trie remain to be covered. Therefore, $C(-1,r) = 0$.

We may obtain an alternative recurrence for $C(j,r)$ in which the range of m on the right side is $r-2..j-1$ rather than $-1..j-1$. First, we obtain the following dynamic programming recurrence for C:

$$C(j,r) = \min\{C(j,r-1), T(j,r)\}, \quad r > 1 \quad (5)$$

$$C(j,1) = 2^{j+1} \quad (6)$$

The rationale for Equation 5 is that the best FST that uses at most r expansion levels either uses at most $r-1$ levels or uses exactly r levels. When at most $r-1$ levels are used, the cost is $C(j,r-1)$, and when exactly r levels are used, the cost is $T(j,r)$, which is defined by Equation 1.

Let $U(j,r)$ be as defined below:

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

From Equations 1 and 5 we obtain:

$$C(j,r) = \min\{C(j,r-1), U(j,r)\} \quad (7)$$

To see the correctness of Equation 7, note that for all j and r such that $r \le j+1$, $T(j,r) \ge C(j,r)$, Furthermore,

$$\begin{aligned} \min_{m \in \{r-2..j-1\}} & \{ T(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ & \ge \min_{m \in \{r-2..j-1\}} \{C(m,r-1) + nodes(m+1) * 2^{j-m}\} \\ & = U(j,r) \end{aligned} \quad (8)$$

Therefore, when $C(j,r-1) \le U(j,r)$, Equations 5 and 7 compute the same value for $C(j,r)$. When

$C(j, r-1) > U(j, r)$, it appears from Equation 8 that Equation 7 may compute a smaller $C(j, r)$ than is computed by Equation 5. However, this is impossible, because

$$\begin{aligned} C(j,r) &= \min_{m \in \{-1..j-1\}} \{C(m, r-1) + \\ &\qquad nodes(m+1) * 2^{j-m}\} \\ &\leq \min_{m \in \{r-2..j-1\}} \{C(m, r-1) + \\ &\qquad nodes(m+1) * 2^{j-m}\} \end{aligned}$$

where $C(-1, r) = 0$. Therefore, the $C(j, r)$s computed by Equations 5 and 7 are equal.

In the remainder of this section, we use Equations 3 and 4 for C. The range for m (in Equation 3) may be restricted to a range that is (often) considerably smaller than $r-2..j-1$. To obtain this narrower search range, we first establish a few properties of 1-bit tries and their corresponding optimal FSTs.

Lemma 1 *For every 1-bit trie O, (a)* $nodes(i) \leq 2^i$, $i \geq 0$ *and (b)* $nodes(i+j) \leq 2^j nodes(i)$, $j \geq 0$, $i \geq 0$.

Proof Follows from the fact that a 1-bit trie is a binary tree. ■

Let $M(j, r)$, $r > 1$, be the smallest m that minimizes

$$C(m, r-1) + nodes(m+1) * 2^{j-m},$$

in Equation 3.

Lemma 2 $\forall (j \geq 0, r > 1)[M(j+1, r) \geq M(j, r)]$.

Proof Omitted. ■

Lemma 3 $\forall (j \geq 0, r > 0)[C(j, r) < C(j+1, r)]$.

Proof ■

The next few lemmas use the function Δ, which is defined as $\Delta(j, r) = C(j, r-1) - C(j, r)$. Since, $C(j, r) \leq C(j, r-1)$, $\Delta(j, r) \geq 0$ for all $j \geq 0$ and all $r \geq 2$.

Lemma 4 $\forall (j \geq 0)[\Delta(j, 2) \leq \Delta(j+1, 2)]$.

Proof Omitted. ■

Lemma 5 $\forall (j \geq 0, k > 2)[\Delta(j, k-1) \leq \Delta(j+1, k-1)] \Longrightarrow \forall (j \geq 0, k > 2)[\Delta(j, k) \leq \Delta(j+1, k)]$.

Proof Omitted. ■

Lemma 6 $\forall (j \geq 0, k \geq 2)[\Delta(j, k) \leq \Delta(j+1, k)]$.

Proof Follows from Lemmas 4 and 5. ■

Lemma 7 *Let* $k > 2$. $\forall (j \geq 0)[\Delta(j, k-1) \leq \Delta(j+1, k-1)] \Longrightarrow \forall (j \geq 0)[M(j, k) \geq M(j, k-1)]$.

Proof Omitted. ■

Lemma 8 $\forall (j \geq 0, k > 2)[M(j, k) \geq M(j, k-1)]$.

Proof Follows from Lemmas 6 and 7. ■

Theorem 1 $\forall (j \geq 0, k > 2)[M(j, k) \geq max\{M(j-1, k), M(j, k-1)\}]$.

Proof Follows from Lemmas 2 and 8. ■

Note 1 *From Lemma 6, it follows that whenever* $\Delta(j, k) > 0$, $\Delta(q, k) > 0$, $\forall q > j$.

Theorem 1 leads to Algorithm *FixedStrides* (Figure 3), which computes $C(W-1, k)$. The complexity of this algorithm is $O(kW^2)$. Using the computed M values, the strides for the OFST that uses at most k expansion levels may be determined in an additional $O(k)$ time. Although our algorithm has the same asymptotic complexity as does the algorithm of Srinivasan and Varghese [16], experiments conducted by us using real prefix sets indicate that our algorithm runs 2 to 4 times as fast.

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 [16]. 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.

