# Efficient Construction Of 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 $W$. They also 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. We improve on these algorithms by providing alternative dynamic programming formulations for both fixed- and variable-stride tries. 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. The complexity of our algorithm for variable-stride tries is $O(n * W * k)$, an improvement by a factor of $W$ over the corresponding algorithm of [16]. 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 [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.

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

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 with a predetermined number of memory accesses per lookup) may be constructed for any prefix set. Srinivasan and Vargese [16] develop 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 new dynamic programming formulations for the construction of space optimal tries of a predetermined height. Our algorithm for optimal fixed-stride tries 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. Our variable-stride algorithm is asymptotically faster, by a factor of $W$, than the corresponding algorithm of [16]. Experiments with IPv4 prefix sets indicate that our variable stride-algorithm is 2 to 17 times as fast as that of [16].

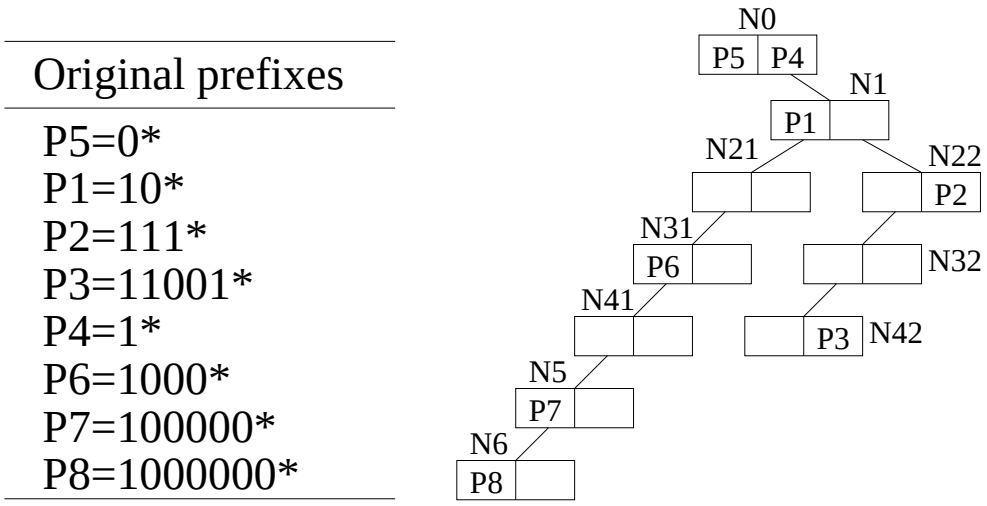
In Section 2, we develop our new dynamic programming formulations, and in Section 3, we present our experimental results.

# 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.

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, and Figure 2 gives a more detailed characterization of the prefixes in the




(a) 8-prefix example of [16]

(b) Corresponding 1-bit trie

Figure 1: Prefixes and corresponding 1-bit trie

| Database | Number of prefixes | Number of 16-bit prefixes | Number of 24-bit prefixes | Number of nodes |
|---|---|---|---|---|
| Paix | 85682 | 3051 | 37413 | 173012 |
| Pb | 35151 | 1705 | 15516 | 91718 |
| MaeWest | 30599 | 1625 | 13137 | 81104 |
| Aads | 26970 | 1470 | 11711 | 74290 |
| MaeEast | 22630 | 1217 | 9493 | 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 $i + 1$.

largest of these five databases, Paix. 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 (Figure 2).

## 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

| Level | Number of prefixes | Number of nodes | Level | Number of prefixes | Number of nodes |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 16 | 169 | 5117 |
| 2 | 0 | 2 | 17 | 430 | 8245 |
| 2 | 0 | 4 | 18 | 2238 | 12634 |
| 3 | 0 | 7 | 19 | 1292 | 15504 |
| 4 | 0 | 11 | 20 | 1309 | 20557 |
| 5 | 0 | 20 | 21 | 2465 | 26811 |
| 6 | 0 | 36 | 22 | 3690 | 32476 |
| 7 | 6 | 62 | 23 | 37413 | 37467 |
| 8 | 0 | 93 | 24 | 11 | 54 |
| 9 | 0 | 169 | 25 | 25 | 44 |
| 10 | 0 | 303 | 26 | 11 | 20 |
| 11 | 0 | 561 | 27 | 4 | 9 |
| 12 | 3 | 1037 | 28 | 3 | 5 |
| 13 | 15 | 1933 | 20 | 1 | 2 |
| 14 | 25 | 3552 | 30 | 0 | 1 |
| 15 | 3051 | 6274 | 31 | 1 | 1 |

Figure 2: Distributions of the prefixes and nodes in the 1-bit trie for Paix

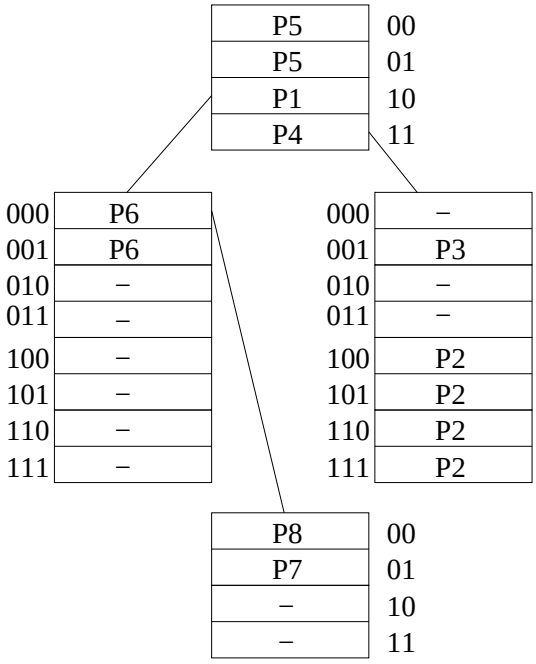
levels may have different strides.

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 3(a) shows the prefixes that result when we expand the prefixes of Figure 1 to lengths 2, 5, and 7. Figure 3(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.

Since the trie of Figure 3(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 3(b) are more than that of

| 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 3: Prefix expansion and fixed stride trie

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 3(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 $e_u = \sum_0^u s_q$, $0 \leq u < k$ as the *expansion levels* of $O$. The expansion levels defined by the FST of Figure 3(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 3(b) is $nodes(0) * 2^2 + nodes(2) * 2^3 + nodes(5) * 2^2 = 24$.

