Efficient Construction of Pipelined Multibit-Trie Router-Tables [*]

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

Abstract

Efficient algorithms to construct multibit tries suitable for pipelined router-table applications are developed. We first enhance the 1-phase algorithm of Basu and Narlikar [1] obtaining a 1-phase algorithm that is 2.5 to 3 times as fast. Next we develop 2-phase algorithms that not only guarantee to minimize the maximum per-stage memory but also guarantee to use the least total memory subject to the former constraint. Our 2-phase algorithms not only generate better pipelined trees than generated by the 1-phase algorithm but they also take much less time. A node pullup scheme that guarantees no increase in maximum per-stage memory as well as a partitioning heuristic that generates pipelined multibit tries requiring less maximum per-stage memory than required by the tries obtained using the 1-phase and 2-phase algorithms also are proposed.

Keywords: Packet routing, longest matching-prefix, controlled prefix expansion, multibit trie, pipelined router-table, dynamic programming.

1 Introduction

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 output 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.

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 [25], which is a variant

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

of a compressed binary trie [11]. 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. [6]. LC tries for longest prefix matching are developed in [19]. Degermark et al. [5] 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 [5], 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 [26] 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 [5], 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 Varghese [26] and Sahni and Kim [22] develop dynamic programming algorithms to obtain space optimal fixed-stride and variable-stride tries of a given height.

Waldvogel et al. [29] 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 $O(\log W)$. Kim and Sahni [13] have developed fast algorithms to optimize the binary search scheme of [29]. An alternative adaptation of binary search to longest prefix matching is developed in [14]. Using this adaptation, a lookup in a table that has n prefixes takes $O(W + \log n)$ time.

Cheung and McCanne [4] 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.

Data structures for dynamic router tables (i.e., tables designed to efficiently support rule insertion and deletion as well as lookup) are proposed in [15, 24, 27]. Hardware-based solutions for high speed packet forwarding are classified into two categories–those that involve the use of content addressable memory [16] and those that use the ASIC-based pipelining technique [1]. Solutions that involve modifications to the Internet Protocol (i.e., the addition of information to each packet) also have been proposed [3, 18, 2].

Ruiz-Sanchez et al. [21] survey data structures for static router-tables (i.e., router tables optimized for the lookup operation) while Sahni et al. [23] survey data structures for both static and dynamic router-tables.

In this paper, we focus on the ASIC-based pipelining technique of Basu and Narlikar [1]. Basu and Narliker [1] propose a pipelined forwarding engine for dynamic router tables. This forwarding engine is based on the fixed-stride trie data structure. When designing a fixed-stride trie structure for non-pipelined applications we optimize total memory subject to a height constrained [22, 26]. For pipelined applications, however, Basu and Narlikar [1] state the following constraints:

C1: Each level in the fixed-stride trie must fit in a single pipeline stage.

C2: The maximum memory allocated to a stage (over all stages) is minimum.

C3: The total memory used is minimized subject to the first two constraints.

Constraint 3 is important because studies conducted by Basu and Narliker [1] using real Internet traces indicate that pipeline disruptions due to update operations are reduced when the total memory used by the trie is reduced. Basu and Narlikar [1] propose a dynamic programming algorithm to construct optimal fixed-stride tries subject to the constraints C1-C3. We refer to this algorithm as the *1-phase algorithm*.

In Section 2, we define the pipelined fixed-stride trie optimization problem and explain why the algorithm proposed by Basu and Narliker [1] doesn't minimize total memory subject to constraints C1 and C2. In Section 3, we propose a modification to the 1-phase algorithm of [1] that reduces its run time to half (or less) while computing the same solution. In Section 4, we develop a 2-phase algorithm that constructs fixed-stride tries that satisfy the three constraints C1-C3. The fixed-stride tries generated by the 2-phase algorithm require the same maximum per-stage memory as required by the fixed-stride tries generated by the 1-phase algorithm. However, the 2-stage algorithm's fixed-stride tries often use less total memory. Faster 2-phase algorithms for 2- and 3-stage pipelines are developed in Section 5. We develop a partitioning algorithm, which reduces the maximum per-stage memory, in Section 6. In Section 7, we provide a new node pullup scheme to reduce total memory without increasing the maximum per-stage memory. In Section 8, we present our experimental results.

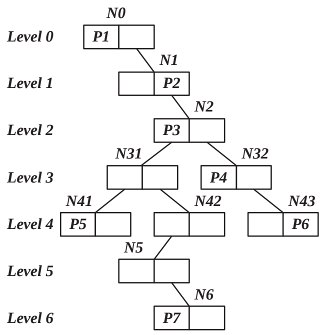
2 Problem Statement

2.1 1-Bit Tries

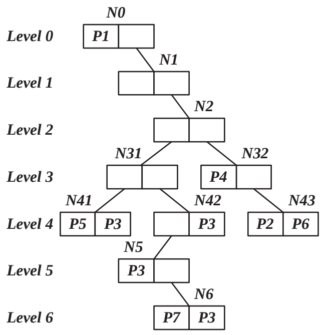
A *1-bit trie* is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level $l-1$ of the trie store prefixes whose length is l (the length of a prefix is the number of bits in that prefix; the terminating * (if present) does not count towards 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 a set of 7 prefixes, and Figure 1(b) shows the corresponding 1-bit trie. 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 [11]. The primary difference being that the 1-bit tries of [11] are for the case when all keys (prefixes) have the same length.

Prefixes
P1 **= 0***
P2 **= 11***
P3 **= 110***
P4 **= 1110***
P5 **= 11000***
P6 **= 11111***
P7 **= 1101010***

(a) A prefix set



(b) 1-bit trie



(c) Leaf-pushed 1-bit trie

Null fields are shown as blanks.

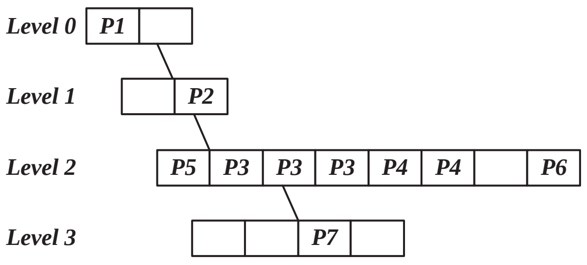
Figure 1: Prefixes and corresponding 1-bit tries

In a *leaf-pushed trie* [26], prefixes residing in non-leaf nodes of a 1-bit trie are pushed down to leaf nodes; longer prefixes take precedence over shorter ones. Figure 1(c) shows the leaf-pushed trie that corresponds to the 1-bit trie of Figure 1(b). Notice that a leaf-pushed trie has the same number of nodes as does its non-leaf-pushed counterpart; however, each node has half as many fields (we assume the existence of an implicit mechanism to distinguish between a field that contains a prefix and one that contains a pointer to a child). As in [1], we assume, in this paper, that leaf-pushed tries are used.

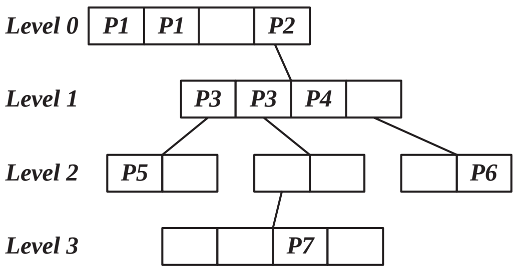
When 1-bit tries (either leaf-pushed or non-leaf-pushed) are used to represent IPv4 router tables, the trie height may be as much as 31. A lookup in such a trie takes up to 32 memory accesses.

2.2 Fixed-Stride Tries

The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. In a leaf-pushed trie, a node whose stride is s has 2^s fields (corresponding to the 2^s possible values for the s bits that are used). Some of these fields may point to subtries and other may contain a prefix. A node whose stride is s 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. In a 1-bit trie, the stride of every node is 1. Figure 2 shows two 4-level (non-leaf-pushed) FSTs for the prefix set of Figure 1(a). The strides for the levels of the FST of Figure 2(a) are 1, 1, 3 and 2 and those for the levels of the FST of Figure 2(b) are 2, 2, 1, 2. Figure 3 shows the corresponding leaf-pushed tries. In the sequel we shall be dealing exclusively with leaf-pushed tries and shall refrain from explicitly calling our FSTs leaf-pushed FSTs.



(a) FST for Figure 1(a)



(b) Another FST for Figure 1(a)

Figure 2: Fixed-stride tries

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 $b = a + s_j - 1$. So, level 0 of the FST of Figure 3(a) covers level 0 of the 1-bit trie of Figure 1(c). Level 1 of this FST covers level 1 of the 1-bit trie of Figure 1(c); level 2 of this FST covers levels 2, 3, and 4 of the 1-bit trie; and level 3

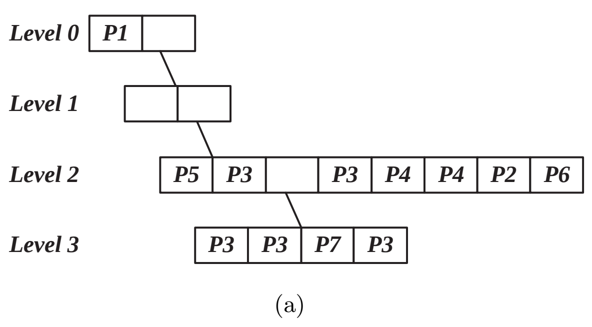


Figure 3: Leaf-pushed fixed-stride tries corresponding to Figure 2

of this FST covers levels 5 and 6 of the 1-bit trie. Let e_u be defined as below:

$$e_u = \sum_{0}^{u-1} s_q, 0 \leq u < k \tag{1}$$

We shall refer to the e_us as the *expansion levels* of O. The expansion levels defined by the FST of Figure 3(a) are 0, 1, 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), $nodes(0:6) = [1,1,1,2,3,1,1]$. The memory required by an FST is $\sum_0^{k-1} nodes(e_q) * 2^{s_q}$. For example, the memory required by the FST of Figure 3(a) is $nodes(0)*2^1 + nodes(1)*2^1 + nodes(2)*2^3 + nodes(5)*2^2$ $= 16$.