Figure 4 shows the memory required by the best k-level FST for each of the five databases of Table 1. Note that the y-axis of Figure 4 uses a semilog scale. The k values used by us range from a low of 2 to a high of 7 (corresponding to a lookup performance of at most 2 memory accesses per lookup to at most 7 memory accesses per lookup). As was the case with the data sets used in [16], using a larger number of levels does not increase the required memory. We note that for $k = 11$ and 12, [16] reports no decrease in memory

```
Algorithm FixedStrides($W$, $k$)
// $W$ is length of longest prefix.
// $k$ is maximum number of expansion levels desired.
// Return $C(W-1,k)$ and compute $M(*,*)$.
{
    for ($j=0; j<W; j++$){
        $C(j,1) := 2^{j+1}$;
        $M(j,1) := -1$;}
    for ($r=1; r<k; r++$)
        $C(-1,r) := 0$;
    for ($r=2; r\le k; r++$)
        for ($j=r-1; j<W; j++$){
        // Compute $C(j,r)$.
            $minJ := max(M(j-1,r), M(j,r-1))$;
            $minCost := C(j,r-1)$;
            $minL := M(j,r-1)$;
            for ($m=minJ; m<j; m++$){
                $cost := C(m,j-1) + nodes(m+1) * 2^{j-m}$;
                if ($cost < minCost$) then
                        {$minCost := cost$; $minL := m$;}}
            $C(j,r) := minCost$; $M(j,r) := minL$;}
    return $C(W-1,k)$;
}
```

Figure 3. Algorithm for fixed-stride tries.

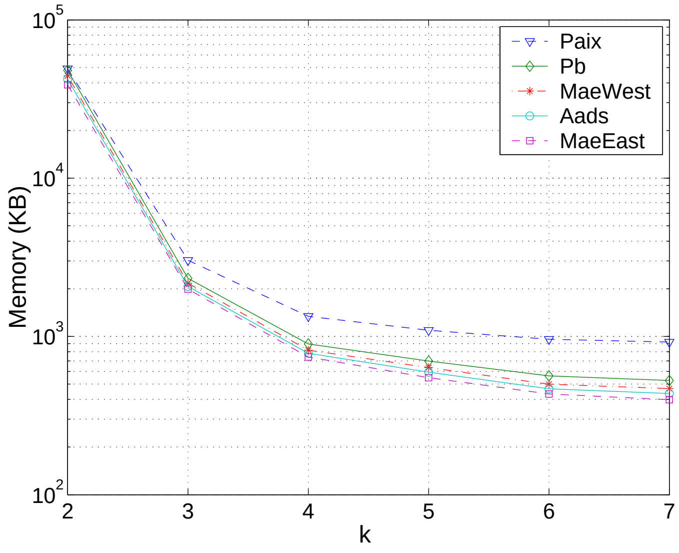


Figure 4. Memory required (in KBytes) by best k-level FST

	Paix		Pb		MaeWest		Aads		MaeEast	
k	[16]	Our	[16]	Our	[16]	Our	[16]	Our	[16]	Our
2	39	21	41	21	39	21	37	20	37	21
3	85	30	81	30	84	31	74	31	96	31
4	123	39	124	40	128	38	122	40	130	40
5	174	46	174	48	147	46	161	45	164	46
6	194	53	201	54	190	55	194	54	190	53
7	246	62	241	63	221	63	264	62	220	62

Table 2. Execution time (in μ sec) for FST algorithms

required for three of their data sets. We did not try such large k values for our data sets.

Table 3 and Figure 5 show the time taken by both our algorithm and that of [16] (we are grateful to Dr. Srinivasan for making his code available to us) to determine the optimal strides of the best FST that has at most k levels. As expected, the run time of the algorithm of [16] is quite insensitive to the number of prefixes in the database. Although the run time of our algorithm is independent of the number of prefixes, the run time does depend on the values of $nodes(*)$ as these values determine $M(*,*)$ and hence determine $minJ$ in Figure 3. As indicated by the graph of Figure 5, the run time for our algorithm varies only slightly with the database. As can be seen, our algorithm provides a speedup of between ≈2 and ≈4 compared to that of [16].

4 Conclusions

We have developed a faster algorithm to compute the optimal strides for fixed-stride tries, than the one proposed in [16]. For IPv4 prefix databases, our algorithm is faster by a factor of between 2 and 4. 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.

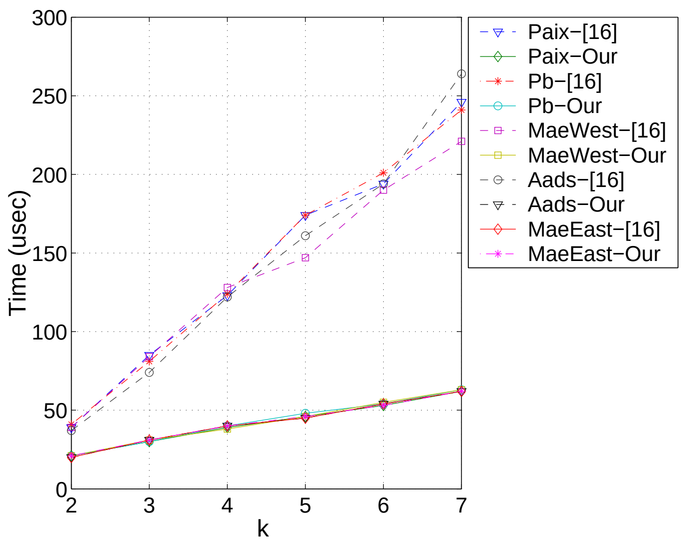


Figure 5. Execution time (in μ sec) for FST algorithms

[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.

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

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

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

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