Let $T(j, r)$, $r \leq j+1$, 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 $\leq j$). 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 $q \leq k$ 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 $C(j,r) \leq C(j,r-1)$, $r > 1$. A simple dynamic programming recurrence for $C$ is:

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

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

To see the correctness of Equations 3 and 4, note that when $j \geq 0$, 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 $C$:

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

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

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:

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

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

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

Therefore, when $C(j,r-1) \leq U(j,r)$, 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

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

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** Let $M(j,r)=a$ and $M(j+1,r)=b$.

Suppose $b<a$. Then,

$$\begin{aligned}
C(j,r) &= C(a,r-1)+nodes(a+1)*2^{j-a} \\
&< C(b,r-1)+nodes(b+1)*2^{j-b}
\end{aligned}$$

since, otherwise, $M(j,r) = b$. Furthermore,

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

Therefore,

$$\begin{aligned} & nodes(a+1) * 2^{j-a} + nodes(b+1) * 2^{j+1-b} \\ < \;& nodes(b+1) * 2^{j-b} + nodes(a+1) * 2^{j+1-a} \end{aligned}$$

So,

$$nodes(b+1) * 2^{j-b} < nodes(a+1) * 2^{j-a}$$

Hence,

$$2^{a-b} * nodes(b+1) < nodes(a+1)$$

This contradicts Lemma 1(b). So, $b \geq a$. ■

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

**Proof** The case $r = 1$ follows from $C(j,1) = 2^{j+1}$. So, assume $r > 1$. From the definition of $M(j,r)$, it follows that

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

where $-1 \leq b = M(j,r) \leq j$. When $b < j$, we get

$$\begin{aligned} C(j,r) &\leq C(b,r-1) + nodes(b+1) * 2^{j-b} \\ &< C(b,r-1) + nodes(b+1) * 2^{j+1-b} \\ &= C(j+1,r) \end{aligned}$$

When $b = j$,

$$C(j+1,r) = C(j,r-1) + nodes(j+1) * 2 > C(j,r-1),$$

since $nodes(j+1) > 0$. ■

The next few lemmas use the function $\Delta$, 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** If $C(j,2) = C(j,1)$, there is nothing to prove as $\Delta(j+1,2) \geq 0$. The only other possibility is $C(j,2) < C(j,1)$ (i.e., $\Delta(j,2) > 0$). In this case, the best cover for levels 0 through $j$ uses exactly 2 expansion levels. From the recurrence for $C$ (Equations 3 and 4), it follows that $C(j,1) = 2^{j+1}$, and

$$\begin{aligned} C(j,2) &= C(a,1) + nodes(a+1) * 2^{j-a} \\ &= 2^{a+1} - nodes(a+1) * 2^{j-a}, \end{aligned}$$

for some $a$, $0 \leq a < j$. Therefore,

$$\begin{aligned} \Delta(j,2) &= C(j,1) - C(j,2) \\ &= 2^{j+1} - 2^{a+1} - nodes(a+1) * 2^{j-a}. \end{aligned}$$

From Equations 3 and 4, it follows that

$$\begin{aligned} C(j+1,2) &\leq C(a,1) + nodes(a+1) * 2^{j+1-a} \\ &= 2^{a+1} + nodes(a+1) * 2^{j+1-a}. \end{aligned}$$

Hence,

$$\Delta(j+1,2) \geq 2^{j+2} - 2^{a+1} - nodes(a+1) * 2^{j+1-a}.$$

Therefore,

$$\begin{aligned} \Delta(j+1,2) - \Delta(j,2) &\geq 2^{j+2} - 2^{a+1} - nodes(a+1) * 2^{j+1-a} \\ &\quad -2^{j+1} + 2^{a+1} + nodes(a+1) * 2^{j-a} \\ &= 2^{j+1} - nodes(a+1) * 2^{j-a} \\ &\geq 2^{j+1} - 2^{a+1} * 2^{j-a} \text{ (Lemma 1(a))} \\ &= 0 \end{aligned}$$

■

**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** Assume that $\forall(j \geq 0, k > 2)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$. We shall show that $\forall(j \geq 0, k > 2)[\Delta(j,k) \leq \Delta(j+1,k)]$. Let $M(j,k) = b$ and $M(j+1,k-1) = c$.

**Case 1:** $c \geq b$.

$$\begin{aligned}
\Delta(j,k) &= C(j,k-1) - C(j,k) \\
&= C(j,k-1) - C(b,k-1) - nodes(b+1) * 2^{j-b} \\
&\leq C(b,k-2) + nodes(b+1) * 2^{j-b} \\
&\quad -C(b,k-1) - nodes(b+1) * 2^{j-b} \\
&= \Delta(b,k-1).
\end{aligned}$$

Also,

$$\begin{aligned}
\Delta(j+1,k) &= C(j+1,k-1) - C(j+1,k) \\
&\geq C(c,k-2) + nodes(c+1) * 2^{j+1-c} \\
&\quad -C(c,k-1) - nodes(c+1) * 2^{j+1-c} \\
&= \Delta(c,k-1).
\end{aligned}$$

Since $c \geq b$, $\Delta(b,k-1) \leq \Delta(c,k-1)$. Therefore,

$$\Delta(j+1,k) \geq \Delta(c,k-1) \geq \Delta(b,k-1) \geq \Delta(j,k).$$

**Case 2:** $c < b$.

Let $M(j+1,k) = a$, $M(j,k) = b$, $M(j+1,k-1) = c$, and $M(j,k-1) = d$. From Lemma 2, $a \geq b$ and $c \geq d$. Since $c < b$, $a \geq b > c \geq d$. Also,

$$\begin{aligned}
\Delta(j,k) &= C(j,k-1) - C(j,k) \\
&= [C(d,k-2) + nodes(d+1) * 2^{j-d}] \\
&\quad -[C(b,k-1) + nodes(b+1) * 2^{j-b}]
\end{aligned}$$

and

$$\begin{aligned}
\Delta(j+1,k) &= C(j+1,k-1) - C(j+1,k) \\
&= [C(c,k-2) + nodes(c+1) * 2^{j+1-c}] \\
&\quad -[C(a,k-1) + nodes(a+1) * 2^{j+1-a}].
\end{aligned}$$

Therefore,

$$\Delta\Delta = \Delta(j+1,k) - \Delta(j,k)$$

$$
\begin{aligned}
= \quad & [C(c, k-2) + nodes(c+1) * 2^{j+1-c}] \\
& -[C(d, k-2) + nodes(d+1) * 2^{j-d}] \\
& +[C(b, k-1) + nodes(b+1) * 2^{j-b}] \\
& -[C(a, k-1) + nodes(a+1) * 2^{j+1-a}]. \qquad (9)
\end{aligned}
$$