In the *fixed-stride trie optimization* (FSTO) problem [22], we are given a set S 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. 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. Note that the number of expansion levels equals the number of levels in the corresponding FST. 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 [26] have obtained the following dynamic programming recurrence for T:

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

$$T(j,1) = 2^{j+1} \tag{3}$$

Sahni and Kim [22] have shown that $T(j,r)$ isn't monotone in r. In particular, it is possible that $T(j,r) < T(j,r+1)$. So Sahni and Kim [22] consider constructing optimal FSTs with *at most* k levels. To maintain compatibility with [1] we consider FSTs with exactly k rather than at most k levels. Our results of this paper extend easily to the case when we require at most k levels.

As noted by Srinivasan and Varghese [26], 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.

Figure 2(a) shows the optimal FST for the prefix set of Figure 1(a), and Figure 3(a) shows the optimal leaf-pushed FST.

2.3 Pipelined Fixed-Stride Tries

When mapping an FST onto a pipelined architecture we place each level of the FST into a different stage of the pipeline. Specifically, level i of the FST is placed into stage i of the pipeline. So if our pipeline has k stages, we seek an FST that has k levels. However, since all stages of the pipeline are identical, the appropriate optimization criteria for the FST isn't total memory required but the maximum memory required by any level of the FST. Note that the memory required by an FST level is the sum of the memory requirements of the nodes at that level. So, for example, level 2 of the FST of Figure 3(a) requires 8 units of memory while level 2 of the FST of Figure 3(b) requires 6 units of memory. Experiments conducted by Basu and Narlikar [1] indicate that minimizing total memory is a good secondary criteria as FSTs with smaller total memory requirement improve the update performance of the pipelined router-table.

The *Pipelined* FSTO problem (PFSTO) is to determine the FST that uses exactly k levels for the given prefix set S and satisfies the three constraints C1-C3. Let $MS(j, r, R)$ be the maximum memory required by any level of the r-level FST R that covers levels 0 through j of the given 1-bit trie. Let $MMS(j, r)$ be the minimum of $MS(j, r, R)$ over all possible FSTs R. $MMS(j, r)$ (called $MinMaxSpace[j, r]$ in [1]) is the minimum value for constraint C2. Let $T(j, r)$ be the minimum total memory for constraint C3. Basu and Narlikar [1] obtain the following dynamic programming recurrence equations for MMS and T:

$$MMS(j, r) = \min_{m \in \{r-2..j-1\}} \{\max\{MMS(m, r-1), nodes(m+1) * 2^{j-m}\}\}, r > 1 \tag{4}$$

$$MMS(j, 1) = 2^{j+1} \tag{5}$$

$$T(j, r) = \min_{m \in \{r-2..j-1\}} \{T(m, r-1) + nodes(m+1) * 2^{j-m}\}, r > 1 \tag{6}$$

$$T(j, 1) = 2^{j+1} \tag{7}$$

where

$$nodes(m+1) * 2^{j-m} \leq P$$

Here P is the memory capacity of a pipeline stage. In solving Equations 4 through 7, Basu and Narlikar [1] "give Equation 4 precedence over Equation 6 when choosing m. When multiple values of m yield the same value of $MMS(j, r)$, Equation 6 is used to choose between these values of m." When the maximum prefix length is W and k pipeline stages are available, $MMS(W-1, k)$ gives the minimum per-stage memory required for a k-level FST. We refer to the Basu and Narlikar [1] dynamic programming algorithm as the 1-phase algorithm because this algorithm computes $MMS(W-1, k)$ and $T(W-1, k)$ concurrently as described above.

As noted by Basu and Narlikar [1], using the above recurrence, we may determine both $MMS(W-1,k)$ and $T(W-1,k)$ in $O(kW^2)$ time. Figure 3(b) shows the FST for the prefixes of Figure 1(a) that results when we use the recurrence of Basu and Narlikar [1]. The total memory required by this FST is $nodes(0) * 2^2 + nodes(2) * 2^2 + nodes(4) * 2^1 + nodes(5) * 2^2 = 18$. This is slightly more than the total memory, 16, required by the optimal FST of Figure 3(a). However, allocating one level per pipeline stage, the maximum per-stage memory required by the FST of Figure 3(a) is 8 (for level 2) whereas for the FST of Figure 3(b) this quantity is 6 (also for level 2).

Although the 1-phase algorithm of Basu and Narlikar [1] computes the MMS correctly, the T values computed are not the best possible. This is because in their algorithm the computation of the Ts is overconstrained. The computation of $T(j,r)$ is constrained by $MMS(j,r)$ rather than by the potentially larger $MMS(W-1,k)$ value. As our experimental studies will show, this overconstraining actually leads to PFSTs with larger total memory requirement than is necessary for real-world prefix databases. In Section 4 we propose a 2-phase algorithm to correctly compute $T(W-1,k)$ under constraints C1 and C2. Our 2-phase algorithm computes $MMS(W-1,k)$ in the first phase and then uses the computed $MMS(W-1,k)$ in the second phase to obtain $T(W-1,k)$.

Let k-PFST denote the optimal k-level pipelined fixed stride trie for the given prefix set. This FST satisfies constraints C1-C3.

3 Fast 1-Phase Algorithm

The 1-phase algorithm of [1] examines all m in the range $[r-2, j-1]$ when solving the recurrence of Equations 4 through 7. We may use Lemma 2 to avoid the examination of some of these values of m.

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

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

Lemma 2 *Let q be an integer in the range $[r-2, j-2]$ and let*

$$minMMS = \min_{m \in \{q+1..j-1\}} \{\max\{MMS(m, r-1), nodes(m+1) * 2^{j-m}\}\}, r > 1$$

If $nodes(q+1) * 2^{j-q} > minMMS$, then for every s in the range $[r-2, q]$

$$\max\{MMS(s, r-1), nodes(s+1) * 2^{j-s}\} > minMMS$$

Further, $MMS(j, r) = minMMS$.

Proof From Lemma 1(b) it follows that

$$nodes(s+1) * 2^{q-s} \geq nodes(q+1)$$

So,

$$nodes(s+1) * 2^{j-s} \geq nodes(q+1) * 2^{j-q} > minMMS$$

Hence,

$$\max\{MMS(s, r-1), nodes(s+1) * 2^{j-s}\} > minMMS$$

From this, the definition of $minMMS$, and Equation 4, it follows that $MMS(j, r) = minMMS$. ■

Lemma 2 results in Algorithm *Fast1Phase* (Figure 4), which computes both $MMS(W-1, k)$ and $T(W-1, k)$ in a single phase using Equations 4 through 7. This algorithm computes the same values for $MMS(W-1, k)$ and $T(W-1, k)$ as are computed by the 1-phase algorithm of [1]. So the algorithm computes the correct value for $MMS(W, k-1)$ but the value computed for $T(W-1, k)$ may not be the minimum total memory subject to constraints C1 and C2. Although the asymptotic complexity, $O(kW^2)$, of Algorithm *Fast1Phase* is the same as that of the 1-phase algorithm of [1], Algorithm *Fast1Phase* is expected to run faster because of its application of Lemma 2. This expectation is borne out by our experiments (see Section 8).

```
**Algorithm** *Fast1Phase*$(W, k)$
// $W$ is length of longest prefix.
// $k$ is maximum number of expansion levels desired.
// Return $MMS(W-1, k)$ and $T(W-1, k)$.
{
        **for** $(j = 0; j < W; j++)$
                $MMS(j, 1) = T(j, 1) = 2^{j+1}$;
        **for** $(r = 2; r \le k; r++)$
                **for** $(j = r-1; j < W; j++)\{$
                // Compute $MMS(j, r)$ and $T(j, r)$.
                        $minMMS = \infty; minT = \infty;$
                        **for** $(m = j-1; m \ge r-2; m--)\{$
                                **if** $(nodes(m+1) * 2^{j-m} > minMMS)$
                                        **break;** // by Lemma 2
                                $mms = max(MMS(m, r-1), nodes(m+1) * 2^{j-m});$
                                **if** $(mms < minMMS)\{$
                                        $minMMS = mms;$
                                        $minT = T(m, r-1) + nodes(m+1) * 2^{j-m};\}$
                                **else if** $(mms == minMMS)\{$
                                        $t = T(m, r-1) + nodes(m+1) * 2^{j-m};$
                                        **if** $(t < minT)$ $minT = t;\}\}$
                        $MMS(j, r) = minMMS; T(j, r) = minT;\}$
        **return** $(MMS(W-1, k), T(W-1, k));$
}
```

Figure 4: 1-phase PFST algorithm using Lemma 2

4 2-Phase Computation of MMS and T

As noted earlier, the 1-phase algorithm of Basu and Narlikar [1] correctly computes $MMS(W-1,k)$. However the $T(W-1,k)$ value computed by this 1-phase algorithm isn't guaranteed to be the minimum total memory subject to constraints C1 and C2. The correct $T(W-1,k)$ value may be computed using a 2-phase algorithm. In the first phase, $MMS(W-1,k)$ is computed. The second phase uses the $MMS(W-1,k)$ computed in phase 1 to arrive at the correct $T(W-1,k)$ value.

In Section 4.1 we describe three algorithms to compute $MMS(W-1,k)$. The first of these (Section 4.1.1), which is based on Equations 4 and 5, takes $O(kW^2)$ time. However, it runs faster than a direct solution of these equations because it reduces the search range for m in Equation 4. The reduction in search range is greater than obtainable by Lemma 2 because we are not trying to compute T at the same time. The second algorithm (Section 4.1.2) computes $MMS(W-1,k)$ in $O(W^2)$ time by performing a binary search over the space of possible values for $MMS(W-1,k)$. Our third algorithm (Section 4.1.3) for $MMS(W-1,k)$ uses the parametric search technique [8] and runs in $O(W \log W)$ time. Two dynamic programming algorithms for stage 2 are given in Section 4.2. The complexity of each is $O(kW^2)$.