Since $j > b > c \geq d = M(j, k-1)$,

$$
\begin{aligned}
C(c, k-2) \quad + \quad & nodes(c+1) * 2^{j-c} \\
\geq \quad & C(d, k-2) + nodes(d+1) * 2^{j-d} \qquad (10)
\end{aligned}
$$

Furthermore, since $M(j+1, k) = a \geq b$,

$$
\begin{aligned}
C(b, k-1) \quad + \quad & nodes(b+1) * 2^{j+1-b} \\
\geq \quad & C(a, k-1) + nodes(a+1) * 2^{j+1-a} \qquad (11)
\end{aligned}
$$

Substituting Equations 10 and 11 into Equation 9, we get

$$\Delta\Delta \geq nodes(c+1) * 2^{j-c} - nodes(b+1) * 2^{j-b}.$$

Lemma 1 and $c < b$ imply $nodes(c+1) * 2^{b-c} \geq nodes(b+1)$. Therefore,

$$nodes(c+1) * 2^{j-c} \geq nodes(b+1) * 2^{j-b}.$$

So, $\Delta\Delta \geq 0$. ■

**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** Assume that $\forall(j \geq 0)[\Delta(j, k-1) \leq \Delta(j+1, k-1)]$. Suppose that $M(j, k-1) = a$, $M(j, k) = b$, and $b < a$ for some $j$, $j \geq 0$. From Equation 3, we get

$$
\begin{aligned}
C(j, k) \quad = \quad & C(b, k-1) + nodes(b+1) * 2^{j-b} \\
\leq \quad & C(a, k-1) + nodes(a+1) * 2^{j-a}
\end{aligned}
$$

and

$$\begin{aligned} C(j,k-1) &= C(a,k-2) + nodes(a+1) * 2^{j-a} \\ &< C(b,k-2) + nodes(b+1) * 2^{j-b}. \end{aligned}$$

Hence,

$$C(b,k-1) + C(a,k-2) < C(a,k-1) + C(b,k-2).$$

Therefore,

$$\Delta(a,k-1) < \Delta(b,k-1).$$

However, $b < a$ and $\forall(j \geq 0)[\Delta(j,k-1) \leq \Delta(j+1,k-1)]$ imply that $\Delta(b,k-1) \leq \Delta(a,k-1)$. Since our assumption that $b < a$ leads to a contradiction, it must be that there is no $j \geq 0$ for which $M(j,k-1) = a$, $M(j,k) = b$, and $b < a$. ■

**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 4), 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.

## 2.3 Variable-Stride Tries

### 2.3.1 Definition and Construction

In a *variable-stride trie* (VST) [16], nodes at the same level may have different strides. Figure 5 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

```
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 ≤ 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 4: Algorithm for fixed-stride tries.

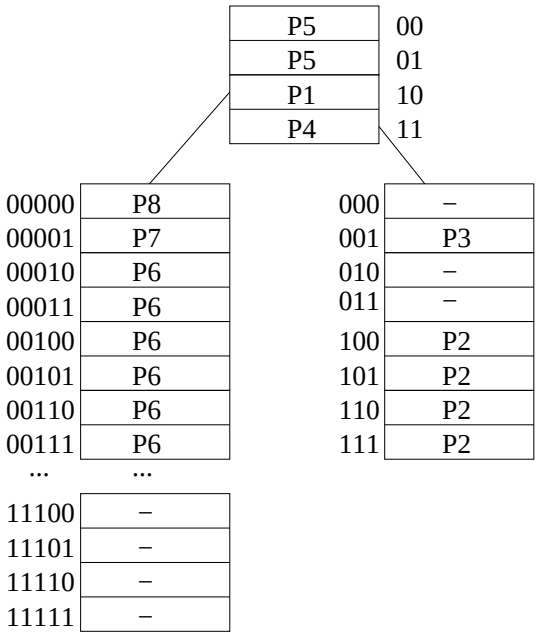



Figure 5: Two-level VST for prefixes of Figure 1(a)

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.

Since FSTs are a special case of VSTs, the memory required by the best VST for a given prefix set $P$ and number of expansion levels $k$ is less than or equal to that required by the best FST for $P$ and $k$. Despite this, FSTs may be preferred in certain router applications "because of their simplicity and slightly faster search time" [16].

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 [16], have obtained the following dynamic programming recurrence for $Opt(N, r)$.

$$Opt(N, r) = \min_{s \in \{1 \ldots 1 + height(N)\}} \{2^s + \sum_{M \in D_s(N)} Opt(M, r-1)\}, \quad r > 1 \tag{12}$$

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 $r = 1$,

$$Opt(N, 1) = 2^{1+height(N)} \tag{13}$$

Equation 13 is equivalent to Equation 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 12 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 [16], describe a way to determine $Opt(R, k)$ using Equations 12 and 13. 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 [16] 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 12 and 13, we obtain:

$$Opt(N,0,r) = min_{s \in \{1 \ldots 1+height(N)\}}\{2^s + Opt(N,s,r-1)\},\ r > 1 \tag{14}$$

and

$$Opt(N,0,1) = 2^{1+height(N)}. \tag{15}$$

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

$$\begin{aligned} Opt(N,s,r) &= \sum_{M \in D_s(N)} Opt(M,r) \\ &= Opt(LeftChild(N), s-1, r) \\ &+ Opt(RightChild(N), s-1, r). \end{aligned} \tag{16}$$

For Equation 16, we need the following initial condition:

$$Opt(null, *, *) = 0 \tag{17}$$

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 14 through 17 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 14 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.

| | |
|---|---|
| $P1 = 0*$ | $0 * (P1)$ |
| $P2 = 1*$ | $1 * (P2)$ |
| $P3 = 11*$ | $101 * (P4)$ |
| $P4 = 101*$ | $110 * (P3)$ |
| $P5 = 10001*$ | $111 * (P3)$ |
| $P6 = 1100*$ | $10001 * (P5)$ |
| $P7 = 110000*$ | $11000 * (P6)$ |
| $P8 = 1100000*$ | $11001 * (P6)$ |
| | $1100000 * (P8)$ |
| | $1100001 * (P7)$ |
| (a) Original prefixes | (b) Expanded prefixes |

Figure 6: A prefix set and its expansion to four lengths

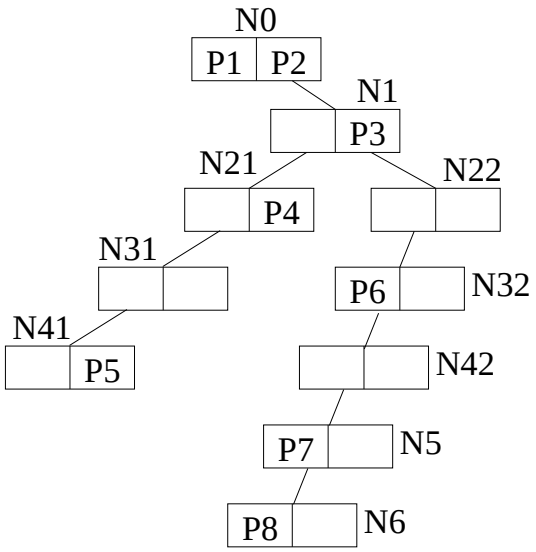



Figure 7: 1-bit trie for prefixes of Figure 6(a)

### 2.3.2 An Example

Figure 6(a) gives a prefix set $P$ that contains 8 prefixes. The length of the longest prefix (P8) is 7. Figure 6(b) gives the prefixes that remain when the prefixes of $P$ are expanded into the lengths 1, 3, 5, and 7. As we shall see, these expanded prefixes correspond to an optimal 4-VST for $P$. Figure 7 gives the 1-bit trie for the prefixes of Figure 6.

To determine the cost, $Opt(N0, 0, 4)$, of the best 4-VST for the prefix set of Figure 6(a), we must compute all the $Opt$ values shown in Figure 8. In this figure $Opt_1$, for example, refers to $Opt(N1, *, *)$ and $Opt_{42}$ refers to $Opt(N42, *, *)$. The $Opt$ arrays shown in Figure 8 are computed in postorder; that is, in the order N41, N31, N21, N6, N5, N42, N32, N22, N1, N0. The $Opt$ values shown in Figure 8 were

| $Opt_0$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 128 | 26 | 20 | 18 |
| 1 | 64 | 18 | 16 | 16 |
| 2 | 40 | 18 | 16 | 16 |
| 3 | 20 | 12 | 12 | 12 |
| 4 | 10 | 8 | 8 | 8 |
| 5 | 4 | 4 | 4 | 4 |
| 6 | 2 | 2 | 2 | 2 |

| $Opt_1$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 64 | 18 | 16 | 16 |
| 1 | 40 | 18 | 16 | 16 |
| 2 | 20 | 12 | 12 | 12 |
| 3 | 10 | 8 | 8 | 8 |
| 4 | 4 | 4 | 4 | 4 |
| 5 | 2 | 2 | 2 | 2 |

| $Opt_{21}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 8 | 6 | 6 | 6 |
| 1 | 4 | 4 | 4 | 4 |
| 2 | 2 | 2 | 2 | 2 |

| $Opt_{22}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 32 | 12 | 10 | 10 |
| 1 | 16 | 8 | 8 | 8 |
| 2 | 8 | 6 | 6 | 6 |
| 3 | 4 | 4 | 4 | 4 |
| 4 | 2 | 2 | 2 | 2 |

| $Opt_{31}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 4 | 4 | 4 | 4 |
| 1 | 2 | 2 | 2 | 2 |

| $Opt_{32}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 16 | 8 | 8 | 8 |
| 1 | 8 | 6 | 6 | 6 |
| 2 | 4 | 4 | 4 | 4 |
| 3 | 2 | 2 | 2 | 2 |

| $Opt_{41}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 2 | 2 | 2 | 2 |

| $Opt_{42}$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 8 | 6 | 6 | 6 |
| 1 | 4 | 4 | 4 | 4 |
| 2 | 2 | 2 | 2 | 2 |

| $Opt_5$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 4 | 4 | 4 | 4 |
| 1 | 2 | 2 | 2 | 2 |

| $Opt_6$ | $k=1$ | 2 | 3 | 4 |
|---|---|---|---|---|
| $s=0$ | 2 | 2 | 2 | 2 |

Figure 8: *Opt* values in the computation of $Opt(N0, 0, 4)$

computed using Equations 14 through 17.

From Figure 8, we determine that the cost of the best 4-VST for the given prefix set is $Opt(N0, 0, 4) = 18$. To construct this best 4-VST, we must determine the strides for all nodes in the best 4-VST. These strides are easily determined if, with each $Opt(*, 0, *)$, we store the $s$ value that minimizes the right side of Equation 14. For $Opt(N0, 0, 4)$, this minimizing $s$ value is 1. This means that the stride for the root of the best 4-VST is 1, its left subtree is empty (because N0 has an empty left subtree), its right subtree is the best 3-VST for the subtree rooted at N1. The minimizing $s$ value for $Opt(N1, 0, 3)$ is 2 (actually, there is a tie between $s = 2$ and $s = 3$; ties may be broken arbitrarily). Therefore, the right child of the root of the best 4-VST has a stride of 2. Its first subtree is the best 2-VST for N31; its second subtree is empty; its third subtree is the best 2-VST for N32; and its fourth subtree is empty. Continuing in this manner, we obtain the 4-VST of Figure 9. The cost of this 4-VST is 18.

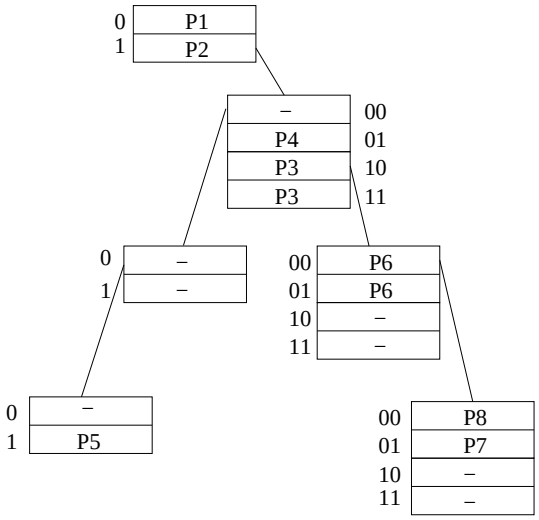



Figure 9: Optimal 4-VST for prefixes of Figure 6(a)

### 2.3.3 Faster $k = 2$ Algorithm

The algorithm of Section 2.3.1 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 12, we see that the cost, $Opt(root, 2)$ of the best 2-VST is

$$\begin{aligned} Opt(root,2) &= min_{s\in\{1\ldots 1+height(root)\}}\{2^s + \sum_{M\in D_s(root)} Opt(M,1)\} \\ &= min_{s\in\{1\ldots 1+height(root)\}}\{2^s + \sum_{M\in D_s(root)} 2^{1+height(M)}\} \quad (18) \\ &= min_{s\in\{1\ldots 1+height(root)\}}\{2^s + C(s)\} \quad (19) \end{aligned}$$

where

$$C(s) = \sum_{M\in D_s(root)} 2^{1+height(M)} \quad (20)$$

We may compute $C(s)$, $1 \leq s \leq 1 + height(root)$, in $O(m)$ time by performing a postorder traversal (see Figure 10) 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 10 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 10: Algorithm to compute $C$ using Equation 20.

Once we have determined the $C$ values using Algorithm *ComputeC* (Figure 10), we may determine $Opt(root, 2)$ and the optimal stride for the root in an additional $O(height(root))$ time using Equation 19. 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.

### 2.3.4 Faster $k = 3$ Algorithm

Using the algorithm of Section 2.3.1 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 12, we see that the cost, $Opt(root, 3)$ of the best 3-VST is

$$\begin{aligned} Opt(root,3) &= min_{s\in\{1\ldots 1+height(root)\}}\{2^s + \sum_{M\in D_s(root)} Opt(M,2)\} \\ &= min_{s\in\{1\ldots 1+height(root)\}}\{2^s + T(s)\} \end{aligned} \quad (21)$$

where

$$T(s) = \sum_{M \in D_s(root)} Opt(M, 2) \tag{22}$$

Figure 11 gives our algorithm to compute $T(s)$, $1 \leq s \leq 1 + height(root)$. The computation of $Opt(M, 2)$ is done using Equations 19 and 20. In Algorithm *ComputeT* (Figure 11), 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 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.