4.1 Computing $MMS(W-1,k)$

4.1.1 Dynamic Programming

$MMS(W-1,k)$ may be computed from Equations 4 and 5 using standard methods for dynamic programming recurrences. The time required is $O(kW^2)$. Although we cannot reduce the asymptotic complexity of the dynamic programming approach, we can reduce the observed complexity on actual prefix-databases by a constant factor by employing certain properties of MMS.

Lemma 3 *When $r \leq l < W$, $MMS(l, r+1) \leq MMS(l, r)$.*

Proof When $l \geq r$, at least one level of the FST F that achieves $MMS(l,r)$ has a stride ≥ 2 (F has r levels while the number of levels of the 1-bit trie that are covered by F is $l+1$). Consider any level v of F whose stride is 2 or more. Let s be the stride for level v and let w be the expansion level represented by level v of F. Split level v into two to obtain an FST G that has $r+1$ levels. The first of these split levels has a stride of 1 and the stride for the second of these split levels is $s-1$. The remaining levels have the same strides as in F. From Lemma 1 it follows that $nodes(w+1) \leq 2 * nodes(w)$. Therefore,

$$\max\{nodes(w) * 2, nodes(w+1) * 2^{s-1}\} \leq nodes(w) * 2^s$$

So, the maximum per-stage memory required by G is no more than that required by F. Since G is an $(r+1)$-level FST it follows that $MMS(l, r+1) \leq MMS(l,r)$. ■

Lemma 4 *When $l < W$ and $\lceil l/2 \rceil \geq r-1 \geq 1$, $MMS(l,r) \leq nodes(\lceil l/2 \rceil) * 2^{l-\lceil l/2 \rceil+1}$.*

Proof From Equation 4 it follows that

$$MMS(l,r) \leq \max\{MMS(\lceil l/2 \rceil - 1, r-1), nodes(\lceil l/2 \rceil) * 2^{l-\lceil l/2 \rceil+1}\}$$

From Lemma 3 and Equation 5, we obtain

$$\begin{aligned} MMS(\lceil l/2 \rceil - 1, r-1) &\leq MMS(\lceil l/2 \rceil - 1, 1) \\ &= 2^{\lceil l/2 \rceil} \end{aligned}$$

The lemma now follows from the observation that $nodes(\lceil l/2 \rceil) \geq 1$ and $2^{l-\lceil l/2 \rceil+1} \geq 2^{\lceil l/2 \rceil}$. ■

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

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

in Equation 4.

Lemma 5 *For $l < W$ and $r \geq 2$, $M(l,r) \geq \lceil l/2 \rceil - 1$.*

Proof From Equation 4, $M(l,r) \geq r-2$. So if $r - 2 \geq \lceil l/2 \rceil - 1$, the lemma is proved. Assume that $r - 2 < \lceil l/2 \rceil - 1$. Now, Lemma 4 applies and $MMS(l,r) \leq nodes(\lceil l/2 \rceil) * 2^{l - \lceil l/2 \rceil + 1}$. From the proof of Lemma 4, we know that this upper bound is attained, for example, when $m = \lceil l/2 \rceil - 1$ in Equation 4. From the proof of Lemma 4 and from Lemma 1 it follows that for any smaller m value, say i,

$$\begin{aligned} & \max\{MMS(i, r-1), nodes(i+1) * 2^{l-i}\} \\ = \quad & nodes(i+1) * 2^{l-i} \\ \geq \quad & nodes(\lceil l/2 \rceil) * 2^{i - \lceil l/2 \rceil + 1} * 2^{l-i} \\ = \quad & nodes(\lceil l/2 \rceil) * 2^{l - \lceil l/2 \rceil + 1} \end{aligned}$$

So a smaller value of m cannot get us below the upper bound of Lemma 4. Hence, $M(l,r) \geq \lceil l/2 \rceil - 1$.

■

Theorem 1 *For $l < W$ and $r \geq 2$, $M(l,r) \geq \max\{r-2, \lceil l/2 \rceil - 1\}$.*

Proof Follows from Lemma 5 and Equation 4. ■

Theorem 1 and a minor extension of Lemma 2 lead to Algorithm *DynamicProgrammingMMS* (Figure 5), which computes $MMS(W-1,k)$. The complexity of this algorithm is $O(kW^2)$.

4.1.2 Binary Search

It is easy to see that for $k > 1$, $0 < MMS(W-1,k) < 2^W$. The range for $MMS(W-1,k)$ may be narrowed by making the following observations:

LB: From Lemma 3, $MMS(W-1,W) \leq MMS(W-1,k)$. Since,

$$MMS(W-1,W) = \max_{0 \leq i < W}\{nodes(i) * 2\} \tag{8}$$

```
**Algorithm** *DynamicProgrammingMMS*$(W, k)$
// $W$ is length of longest prefix.
// $k$ is maximum number of expansion levels desired.
// Return $MMS(W-1, k)$.
{
    **for** $(j = 0; j < W; j++)$
        $MMS(j, 1) = 2^{j+1}$;
    **for** $(r = 2; r \leq k; r++)$
        **for** $(j = r-1; j < W; j++)\{$
        // Compute $MMS(j, r)$.
            $minJ = max(r-2, \lceil j/2 \rceil - 1)$; // by Theorem 1
            $minMMS = \infty$;
            **for** $(m = j-1; m \geq minJ; m--)\{$
                **if** $(nodes(m+1) * 2^{j-m} >= minMMS)$
                    **break;** // by Lemma 2
                $mms = max(MMS(m, r-1), nodes(m+1) * 2^{j-m})$;
                **if** $(mms < minMMS)$
                    $minMMS = mms$; }
            $MMS(j, r) = minMMS$;}
    **return** $MMS(W-1, k)$;
}
```

Figure 5: Dynamic programming algorithm to compute $MMS(W-1, k)$.

we obtain a better lower bound on the search range for $MMS(W-1, k)$.

UB: For real-world databases and practical values for the number k of pipeline stages, Lemma 4 applies with $l = W-1$ and $r = k$. So this lemma provides a tighter upper bound for the binary search than the bound $2^W - 1$.

To successfully perform a binary search in the stated search range, we need a way to determine for any value p in this range whether or not there is an FST with at most k levels whose per-stage memory requirement is at most p. We call this a *feasibility test* of p. Note that from Lemma 3 it follows that such an FST exists iff such an FST with exactly k levels exists. We use Algorithm *GreedyCover* (Figure 6) for our feasibility test. For a given memory constraint p, *GreedyCover* covers as many levels of the 1-bit trie as possible by level 0 of the FST. Then it covers as many of the remaining levels as possible by level 1 of the FST, and so on. The algorithm returns the value **true** iff it is able to cover all W levels by at

```
Algorithm GreedyCover(k, p)
// k is maximum number of expansion levels (i.e., pipeline stages) desired.
// p is maximum memory for each stage.
// W is length of longest prefix.
// Return true iff p is feasible.
{
        $stages = eLevel = 0$;
        while ($stages < k$){
                $i = 1$;
                while ($nodes(eLevel) * 2^i \leq p$ && $eLevel + i \leq W$) $i++$;
                if ($eLevel + i > W$) return true;
                if ($i == 1$) return false;
                // start next stage
                $eLevel += i - 1$;
                $stages++$; }
        return false ; // $stages \geq k$. So, infeasible
}
```

Figure 6: Feasibility test for p

most k FST levels.

The correctness of Algorithm *GreedyCover* as a feasibilty test may be established using standard proof methods for greedy algorithms [12]. The time complexity of *GreedyCover* is $O(W)$.

Algorithm *BinarySearchMMS* (Figure 7) is our binary search algorithm to determine $MMS(W-1,k)$. The input value of p is the lower bound as determined by Equation 8. It first checks if the lower bound is, in fact, the value of $MMS(W-1,k)$. If not, it proceeds with a binary search. However, rather than use the upper bound from Lemma 4, we use a doubling procedure to determine a suitable upper bound. Experimentally, we observed that this strategy worked better than using the upper bound of Lemma 4 except for small k such as $k = 2$ and $k = 3$. The binary search algorithm invokes *GreedyCover* $O(W)$ times. So its complexity is $O(W^2)$.

4.1.3 Parametric Search

Parametric search was developed by Frederickson [8]. Several enhancements to the original search scheme were proposed in [7, 9, 10]. Our application of parametric search to compute $MMS(W-1,k)$ closely

```
Algorithm BinarySearchMMS(k, p)
// k is the number of pipeline stages.
// Initially p is $nodes(l) * 2$ where level l has
// the max number of nodes in the 1-bit trie.
// Return $MMS(W-1, k)$.
{
        if (GreedyCover(k, p)) return p;
        do {
              $p = p * 2$;
              } while (!GreedyCover(k, p));
        $low = \lfloor p/2 \rfloor + 1; high = p; p = \lfloor (low + high)/2 \rfloor$;
        while ($low < high$){
              if (GreedyCover(k, p)) $high = p$; // feasible
              else $low = p + 1$; // infeasible
              $p = \lfloor (low + high)/2 \rfloor$; }
        return high;
}
```

Figure 7: Binary search algorithm to compute $MMS(W-1,k)$

follows the development in [28].

We start with an implicitly specified sorted matrix M of $O(W^2)$ candidate values for $MMS(W-1,k)$. This implicit matrix of candidate values is obtained by observing that $MMS(W-1,k) = nodes(i) * 2^j$ for some i and j, $0 \le i < W$ and $1 \le j \le W$. Here i denotes a possible expansion level and j denotes a possible stride for this expansion level. To obtain the implicit matrix M, we sort the distinct $nodes(i)$ values into ascending order. Let the sorted values be $n_1 < n_2 < \cdots < n_t$, where t is the number of distinct $nodes(i)$ values. The matrix value $M_{i,j}$ is $n_i * 2^j$. Since we do not explicitly compute the $M_{i,j}$s, the implicit specification can be obtained in $O(W \log W)$ time (i.e., the time needed to sort the $nodes(i)$ values and eliminate duplicates).

We see that M is a sorted matrix. That is,

$$M_{i,j} \le M_{i,j+1}, 1 \le i \le t, 1 \le j < W$$

$$M_{i,j} \le M_{i+1,j}, 1 \le i < t, 1 \le j \le W$$

Again, it is important to note that the matrix M is provided implicitly. That is, we are given a way to compute $M_{i,j}$, in constant time, for any value of i and j. We are required to find the least $M_{i,j}$ that satisfies the feasibility test of Algorithm *GreedyCover* (Figure 6). Henceforth, we refer to this feasibility test as criterion GC.

We know that the criterion GC has the following property. If $GC(x)$ is false (i.e., x is infesible), then $GC(y)$ is false (i.e., y is infeasible) for all $y \leq x$. Similarly, if $GC(x)$ is true (i.e., x is feasible), then $GC(y)$ is true for all $y \geq x$. In parametric search, the minimum $M_{i,j}$ for which GC is true is found by trying out some of the $M_{i,j}$'s. As different $M_{i,j}$'s are tried, we maintain two values λ_1 and λ_2, $\lambda_1 < \lambda_2$ with the properties: (a) $GC(\lambda_1)$ is false; and (b) $GC(\lambda_2)$ is true.

Initially, $\lambda_1 = 0$ and $\lambda_2 = \infty$ (we may use the bounds $LB - 1$ and UB as tighter bounds). With M specified implicitly, a feasibility test in place, and the initial values of λ_1 and λ_2 defined, we may use parametric search to determine the least candidate in M that is feasible. Details of the parametric search process to be employed appear in [28]. Frederickson [8] has shown that parametric search of a sorted $W \times W$ matrix M performs at most $O(\log W)$ feasibility tests. Since each feasibility test takes $O(W)$ time, $MMS(W-1,k)$ is determined in $O(W \log W)$ time.

4.2 Computing $T(W-1,k)$

4.2.1 Algorithm ***FixedStrides***

Let $C((i,j),r)$ be the minimum total memory required by an FST that uses exactly r expansion levels, covers levels i through j of the 1-bit trie, and each level of which requires at most $p = MMS(W-1,k)$ memory. A simple dynamic programming recurrence for C is:

$$C((i,j),r) = \min_{m \in X(i,j,r)} \{C((i,m),r-1) + nodes(m+1) * 2^{j-m}\}, r > 1, j-i+1 \geq r \quad (9)$$

$$C((i,j),1) = \begin{cases} nodes(i) * 2^{j-i+1} & \text{if } nodes(i) * 2^{j-i+1} \leq p \\ \infty & \text{otherwise} \end{cases} \quad (10)$$

```
Algorithm $FixedStrides(s, f, k, p)$
// $s$ and $f$ are the first and last levels of 1-bit trie, respectively.
// $k$ is number of expansion levels desired.
// $p$ is $MMS(W-1, k)$.
// Compute $C((s, f), *)$ and $M(*, *)$.
{
    for ($j = s; j \leq f; j++$){
        $C((s, j), 1) = nodes(s) * 2^{j-s+1}$;
        if ($C((s, j), 1) > p$) $C((s, j), 1) = \infty$;
        $M((s, j), 1) = s - 1$; }
    for ($r = 2; r \leq k; r++$)
        for ($j = s + r - 1; j \leq f; j++$){
        // Compute $C((s, j), r)$.
            $minCost = \infty; minL = \infty$;
            for ($m = j - 1; m \geq s + r - 2; m--$){
                if ($nodes(m+1) * 2^{j-m} \leq p$)
                    $cost = C((s, m), r-1) + nodes(m+1) * 2^{j-m}$;
                else break; // by Lemma 2
                if ($cost < minCost$) {
                    $minCost = cost$; $minL = m$;}}
            $C((s, j), r) = minCost$; $M((s, j), r) = minL$;}
}
```

Figure 8: Algorithm *FixedStrides*

where

$$X(i, j, r) = \{m | i + r - 2 \leq m < j \text{ and } nodes(m+1) * 2^{j-m} \leq p\}$$

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

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

in Equation 9.

For a given 1-bit trie, desired number of expansion levels k, and given per-stage memory constraint $MMS(W-1, k)$, we may compute $T(W-1, k)$ by invoking Algorithm *FixedStrides* (Figure 8) with the parameter tuple $(0, W-1, k, MMS(W-1, k))$. The complexity of this algorithm is $O(kW^2)$. Using the computed M values, the strides for the PFST that uses k expansion levels may be determined in an additional $O(k)$ time.

```
**Algorithm** $CheckUnique(p)$
// $p$ is $MMS(W-1,k)$.
// Return **true** iff the expansion level for p is unique.
{
        $count = 0$;
        **for** $(i = 0; i < W; i++)\{$
                $q = p/nodes(i)$;
                $r = p\%nodes(i)$; // % is mod operator
                **if** $(r > 0)$ **continue**;
                **if** ($q$ is a power of 2){
                        **if** $(count > 0)$ **return false**;
                        **else** $count++$; }}
        **return true**;
}
```

Figure 9: Algorithm to check whether expansion level for p is unique

4.2.2 Algorithm *PFST*

An alternative, and faster by a constant factor, algorithm to determine $T(W-1,k)$ results from the following observations.

O1: At least one level of the FST F that satisfies constraints C1-C3 requires exactly $MMS(W-1,k)$ memory.

O2: If there is exactly one i for which $MMS(W-1,k)/nodes(i) = 2^j$ for some j, then one of the levels of F must cover levels i through $i+j-1$ of the 1-bit trie.

We may use Algorithm *CheckUnique* (Figure 9) to determine whether there is exactly one i such that $MMS(W-1,k)/nodes(i)$ is power of 2. This algorithm can check whether q is a power of 2 by performing a binary search over the range 1 through $W-1$. The complexity of *CheckUnique* is $O(W \log W)$.

Suppose that $CheckUnique(MMS(W-1,k))$ is **true** and that $GreedyCover(k, MMS(W-1,k))$ covers the W levels of the 1-bit trie using $\#Stages$ stages and that stage $sMax$ is the unique stage that requires exactly $MMS(W-1,k)$ memory. Let this stage cover levels $lStart$ through $lFinish$ of the 1-bit trie. Note that the stages are numbered 0 through $\#Stages-1$. It is easy to see that in a k-PFST (i.e., a k-level FST that satisfies C1-C3),

1. One level covers levels $lStart$ through $lFinish$.

2. The k-PFST uses at least $sMax$ levels to cover levels 0 through $lStart - 1$ of the 1-bit trie (this follows from the correctness of $GreedyCover$; it isn't possible to cover these levels using fewer levels of the k-PFST without using more per-stage memory than $MMS(W-1,k)$). An additional FST level if required to cover levels $lStart$ through $lFinish$ of the 1-bit trie. So at most $k - sMax - 1$ levels are available for levels $lFinish + 1$ through $W - 1$ of the 1-bit trie.

3. The k-PFST uses at least $\#Stages - sMax - 1$ levels to cover levels $lFinish + 1$ through $W - 1$ of the 1-bit trie plus an additional stage for levels $lStart$ through $lFinish$. So at most $k + sMax - \#Stages$ levels are available for levels 0 through $lStart - 1$ of the 1-bit trie.

These observations lead to Algorithm *PFST* (Figure 10), which computes $T(W-1,k) = C((0, W-1), k)$. The complexity of this algorithm is $O(kW^2)$. Since the complexity of *FixedStrides* is $O(kW^2)$, it is cheaper to run this algorithm twice with small values of W than once with a W value (say) twice as large. We expect, in practice, that the cost of the two calls to *FixedStrides* made by *PFST* will be sufficiently small to make $PFST(W,k,p)$ faster than $FixedStrides(0, W-1, k, p)$. The algorithm *FullGreedyCover* used in *PFST* is *GreedyCover* enhanced to report the values $\#Stages$, $\#sMax$, $lStart$ and $lFinish$ as defined above.

Using the M values computed by *PFST*, the strides for the k-PFST may be determined in an additional $O(k)$ time.

4.2.3 Accelerating Algorithm $PFST$

When $CheckUnique$ returns **true**, Algorithm $PFST$ invokes $FixedStrides$ to compute $C((0, lStart - 1), j)$, $j \leq k + sMax - \#Stages$ and $C((lFinish + 1, W - 1), j)$, $j \leq k - sMax - 1$. When $k + sMax -$

```
**Algorithm** $PFST(W, k, p)$
// $W$ is length of longest prefix.
// $k$ is the number of expansion levels (i.e., pipeline stages) desired.
// $p = MMS(W - 1, k)$.
// Return $T(W - 1, k) = C((0, W - 1), k)$ and compute $M(*, *)$.
{
    **if** $(!CheckUnique(p))$ {
        // Compute $C((0, W - 1), k)$.
        $FixedStrides(0, W - 1, p, k)$;
        **return** $C((0, W - 1), k)$; }

    $FullGreedyCover(\#Stages, sMax, lStart, lFinish)$;

    // Compute $C((0, lStart - 1), j)$, $j \leq k + sMax - \#Stages$.
    $FixedStrides(0, lStart - 1, p, k + sMax - \#Stages)$;

    // Compute $C((lFinish + 1, W - 1), j)$, $j \leq k - sMax - 1$.
    $FixedStrides(lFinish + 1, W - 1, p, k - sMax - 1)$;

    // Determine $T(W - 1, k)$.
    $minCost = C((0, lStart - 1), sMax) + C((lFinish + 1, W - 1), k - sMax - 1)$;
    $minL = sMax$;
    **for** $(r = sMax + 1; r \leq k + sMax - \#Stages; r++)${
        $cost = C((0, lStart - 1), r) + C((lFinish + 1, W - 1), k - r - 1)$;
        **if** $(cost < minCost)$ {
            $minCost = cost; minL = r$; } }
    **return** $minCost + nodes(lStart) * 2^{lFinish - lStart + 1}$;
}
```

Figure 10: Algorithm $PFST$

$\#Stages \leq 3$ run time is reduced by invoking Algorithm $FixedStridesK123$ (Figure 11) instead. Similarly, when $k - sMax - 1 \leq 3$, faster run time results using $FixedStridesK123$ rather than $FixedStrides$.