```
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 11: Algorithm to compute $T$ using Equation 22.

| Levels($k$) | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| Paix | 49192 | 3030 | 1340 | 1093 | 960 | 922 |
| pb | 47925 | 2328 | 896 | 699 | 563 | 527 |
| MaeWest | 44338 | 2168 | 819 | 636 | 499 | 468 |
| aads | 42204 | 2070 | 782 | 594 | 467 | 436 |
| MaeEast | 38890 | 1991 | 741 | 549 | 433 | 398 |

Table 2: Memory required (in KBytes) by best $k$-level FST

# 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.

## 3.1 Performance of Fixed-Stride Algorithm

Table 2 and Figure 12 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 12 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 required for three of their data sets. We did not try such large $k$ values for our data sets.

Table 3 and Figure 13 show the time taken by both our algorithm and that of [16] (we are grateful to Dr. Srinivasan for making his fixed- and variable-stride codes 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 4. As indicated by the graph of Figure 13, the run time for our algorithm varies only slightly with the database. As can be seen, our algorithm provides a speedup of between $\approx$2 and $\approx$4 compared to that of [16].

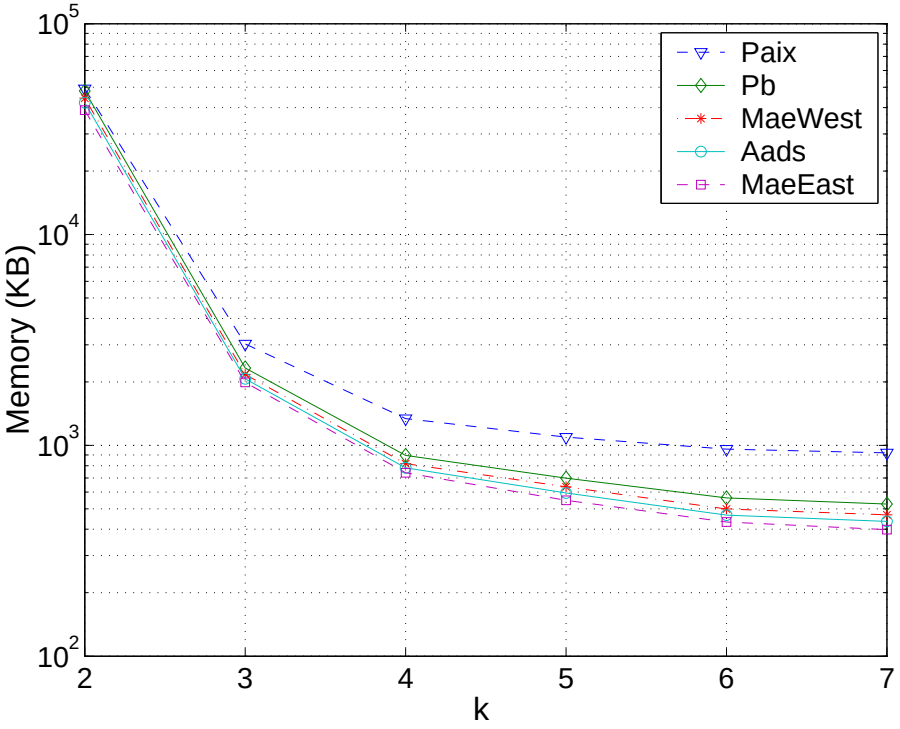



Figure 12: Memory required (in KBytes) by best $k$-level FST

| | 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 3: Execution time (in $\mu$ sec) for FST algorithms