5 A Faster 2-Phase Algorithm for $k = 2$ and $k = 3$

5.1 $k = 2$

The 2-phase algorithms of Section 4 determine the optimal pipelined FST in $O(kW^2)$ time. When $k = 2$, this optimal FST may be determined in $O(W)$ time by performing an exhaustive examination of the choices for the second expansion level $M(W-1,2)$ (note that the first expansion level is 0). Note that in the first **for** loop of Algorithm $FixedStridesK123$, we perform an exhaustive search over the choices for the second expansion level. This exhaustive search works under the constraint that no level of the FST take more than P memory. For the problem we are looking at in this section, $P = MMS(W-1,2)$ is unknown. $MMS(W-1,2)$ may be computed in $O(W \log W)$ time using parametric search. But such a computation would result in an overall complexity of $O(W \log W)$ rather than $O(W)$.

From Theorem 1 and Equation 4, we see that the search for $M(W-1,2)$ may be limited to the range $[\lceil (W-1)/2 \rceil, W-2]$. So for $k = 2$, Equation 4 becomes

$$\begin{aligned} MMS(W-1,2) &= \min_{m \in \{\lceil (W-1)/2 \rceil - 1..W-2\}} \{max\{MMS(m,1), nodes(m+1) * 2^{W-1-m}\}\} \\ &= \min_{m \in \{\lceil (W-1)/2 \rceil - 1..W-2\}} \{max\{2^{m+1}, nodes(m+1) * 2^{W-1-m}\}\} \end{aligned} \tag{11}$$

The run time for $k = 2$ may further be reduced to $O(\log W)$ (assuming that 2^i, $1 \leq i < W$ are precomputed) using the observations (a) 2^{m+1} is an increasing function of m and (b) $nodes(m+1) * 2^{W-1-m}$ is a non-increasing function of m (Lemma 1). Algorithm $PFSTK2$ (Figure 12) uses these observations to compute $MMS(W-1,2)$, $T(W-1,2)$, and $M(W-1,2)$ in $O(\log W)$ time. The stride for the root level of the PFST is $M(W-1,2)+1$ and that for the only other level of the PFST is $W - M(W-1,2) - 1$.

```
**Algorithm** $FixedStridesK123(s, f, k, p)$
// $s$ and $f$ are the first and last levels of 1-bit trie, respectively.
// $k \leq 3$ is maximum number of expansion levels desired.
// $p$ is the maximum per-stage memory.
// Compute $C((s, f), r)$, $1 \leq r \leq k$ and $M(*, *)$.
{
    if ($k == 1$) {
        $C((s, f), 1) = nodes(s) * 2^{f-s+1}; M((s, f), 1) = s - 1;$ }
    else if ($k == 2$) {
        $cost = \infty;$
        for ($i2 = f; i2 > s; i2 - -$){
            $p1 = nodes(s) * 2^{i2-s};$
            if ($p1 > p$) continue;
            $p2 = nodes(i2) * 2^{f-i2+1};$
            if ($p2 > p$) break; // by Lemma 2
            if ($p1 + p2 < cost$) {
                $cost = p1 + p2;$
                $C((s, f), 2) = cost; M((s, f), 2) = i2 - 1;$
                $C((s, i2 - 1), 1) = p1; M((s, i2 - 1), 1) = s - 1;$ } } }
    else { // $k == 3$
        $cost = \infty;$
        for ($i3 = f; i3 > s + 1; i3 - -$){
            $p3 = nodes(i3) * 2^{f-i3+1};$
            if ($p3 > p$) break; by Lemma 2
            for ($i2 = i3 - 1; i2 > s; i2 - -$){
                $p1 = nodes(s) * 2^{i2-s};$
                if ($p1 > p$) continue;
                $p2 = nodes(i2) * 2^{i3-i2};$
                if ($p2 > p$) break; by Lemma 2
                if ($p1 + p2 + p3 < cost$) {
                    $cost = p1 + p2 + p3;$
                    $C((s, f), 3) = cost; M((s, f), 3) = i3 - 1;$
                    $C((s, i3 - 1), 2) = p1 + p2; M((s, i3 - 1), 2) = i2 - 1;$
                    $C((s, i2 - 1), 1) = p1; M((s, i2 - 1), 1) = s - 1;$ } } } }
}
```

Figure 11: Optimal PFSTs for $k \leq 3$

```
Algorithm $PFSTK2()$
// Compute $MMS(W-1,2)$ and $T(W-1,2)$ values as well as optimal
// strides for two pipeline stages.
// $M(W-1,2)+1$ is the stride for the first pipeline stage.
{

Phase 1: [Determine $MMS(W-1,2)$]

    Step 1: [Determine $leastM$] Perform a binary search in the range $[\lceil (W-1)/2\rceil - 1, W-2]$ to
        determine the least $m$, $leastM$, for which $2^{m+1} > nodes(m+1) * 2^{W-1-m}$.
    Step 2: [Determine $MMS(W-1,2)$ and case] If there is no such $m$, set $MMS(W-1,2) =$
        $2 * nodes(W-1)$, $bestM = W-2$, and $case =$ "noM". Go to Phase 2.
        Set $X = nodes(leastM) * 2^{W-leastM}$.
        If $X < 2^{leastM+1}$, set $MMS(W-1,2) = X$, $bestM = leastM - 1$, and $case =$ "$X <$ ".
        If $X = 2^{leastM+1}$, set $MMS(W-1,2) = X$, $bestM = leastM - 1$, and $case =$ "$X =$ ".
        Otherwise, set $MMS(W-1,2) = 2^{leastM+1}$ and $case =$ "$X >$ ".

Phase 2: [Determine $M(W-1,2)$ and $T(W-1,2)$]

    $case =$ "$X >$ "
        Set $M(W-1,2) = leastM$ and $T(W-1,2) = 2^{leastM+1} + nodes(leastM+1) * 2^{W-1-leastM}$.
    Remaining Cases
        Use binary search in the range $[\lceil (W-1)/2\rceil - 1, bestM]$ to find the least $m$ for which $nodes(m+1) * 2^{W-1-m} = nodes(bestM+1) * 2^{W-1-bestM}$.
        Set $M(W-1,2)$ to be this value of $m$.
        Set $T(W-1,2) = 2^{m+1} + nodes(m+1) * 2^{W-1-m}$, where $m = M(W-1,2)$.
        Set $Y = 2^{leastM+1} + nodes(leastM+1) * 2^{W-1-leastM}$.
        If $case =$ "$X =$ " and $T(W-1,2) > Y$ then set $M(W-1,2) = leastM$ and $T(W-1,2) = Y$.

}
```

Figure 12: Algorithm to compute $MMS(W-1,2)$ and $T(W-1,2)$ using Equation 11

Although Algorithm $PFSTK2$ employs binary search to achieve the stated $O(\log W)$ asymptotic complexity, in practice, it may be more effective to use a serial search. This is so as the values being searched for are expected to be close to either the lower or upper end of the range being searched.

5.2 $k = 3$

The $O(\log W)$ phase 1 algorithm of Section 5.1 for the case $k = 2$ may be extended to an $O(\log^k W)$ algorithm to compute $MMS(W-1, k)$, $k \geq 2$. First consider the case $k = 3$. Applying Theorem 1 to Equation 4, results in

$$MMS(W-1,3) \quad = \min_{m \in \{\lceil (W-1)/2 \rceil - 1..W-2\}} \{max\{MMS(m,2), nodes(m+1) * 2^{W-1-m}\}\} \tag{12}$$

With respect to the two terms on the right side of Equation 12, we make the two observations (a) $MMS(m,2)$ is a non-decreasing function of m and (b) $nodes(m+1)*2^{W-1-m}$ is a non-increasing function of m (Lemma 1). Therefore, the binary search scheme of Phase 1 of Algorithm $PFSTK2$ (Figure 12) may be adapted to compute $MMS(W-1,3)$. The adaptation is given in Figure 13. This adaptation does $O(\log W)$ $MMS(m,2)$ computations at a total cost of $O(\log^2 W)$. It is easy to see that this adaptation generalizes to arbitrary k (replace all occurrences of $MMS(*,2)$ with $MMS(*,k-1)$ and of $MMS(*,3)$ with $MMS(*,k)$). The resulting complexity is $O(\log^k W)$. Hence, this generalization provides a faster way to compute $MMS(W-1,k)$ than the parametric search scheme of Section 4.1.3 for small k.

Unfortunately, we are unable to similarly extend the phase 2 algorithm of Section 5.1. When $k = 3$ we may determine the best PFST in $O(W^2)$ time using the value of $MMS(W-1,3)$ computed by the phase 1 algorithm of Figure 13. Although this is the same asymptotic complexity as achieved by the general algorithms of Section 4, we expect the new algorithm to be faster by a constant factor.

Let $T(j, 2, P)$ be the minimum total memory required by a 2-level FST for levels 0 through j of the

Step 1: [Determine $leastM$] Perform a binary search in the range $[\lceil (W-1)/2 \rceil - 1, W-2]$ to determine the least m, $leastM$, for which $MMS(m,2) > nodes(m+1) * 2^{W-1-m}$.

Step 2: [Determine $MMS(W-1,3)$ and case] If there is no such m, set $MMS(W-1,3) = 2 * nodes(W-1)$, $bestM = W-2$, and $case =$ "noM". Done.
Set $X = nodes(leastM) * 2^{W-leastM}$.
If $X < MMS(leastM, 2)$, set $MMS(W-1,3) = X$, $bestM = leastM - 1$, and $case =$ "$X <$ ".
If $X = MMS(leastM, 2)$, set $MMS(W-1,3) = X$ and $case =$ "$X =$ ".
Otherwise, set $MMS(W-1,3) = MMS(leastM, 2)$ and $case =$ "$X >$ ".

Figure 13: Compute $MMS(W-1,3)$

1-bit trie under the constraint that no level of the FST requires more than P memory. Let $M(j,2,P)+1$ be the stride of the root level of this FST. Note that $T(j,2,P)$ equals $C((0,j),2)$ as computed by $FixedStrides(0,j,2,P)$ and that $M(j,2,P)$ equals the corresponding $M((0,j),2)$. $T(j,2,P)$ may be determined in $O(W)$ time using the first **for** loop of $FixedStridesK123$. Let $mSmall$ and $mLarge$, respectively, be the smallest and largest m such that $1 \le m < j$ and $\max\{2^{m+1}, nodes(m+1)*2^{j-m}\} \le P$. From the **continue** and **break** statements, we see that the first **for** loop of $FixedStridesK123(0,j,2,P)$ computes $T(j,2,P) = C((0,j),2) = \min\{2^{m+1} + nodes(m+1) * 2^{j-m}\}$, where the min is taken over m in the range $[mSmall, mLarge]$. $M(j,2,P)$ is the value of m that yields this minimum. Notice that $T(j,2,P)$ is a nondecreasing function of j.

$T(W-1,3)$ and the expansion levels of the PFST may be computed using the algorithm of Figure 14. Its complexity is $O(W^2)$. Once again, in practice, better run-time performance may be observed using a serial rather than a binary search.

6 A Partitioning Scheme

We may reduce the MMS of the constructed pipelined tries by relaxing the restriction that the trie by an FST. Basu and Narlikar [1] propose a node pullup scheme that results in improved multibit tries that are not FSTs. In this section we propose a partitioning stratgey to construct pipelined multibit tries. In

Step 1: [Determine $mSmall$ and $mLarge$]

$case = "X > "$
Use binary search in the range $[leastM, W-2]$ to find the largest m, $mLarge$, for which $MMS(m,2) = MMS(leastM,2)$.
Set $mSmall = leastM$.

$case = "X < "or"noM"$
Use binary search in the range $[\lceil (W-1)/2 \rceil - 1, bestM]$ to find the least m, $mSmall$, for which $nodes(m+1) * 2^{W-1-m} = nodes(bestM+1) * 2^{W-1-bestM}$.
Set $mLarge = bestM$.

$case = "X = "$
Determine $mLarge$ as for the case $"X > "$ and $mSmall$ as for the case $"X < "or"noM"$.

Step 2: [Determine $T(W-1,3)$ and the expansion levels]
Compute $T(j, 2, MMS(W-1,3))$, $mSmall \le j \le mLarge$ and determine the j value, $jMin$, that minimizes $T(j,2,MMS(W-1,3)) + nodes(j+1) * 2^{W-1-j}$.
Set $T(W-1,3) = T(jMin, 2, MMS(W-1,3)) + nodes(jMin+1) * 2^{W-1-jMin}$.
The expansion levels are 0, $M(jMin, 2, MMS(W-1,3)) + 1$, and $jMin + 1$.

Figure 14: Compute $T(W-1,3)$ and expansion levels

the next section, we propose a modification to the node pullup scheme of [1]. In our partitioning scheme, we partition the prefixes in the router table into two groups AB and CD. Group AB comprises all prefix that begin with 0 or 10. These prefixes represent the traditional Class A and Class B networks. The remaining prefixes form the group CD. The multibit trie is constructed using the following steps.

Step 1: Find the optimal pipelined FST for the prefixes in group CD using any of the algorithms discussed earlier.

Step 2: Let $mms(CD)$ be the MMS for the FST of Step 1. Use a binary search scheme to determine $mms'(AB+CD)$. The binary search first tests $2 * mms(CD)$, $4 * mms(CD)$, $8 * mms(CD)$ and so on for feasibility and stops at the first feasible value, u, found. Then it does a standard binary search in the range $[mms(CD), u]$ to determine the smallest feasible value. The feasibility test is done using, as a heuristic, a modified version of *GreedyCover*, which doesn't change the FST for CD obtained in Step 1. The heuristic covers as many levels of the 1-bit trie for AB

as possible using at most $mms'(AB + CD) - 2^{s_0}$ space, where s_i is the stride for level i of the FST for CD. Then as many of the remaining levels of AB as possible are covered using at most $mms'(AB + CD) - nodes(CD, 1) * 2^{s_1}$ space, where $nodes(CD, 1)$ is the number of nodes at level 1 of the FST for CD, and so on.

Step 3: Merge the roots of the FSTs for AB and CD obtained as above into a single node. The stride of the merged root is $\max\{stride(AB), stride(CD)\}$, where $stride(AB)$ is the stride of the root of the FST for AB. If $stride(AB) > stride(CD)$ then the stride of the level 1 nodes of the CD FST is reduced by $stride(AB) - stride(CD)$. In practice, the new stride is ≥ 1 and so the stride reduction doesn't propogate further down the CD FST. When $stride(AB) < stride(CD)$, the stride reduction takes place in the AB FST. When $stride(AB) = stride(CD)$, there is no stride reduction in the lower levels of either FST.

Step 1 of the partitioning scheme takes $O(kW^2)$ time, Step 2 takes $O(W^2)$ time, and Step 3 takes $O(k)$ time (in practice, Step 3 takes $O(1)$ time). The overall complexity, therefore is, $O(kW^2)$.

7 Node Pullups

Basu and Narlikar [1] have used an optimization (called *node pullups*) aimed at spreading out the 24-bit prefixes in the multibit trie. The rationale for pullups is [1]:

- All 24-bit prefixes reside in a single level of the fixed-stride trie.
- Since most updates are to these 24-bit prefixes, a large fraction of the pipeline writes are directed to this single stage of the pipeline.
- By using node pullups, we can move many groups of neighboring 24-bit prefixes to lower levels of the multibit trie.

So, node pullups improve the update performance of the pipelined trie by spreading the 24-bit prefixes over several levels. "Let l be the level that contains the 24-bit prefixes. Consider a node in a level k above level l; say k terminates at bit position n (where $n < 24$). For some node in level k, if all of the 2^{24-n} possible 24-bit prefixes that can be descendents of this node are present in the trie, we pull all of them up into level k. The stride of the parent of the pulled-up node is increased by $24 - n$." Basu and Narlikar [1] perform node pullups in a top-down manner.

Although the node pullup scheme of [1] ensures that the total memory required by the transformed trie is no more than that of the original FST, this pullup scheme may result in a trie that has a larger MMS value (this, in fact, is the case for one of our 6 data sets of Section 8. Node-pullups may be done on the FST obtained using either a 1-phase or 2-phase FST construction algorithm. Node pullups are not done on the multibit trie that results from the partitioning scheme of Section 6.

Our node-pullup strategy is a bottom-up strategy. We start at the level just above the one that contains the 24-bit prefixes. A pullup is performed at a node iff that node has all possible descendents and the pullup doesn't result in an increase in MMS. When we are done with node pullups at the current level, we move to the next level (i.e., current level $- 1$).

8 Experimental Results

We programmed our dynamic programming algorithms in C and compared their performance against that of the algorithm of Basu and Narlikar [1]. All codes were run on a 2.26GHz Pentium 4 PC and were compiled using Microsoft Visual C++ 6.0 and optimization level `O2`. For test data, we used the six IPv4 prefix databases of Table 1. These databases correspond to backbone-router prefix sets. For our six databases, the number of nodes in the corresponding 1-bit trie is about $2n$, where n is the number of prefixes in the database (Table 1).

Database	Number of prefixes	Number of 16-bit prefixes	Number of 24-bit prefixes	Number of nodes
RRC04	109600	7000	62840	217749
RRC03b	108267	7069	62065	216462
RRC01	103555	6938	59854	206974
MW02	87618	6306	51509	174379
PA	85987	6606	49993	173626
ME02	70306	6496	39620	145868

Table 1: Prefix databases obtained from RIS [20] and IPMA project [17]. 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 $i + 1$.

8.1 FST Memory Comparison

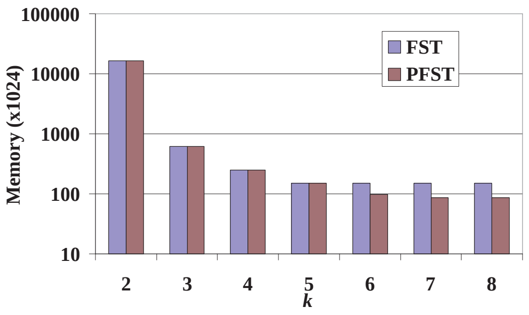
Table 2 shows the maximum per-stage memory, $MS(W-1,k,R)$, required by the k-level FST (k-FST) R for each of the six databases of Table 1. Rows labeled FST are for the case when R is the fixed stride trie that minimizes total memory ([26]), and rows labeled PFST are for the case when R is the fixed-stride trie for which $MS(W-1,k,R) = MMS(W-1,k)$. Figure 15(a) shows $MS(W-1,k,R)$ for the RRC01 data set. Note that the y-axis of this figure use a logarithmic scale. In 25 of the 42 tests (shown in boldface)[1] $MMS(W-1,k) = MS(W-1,k,PFST) < MS(W-1,k,FST)$. In the remaining 17 tests the FST that minimizes total memory also minimizes $MS(W-1,k,R)$. The maximum reduction in $MS(W-1,k,R)$ observed for our test set is approximately 44%.