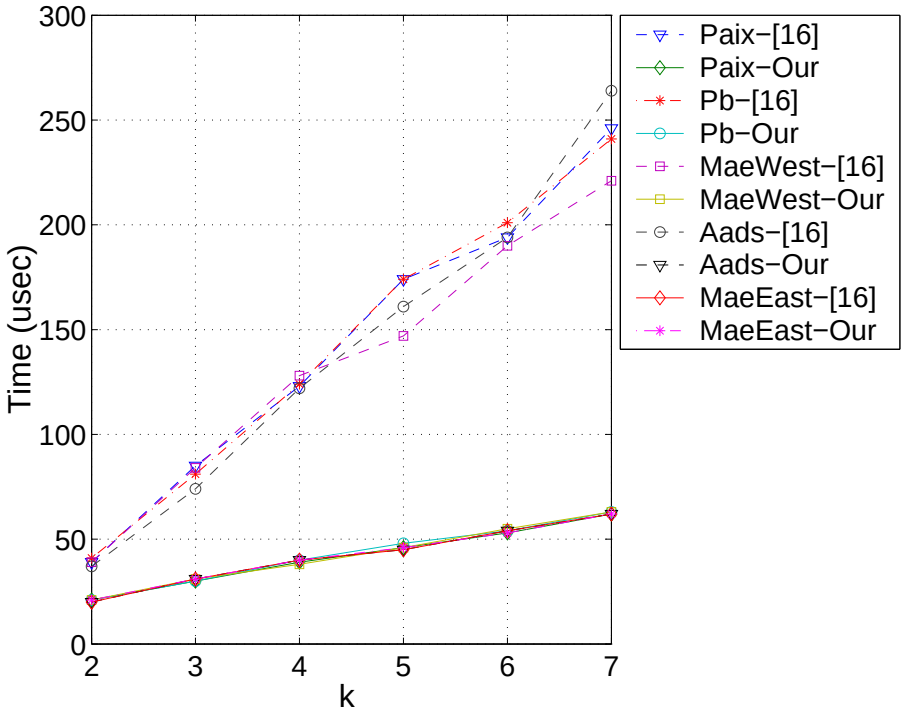



Figure 13: Execution time (in $\mu$ sec) for FST algorithms

### 3.2 Performance of Variable-Stride Algorithm

Table 4 shows the memory required by the best $k$-level VST for each of the five 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). The data of Table 4 as well as the memory requirements of the best FST are plotted in Figure 14. 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 2.3.1. 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 2.3.1. The $O(W^2k)$ memory required by this implementation for the cost arrays is

| | Paix | | Pb | | MaeWest | | Aads | | MaeEast | |
|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | No | Yes | No | Yes | No | Yes | No | Yes | No | Yes |
| 2 | 2528 | 1722 | 1806 | 1041 | 1754 | 949 | 1631 | 891 | 1621 | 837 |
| 3 | 1080 | 907 | 677 | 496 | 619 | 443 | 582 | 405 | 537 | 367 |
| 4 | 845 | 749 | 489 | 397 | 441 | 351 | 410 | 320 | 371 | 286 |
| 5 | 780 | 706 | 440 | 370 | 393 | 327 | 363 | 297 | 326 | 264 |
| 6 | 763 | 695 | 426 | 361 | 379 | 319 | 350 | 290 | 313 | 257 |
| 7 | 759 | 692 | 422 | 358 | 376 | 316 | 346 | 287 | 310 | 254 |

Table 4: Memory required (in KBytes) by best $k$-VST

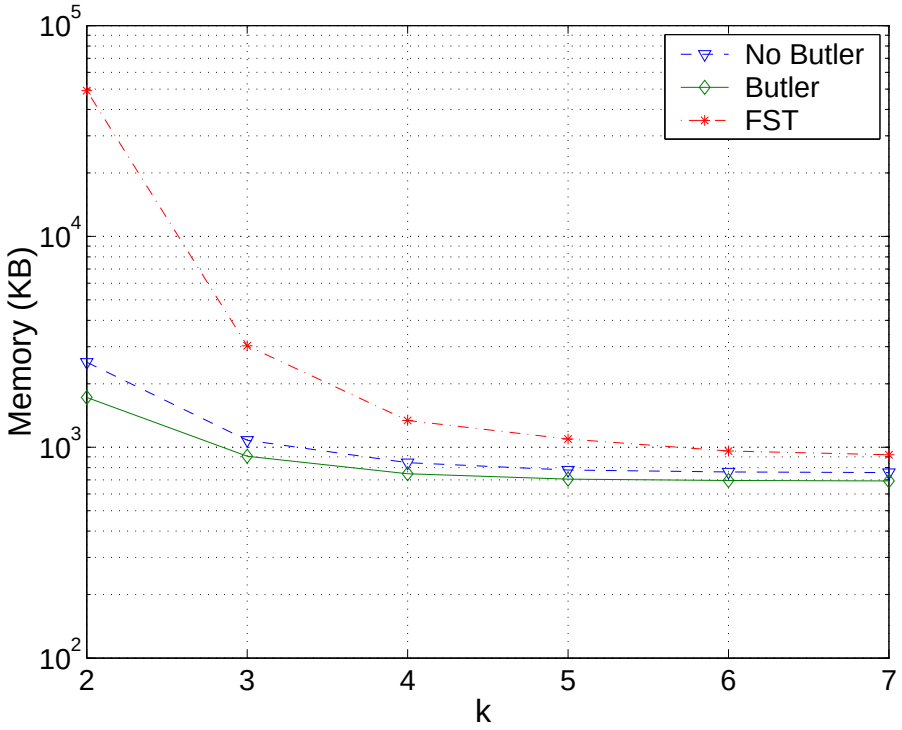



Figure 14: Memory required (in KBytes) by best $k$-VST and best FST

| | Paix | | Pb | | MaeWest | | Aads | | MaeEast | |
|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | S | D | S | D | S | D | S | D | S | D |
| 2 | 290 | 500 | 150 | 280 | 150 | 260 | 120 | 200 | 120 | 230 |
| 3 | 360 | 790 | 190 | 460 | 180 | 430 | 150 | 340 | 150 | 340 |
| 4 | 430 | 900 | 210 | 520 | 220 | 430 | 180 | 430 | 160 | 390 |
| 5 | 490 | 1140 | 260 | 610 | 240 | 570 | 200 | 520 | 190 | 470 |
| 6 | 530 | 1170 | 290 | 670 | 270 | 570 | 270 | 550 | 220 | 510 |
| 7 | 590 | 1390 | 330 | 780 | 300 | 690 | 300 | 630 | 260 | 560 |

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

Table 5: Execution times (in msec) for first two implementations of our VST algorithm

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 70 | 30 | 30 | 20 | 20 |
| 3 | 210 | 100 | 90 | 80 | 70 |
| 4 | 550 | 290 | 270 | 270 | 240 |
| 5 | 640 | 350 | 370 | 330 | 260 |
| 6 | 740 | 430 | 390 | 410 | 350 |
| 7 | 920 | 530 | 450 | 400 | 350 |

Table 6: Execution times (in msec) for third implementation of our VST algorithm

allocated at compile time. During execution, memory segments 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 2.3.3 and 2.3.4. Table 5 gives the run time for these two implementations.

The third implementation of our VST algorithm uses the faster $k = 2$ and $k = 3$ algorithms of Section 2.3.3 and 2.3.4 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 6. The run times for all three of our implementations is plotted in Figure 15. 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 2.3.3 and 2.3.4).