Table 3 shows the total memory, $T(W-1,k)$, required by each of 3 k-level FSTs for each of our 6 data sets. The 3 FSTs are labeled FST (the FST that minimizes total memory [26]), PFST-1 (the pipelined FST generated using the 1-phase algorithm of [1]) and PFST-2 (the pipelined FST generated using any of our 2-phase algorithms). Note that PFST-1 and PFST-2 minimize $MS(W-1,k,R)$ and that the total memory required by the PFST generated by the various 2-phase algorithms proposed in this paper is the same. Figure 15(b) shows $T(W-1,k)$ for the RRC01 data set. As expected, whenever

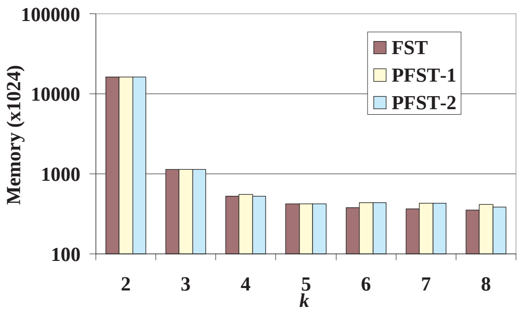
[1]Although the MW02, $k = 5$, data rounds to 128K for both FST and PFST, the unrounded data for PFST is less than that for FST.

k		2	3	4	5	6	7	8
RRC04	FST	16384	637	259	259	157	157	157
	PFST	16384	637	**256**	**157**	**137**	**100**	**92**
RRC03b	FST	16384	633	259	259	157	157	157
	PFST	16384	633	**256**	**157**	**152**	**100**	**91**
RRC01	FST	16384	616	249	151	151	151	151
	PFST	16384	616	249	151	**97**	**87**	**87**
MW02	FST	16384	512	210	128	128	128	128
	PFST	16384	512	210	**128**	**80**	**75**	**75**
PA	FST	16384	512	210	127	127	127	127
	PFST	16384	512	210	127	**81**	**74**	**74**
ME02	FST	16384	626	165	165	102	102	102
	PFST	**15248**	**512**	165	**118**	102	**66**	**64**

Table 2: Maximum memory required (KB) by any level of the k-level fixed-stride trie



(a) $MS(W-1,k,R)$



(b) $T(W-1,k)$

Figure 15: Maximum per-stage and total memory required (KB) for RRC01

$MS(W-1,k,PFST) < MS(W-1,k,FST)$, the $T(W-1,k)$s for PFST-1 and PFST-2 are greater than that for FST. That is the reduction in maximum per-stage memory comes with a cost of increased total memory required. The maximum increase observed on our data set is about 35% for PFST-1 and about 30% for PFST-2. Although $MS(W-1,k,\text{PFST-1}) = MS(W-1,k,\text{PFST-2}) = MMS(W-1,k)$, the total memory required by PFST-2 is less than that required by PFST-1 for 12 of the 42 test cases (shown in boldface). On several of these 12 cases, PFST-1 takes 7% more memory than required by PFST-2.

Tables 4 and 5 show the memory required by each stage of an 8-stage trie for each of our 6 data

k		2	3	4	5	6	7	8
RRC04	FST	16521	1286	661	539	436	393	381
	PFST-1	16521	1286	799	558	**571**	449	444
	PFST-2	16521	1286	799	558	**552**	449	444
RRC03b	FST	16536	1296	672	535	434	391	379
	PFST-1	16536	1296	812	571	**584**	446	441
	PFST-2	16536	1296	812	571	**565**	446	441
RRC01	FST	16399	1143	522	423	381	367	355
	PFST-1	16399	1143	**549**	423	435	429	**416**
	PFST-2	16399	1143	**522**	423	435	429	**387**
MW02	FST	16429	1035	477	397	352	311	301
	PFST-1	16429	1035	**510**	480	408	400	**360**
	PFST-2	16429	1035	**477**	480	408	400	**356**
PA	FST	16398	1012	448	366	321	308	298
	PFST-1	16398	1012	**481**	366	375	368	**355**
	PFST-2	16398	1012	**448**	366	375	368	**324**
ME02	FST	16502	1001	481	385	317	272	261
	PFST-1	23440	1015	**516**	**414**	**318**	327	**314**
	PFST-2	23440	1015	**481**	**413**	**317**	327	**283**

Table 3: Total memory required (KB) by k-level fixed-stride trie

sets. The stage that requires the maximum memory is shown in boldface. The row labeled PU-1 is for the trie that results using the node pullup scheme of [1] on PFST-1; row PU-1n is for our node pullup scheme (Section 7) applied to PFST-1; row PU-2n is for our node pullup scheme applied to PFST-2, and row PART is for the trie that results using our partitioning scheme (Section 6). Figure 16 plots this data for the RRC01 data set. For all 6 of our databases, our partitioning scheme generates tries with the least per-stage memory requirement. The pullup scheme of [1] increased the MS value for one of our data sets (ME02); other than this anomalous case, the MS value was consistently decreased by the application of node pullups using either the scheme of [1] or ours. The MS reduction obtained by our partitioning scheme was accompanied by a significant increase in total memory required. For the RRC01 data set, for example, PART takes almost 17% more total memory than does PU-2n while reducing the maximum per-stage memory requirement by about 4%. Of the algorithms developed for pipelined tries, PU-2n generated tries with least total memory requirement for all 6 test sets (it tied with one or more

Stage		0	1	2	3	4	5	6	7	Total
RRC04	FST	2048	19968	25708	63852	102344	**161008**	8752	6288	389968
	PFST-1	65536	51416	81592	66370	80504	**93788**	8752	6288	454246
	PFST-2	65536	51416	81592	66370	80504	**93788**	8752	6288	454246
	PU-1	65536	51416	**84028**	78976	83222	49712	8752	6288	427030
	PU-1n	65536	51416	**84028**	78976	83222	49712	8752	6288	427030
	PU-2n	65536	51416	**84028**	78976	83222	49712	8752	6288	427030
	PART	65536	51416	**81592**	80488	81310	79590	7936	4960	452828
RRC03b	FST	2048	19904	25280	63244	102096	**160748**	9712	4864	387896
	PFST-1	65536	50560	80984	66282	80374	**93174**	9712	4864	451486
	PFST-2	65536	50560	80984	66282	80374	**93174**	9712	4864	451486
	PU-1	65536	50560	**83224**	77534	83622	49846	9712	4864	424898
	PU-1n	65536	50560	**83224**	77534	83622	49846	9712	4864	424898
	PU-2n	65536	50560	**83224**	77534	83622	49846	9712	4864	424898
	PART	65536	68384	71656	80480	**81214**	80048	7888	3200	458406
RRC01	FST	2048	19840	24812	61964	98816	**154156**	976	512	363124
	PFST-1	65536	49624	78784	63822	77078	**89152**	976	512	425484
	PFST-2	2048	19840	49624	78784	63822	77078	**89152**	15616	395964
	PU-1	65536	49624	**81024**	74624	80174	47328	976	512	399798
	PU-1n	65536	49624	**81024**	74624	80174	47328	976	512	399798
	PU-2n	2048	19840	49624	**81024**	74624	80174	47328	15616	370278
	PART	66536	66736	69798	76992	**77768**	74720	928	496	432974

Table 4: Memory required by each stage of an 8-level multibit trie for RRC04, RRC03b, and RRC01 databases

other algorithms in 2 cases).

8.2 Execution Time Comparison

First we compare the execution time of the 1-phase algorithm of [1] and that of our reduced-range 1-phase algorithm (Section 3). Note that both algorithms generate identical FSTs. Table 6 shows the times for the two algorithms for k in the range [2,8]. Figure 17 plots this data for the RRC04 data set. As can be seen, our 1-phase algorithm achieves a speedup, relative to the 1-phase algorithm of [1], of about 2.5 when $k = 2$ and about 3 when $k = 8$.

Table 7 shows the time taken by two–dynamic programming (DP, Section 4.1.1) and binary search (BP, Section 4.1.2)–of our proposed algorithms to compute $MMS(W - 1, k)$. Figure 18 plots this data for the RRC04 data set. For $k \leq 3$, our dynamic programming algorithm is faster. When $k \geq 4$, our

Stage		0	1	2	3	4	5	6	7	Total
	FST	2048	17984	20272	49756	81892	**131548**	2896	1552	307948
	PFST-1	65536	40544	61180	53710	65774	**76946**	2896	1552	368138
	PFST-2	2048	17984	40544	61180	53710	65774	**76946**	46336	364522
MW02	PU-1	65536	40544	63364	63196	**68302**	39938	2896	1552	345328
	PU-1n	65536	40544	63364	63196	**68302**	39938	2896	1552	345328
	PU-2n	2048	17984	40544	63364	63196	**68302**	39938	46336	341712
	PART	66536	50880	62980	63714	64666	**67476**	992	160	376404
	FST	2048	17952	20468	50652	82552	**130492**	432	288	304884
	PFST-1	65536	40936	62220	53852	65246	**75288**	432	288	363798
	PFST-2	2048	17952	40936	62220	53852	65246	**75288**	13824	331366
PA	PU-1	65536	40936	64068	62408	**68470**	40392	432	288	342530
	PU-1n	65536	40936	64068	62408	**68470**	40392	432	288	342530
	PU-2n	2048	17952	40936	64068	62408	**68470**	40392	13824	310098
	PART	66536	51544	63684	62600	59836	**67058**	352	224	370834
	FST	2048	16960	34008	98440	**104300**	3784	6824	892	267256
	PFST-1	32768	47504	40080	**65176**	52150	60992	7568	15152	321390
	PFST-2	4096	31360	53496	**65176**	52150	60992	7568	15152	289990
ME02	PU-1	32768	47504	40380	**67868**	62230	32800	7568	15152	306270
	PU-1n	32768	47504	40080	**65176**	65174	34944	7568	15152	308366
	PU-2n	4096	31360	53496	**65176**	65174	34944	7568	15152	276966
	PART	32768	48928	51960	51546	52576	**56508**	40544	8656	343486