To compare the run time performance of our algorithm with that of [16], we use the times for imple-

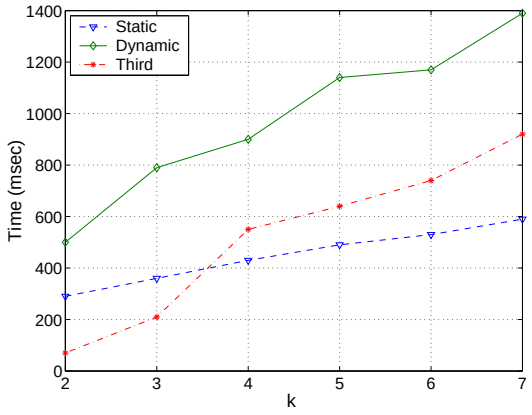



Figure 15: Execution times (in msec) for our three VST implementations

| | Paix | | Pb | | MaeWest | | Aads | | MaeEast | |
|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | [16] | Our | [16] | Our | [16] | Our | [16] | Our | [16] | Our |
| 2 | 190 | 70 | 130 | 30 | 50 | 30 | 40 | 20 | 40 | 20 |
| 3 | 1960 | 210 | 1230 | 100 | 360 | 90 | 320 | 80 | 280 | 70 |
| 4 | 3630 | 430 | 2330 | 210 | 700 | 220 | 590 | 180 | 530 | 160 |
| 5 | 5340 | 490 | 3440 | 260 | 1030 | 240 | 860 | 200 | 780 | 190 |
| 6 | 7510 | 530 | 4550 | 290 | 1340 | 270 | 1150 | 270 | 1020 | 220 |
| 7 | 9280 | 590 | 5650 | 330 | 1650 | 300 | 1420 | 300 | 1270 | 260 |

Table 7: Execution times (in msec) for our best VST implementation and the VST algorithm of [16]

mentation 3 when $k = 2$ or $k = 3$ and the times for implementation 1 when $k > 3$. That is, we compare our best times with the times for the algorithm of [16]. The times for the algorithm of [16] were obtained using their code and running it with the Butler node option off. The run times are shown in Table 7 and these times are plotted in Figure 16. For our largest database, Paix, our new algorithm takes less than half the time taken by the algorithm of [16] 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 5–7 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 8 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

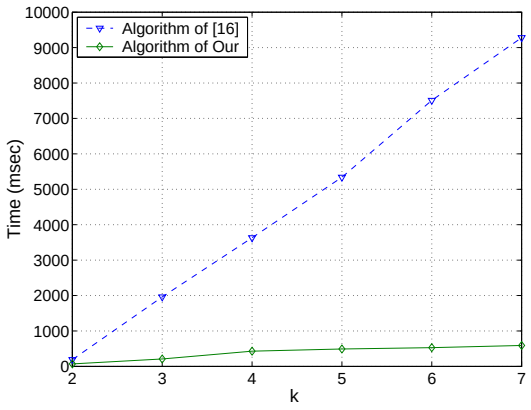



Figure 16: Execution times (in msec) for our best VST implementation and the VST algorithm of [16]

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 290 | 170 | 100 | 170 | 120 |
| 3 | 340 | 210 | 170 | 130 | 160 |
| 4 | 410 | 250 | 220 | 210 | 130 |
| 5 | 510 | 290 | 230 | 200 | 170 |
| 6 | 470 | 270 | 280 | 230 | 230 |
| 7 | 500 | 300 | 270 | 260 | 200 |

Table 8: Time (in msec) to construct optimal VST from optimal stride data

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 486 | 424 | 426 | 417 | 405 |
| 3 | 598 | 467 | 494 | 447 | 484 |
| 4 | 659 | 569 | 566 | 572 | 541 |
| 5 | 757 | 628 | 638 | 609 | 616 |
| 6 | 841 | 665 | 699 | 698 | 660 |
| 7 | 918 | 730 | 719 | 701 | 704 |

Table 9: Search time (in nsec) in optimal VST

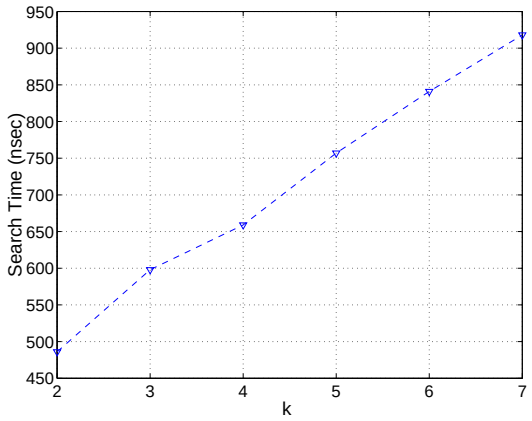



Figure 17: Search time (in nsec) in optimal VST

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 9 and Figure 17 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.

Inserts and deletes are performed less frequently than searches in a VST. We experimented with three strategies for these two operations:

1. **[OptVST]** In this strategy, the VST was always the best possible $k$-VST for the current set of prefixes. To insert a new prefix, we first insert the prefix into the 1-bit trie of all prefixes. Then, the cost arrays on the insert path are recomputed. This is done efficiently using implementation 2 (i.e., the $O(nWk)$ dynamic memory implementation) of our VST stride computation algorithm. Following this, the optimal strides for vertices on the insert path are computed. Since, the optimal VST for the new prefix set differs from the optimal VST for the original prefix set only along

the insert path, we modify the original optimal VST only along this insert path using the newly computed strides for the vertices on this path. Deletion works in a similar fashion.

2. [**Batch1**] In this strategy, the optimal VST is computed periodically (say, after a sufficient number of inserts/deletes have taken place) rather than following each insert/delete. Inserts and deletes are done directly into the current VST without regard to maintaining optimality. If the insertion results in the creation of a new node, the stride of this new node is such that the sum of the strides of the nodes on the path from the root to this new node equals the length of the newly inserted prefix. The deletion of a prefix may require us to search a node for a replacement prefix of the next (lower) length that matches the deleted prefix.

3. [**Batch2**] This differs from strategy Batch1 in that inserts and deletes are done in both the current VST and in the 1-bit trie. This increases the time for an insert but reduces the time for a delete. Insert times increase because we insert into two structures. In the case of deletion, by first deleting from the 1-bit trie, we determine the next (lower) length matching prefix from the delete path taken in the 1-bit trie. This eliminates the need to search a node for this next (lower) length matching prefix when deleting from the VST. The result is a net reduction in time for the delete operation.

The batch modes described above may also be useful when the insert/delete rate is sufficiently small that following each insert or delete done as above, the optimal VST is computed in the background using another processor. While this computation is being done, routes are made using the suboptimal VST resulting from the insert or delete that was done as described for the batch modes. When the new optimal VST has been computed, the new optimal VST is swapped with the suboptimal one.

Tables 10-15 give the measured run times for the insert and delete operations using each of the three strategies described above. Figures 18 and 19 plot these times for the Paix database. For the insert time experiments, we started with an optimal VST for 90% of the prefixes in the given database and then measured the time to insert the remaining 10%. The reported times are the average time for one insert. Although batch insertion is considerably faster than insertion using strategy optVST, batch insertion increases the number of levels in the VST, and so results in slower searches. For example, in the experiments with Paix, the batch inserts increased the number of levels in the initial $k$-VST from $k$ to 6 for $k = 2$, 3, and 6, to 7 for $k = 4$ and 5, and to 8 for $k = 7$. The delete times were measured by starting with an optimal VST for 100% of the prefixes in the given database and then measuring the time to delete 10% of these prefixes. Once again, the average time for a single delete is reported.

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 161 | 155 | 159 | 195 | 162 |
| 3 | 245 | 235 | 237 | 380 | 312 |
| 4 | 338 | 320 | 322 | 303 | 330 |
| 5 | 954 | 390 | 745 | 413 | 902 |
| 6 | 547 | 532 | 1012 | 1485 | 554 |
| 7 | 823 | 651 | 1113 | 650 | 730 |

Table 10: Insertion time (in $\mu$sec) for optVST

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 794 | 1203 | 1940 | 1189 | 889 |
| 3 | 252 | 209 | 211 | 229 | 223 |
| 4 | 262 | 269 | 270 | 269 | 295 |
| 5 | 325 | 337 | 341 | 343 | 352 |
| 6 | 400 | 402 | 397 | 402 | 427 |
| 7 | 459 | 464 | 475 | 461 | 506 |

Table 11: Deletion time (in $\mu$sec) for optVST

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 4.1 | 4.5 | 5.2 | 5.3 | 5.0 |
| 3 | 3.7 | 5.7 | 5.3 | 5.2 | 4.4 |
| 4 | 3.8 | 4.4 | 5.6 | 4.6 | 5.5 |
| 5 | 3.8 | 4.2 | 5.1 | 4.3 | 5.3 |
| 6 | 4.3 | 4.3 | 6.0 | 6.8 | 6.9 |
| 7 | 5.1 | 7.4 | 7.2 | 8.4 | 6.0 |

Table 12: Insertion time (in $\mu$sec) for Batch1

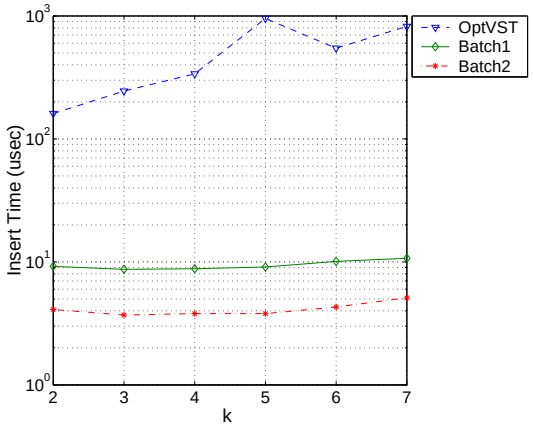



Figure 18: Insertion time (in $\mu$sec) for Paix

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 953.7 | 1167.1 | 1090.8 | 1977.1 | 1730.5 |
| 3 | 83.3 | 14.7 | 17.6 | 17.0 | 16.7 |
| 4 | 5.2 | 5.9 | 5.5 | 5.9 | 6.2 |
| 5 | 6.3 | 7.1 | 5.9 | 6.3 | 9.7 |
| 6 | 11.5 | 12.7 | 6.5 | 6.7 | 7.0 |
| 7 | 11.9 | 6.8 | 6.2 | 7.4 | 7.5 |

Table 13: Deletion time (in $\mu$sec) for Batch1

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 9.2 | 11.0 | 11.7 | 11.1 | 11.0 |
| 3 | 8.7 | 12.7 | 8.8 | 13.3 | 11.4 |
| 4 | 8.8 | 11.3 | 11.4 | 12.2 | 10.6 |
| 5 | 9.1 | 11.9 | 11.1 | 10.7 | 12.3 |
| 6 | 10.1 | 12.5 | 12.7 | 11.8 | 13.2 |
| 7 | 10.7 | 15.3 | 13.4 | 14.4 | 14.1 |

Table 14: Insertion time (in $\mu$sec) for Batch2

| $k$ | Paix | Pb | MaeWest | Aads | MaeEast |
|---|---|---|---|---|---|
| 2 | 4.9 | 4.9 | 5.5 | 5.3 | 5.5 |
| 3 | 4.7 | 5.4 | 4.8 | 4.9 | 5.1 |
| 4 | 4.8 | 5.0 | 4.9 | 4.8 | 5.1 |
| 5 | 4.9 | 5.1 | 4.9 | 5.0 | 5.3 |
| 6 | 5.1 | 5.2 | 5.2 | 5.5 | 5.5 |
| 7 | 5.3 | 5.7 | 5.4 | 5.4 | 5.6 |

Table 15: Deletion time (in $\mu$sec) for Batch2

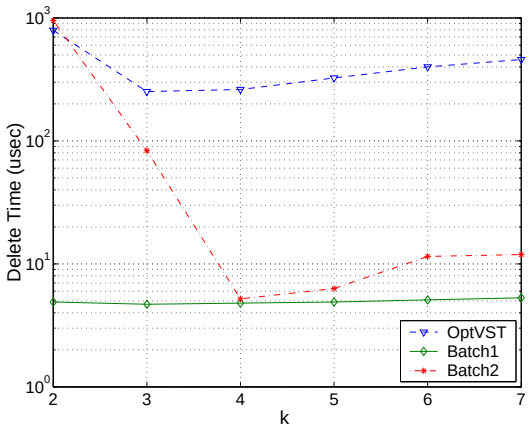



Figure 19: Deletion time (in $\mu$sec) for Paix

# 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 also have developed a faster algorithm, to compute the optimal strides for variable stride tries, than the one proposed in [16]. 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.

[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.