Table 5: Memory required by each stage of an 8-level multibit trie for MW02, PA, and ME02 databases

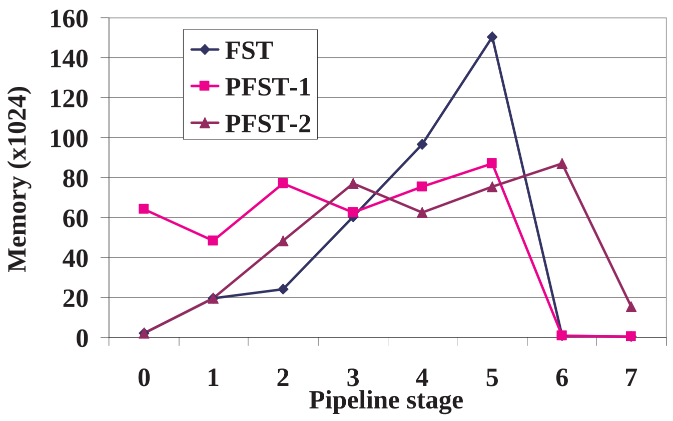


Figure 16: Memory required in each stage of an 8-stage pipeline for RRC01

k		RRC04	RRC03b	RRC01	MW02	PA	ME02
2	[1]	4.95	5.02	5.02	5.03	4.96	5.18
	Our	2.03	2.03	2.04	2.06	2.07	2.08
3	[1]	8.94	8.96	8.83	8.87	8.88	8.97
	Our	3.09	3.08	3.14	3.14	3.09	3.14
4	[1]	12.46	12.47	12.26	12.49	12.48	12.84
	Our	4.15	4.11	4.16	4.14	4.16	4.12
5	[1]	16.06	15.92	15.84	16.00	15.94	16.33
	Our	5.17	5.14	5.15	5.19	5.11	5.19
6	[1]	19.00	19.17	19.20	19.05	19.07	19.37
	Our	6.24	6.12	6.17	6.20	6.10	6.20
7	[1]	21.81	22.12	22.36	21.93	21.95	22.18
	Our	7.11	7.04	7.10	7.13	7.02	7.10
8	[1]	25.07	25.12	24.96	25.02	24.79	24.73
	Our	8.08	8.00	8.03	8.06	8.00	7.93

Table 6: Execution time (in μsec) for the 1-phase algorithm of [1] and our 1-phase algorithm

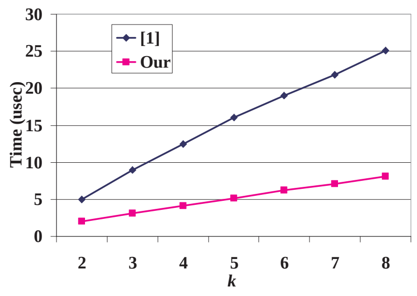


Figure 17: Execution time (in μsec) for the 1-phase algorithm of [1] and our 1-phase algorithm for RRC04

k		RRC04	RRC03b	RRC01	MW02	PA	ME02
2	DP	1.63	1.63	1.65	1.67	1.72	1.72
	BS	4.38	4.13	4.26	4.18	4.32	4.22
3	DP	2.55	2.52	2.58	2.63	2.66	2.68
	BS	3.28	3.17	3.32	3.14	3.06	3.48
4	DP	3.46	3.45	3.47	3.51	3.57	3.52
	BS	3.43	3.23	3.13	3.40	3.35	3.24
5	DP	4.21	4.21	4.24	4.27	4.32	4.34
	BS	2.98	3.04	3.00	3.00	2.95	3.16
6	DP	5.04	5.03	5.06	5.06	5.13	5.23
	BS	3.56	3.55	3.28	3.29	3.36	3.45
7	DP	5.78	5.76	5.80	5.81	5.86	5.93
	BS	3.61	3.70	0.28	0.29	0.27	3.55
8	DP	6.50	6.45	6.53	6.57	6.58	6.59
	BS	0.29	0.28	0.27	0.29	0.27	3.66

Table 7: Execution time (in μsec) for computing $MMS(W-1,k)$

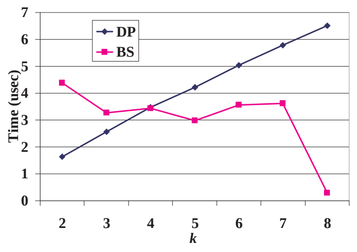


Figure 18: Execution time for computing $MMS(W-1,k)$ for RRC04

binary search algorithm is faster. In fact when $k \geq 7$, binary search took about 1/20th the time on some of our data sets. The run time for our binary search algorithm is quite small for most of our data sets when $k = 7$ or 8. This is because $GreedyCover(k,p)$ is true, in the first line of *BinarySearchMMS* (Figure 7), for these combinations of data set and k.

Table 8 gives the execution times for our three algorithms of Section 4.2 to compute $T(W-1,k)$. The algorithms of Sections 4.2.1, 4.2.2 and 4.2.3 are, respectively, denoted by FS, PFST, and K123 in Table 8. Figure 19 plots this data for the RRC04 data set. Algorithm K123 is the fastest of our 3 algorithms. On RRC04 with $k = 5$, FS took about 4.6 as much time as taken by K123 and PFST took 2.9 times as much time as taken by K123.

k		RRC04	RRC03b	RRC01	MW02	PA	ME02
2	FS	3.17	3.12	3.27	3.28	3.28	3.29
	PFST	3.12	3.14	3.18	3.18	3.24	1.52
	K123	3.11	3.08	3.09	3.16	3.15	1.31
3	FS	4.30	4.27	4.26	4.28	4.26	4.46
	PFST	1.71	1.63	1.77	4.31	4.40	4.56
	K123	1.41	1.38	1.51	4.35	4.38	4.58
4	FS	5.56	5.53	5.73	5.69	5.61	5.62
	PFST	5.53	5.52	3.02	2.92	3.05	2.91
	K123	5.54	5.52	1.50	1.41	1.61	1.34
5	FS	6.50	6.51	6.50	6.63	6.67	6.72
	PFST	4.11	4.10	4.05	6.78	4.25	5.40
	K123	1.41	1.41	1.41	6.78	1.62	5.34
6	FS	7.64	7.65	7.44	7.46	7.47	8.00
	PFST	6.19	6.26	2.62	3.60	3.74	4.42
	K123	6.22	6.23	2.61	1.60	1.86	1.62
7	FS	8.62	8.63	8.41	8.40	8.38	8.61
	PFST	3.04	3.87	6.08	5.93	6.15	3.07
	K123	2.99	2.59	5.99	5.67	6.01	3.02
8	FS	9.33	9.40	9.39	9.36	9.32	9.33
	PFST	6.10	6.11	7.12	6.96	7.20	4.90
	K123	5.89	5.85	6.86	6.76	7.07	2.79

Table 8: Execution time (in μsec) for computing $T(W-1,k)$

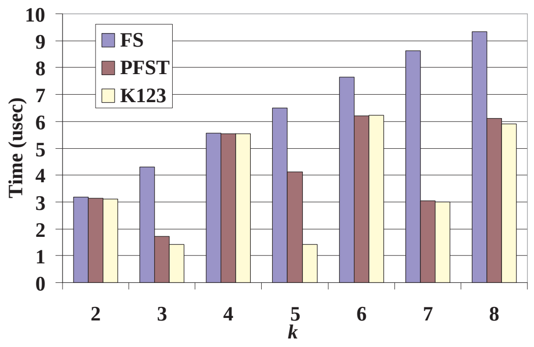


Figure 19: Execution time for computing $T(W-1,k)$ for RRC04

k	RRC04	RRC03b	RRC01	MW02	PA	ME02
2	0.13	0.12	0.13	0.13	0.13	0.13
3	0.76	0.76	0.76	0.76	0.76	0.75

Table 9: Execution time (in μsec) for $k = 2$ and $k = 3$ algorithms of Section 5

The run times for our customized = 2 and $k = 3$ algorithms of Section 5 are given in Table 9. On RRC04 with $k = 2$, K123 takes 24 times as much time as taken by the $k = 2$ algorithm of Section 5.1. For $k = 3$, K123 took 1.9 times the time taken by the $k = 3$ algorithm of Section 5.2. Note that K123 determines $T(W-1,k)$ only and must be preceded by a computation of $MMS(W-1,k)$. However, the $k = 2$ and $k = 3$ algorithms of Section 5 determine both $MMS(W-1,k)$ and $T(W-1,k)$.

To compare the run time performance of our algorithm with that of [1], we use the times for implementation of faster $k = 2$ and $k = 3$ algorithms when $k = 2$ or 3 and the times for the faster of implementations of our binary search algorithms when $k > 3$. That is, we compare our best times with the times for the algorithm of [1].

Figure 20 plots the run time of the 1-phase algorithm of [1] and that of our composite algorithm, which uses the $k = 2$ and $k = 3$ algorithms of Section 5 when $2 \leq k \leq 3$. When $k > 3$, our composite algorithm uses binary search to compute $MMS(W-1,k)$ and $FixedStridesK123$ to compute $T(W-1,k)$. The plot of Figure 20 is for our largest database RRC04. Our new algorithm takes less than 1/38th the time taken by the algorithm of [1] when $k = 2$ and less than 1/4th the time when $k = 8$.

9 Conclusion

We have developed a 1-phase algorithm for pipelined fixed-stride tries that computes the same tries as computed by the algorithm of [1]. Our algorithm is, however, faster by a factor between 2.5 and 3 on publically available router tables. We have shown that although the algorithm of [1] determines FSTs that minimize the maximum per-stage memory, the generated FSTs may not minimize total memory under the constraint that maximum per-stage memory is minimized. We have proposed several 2-

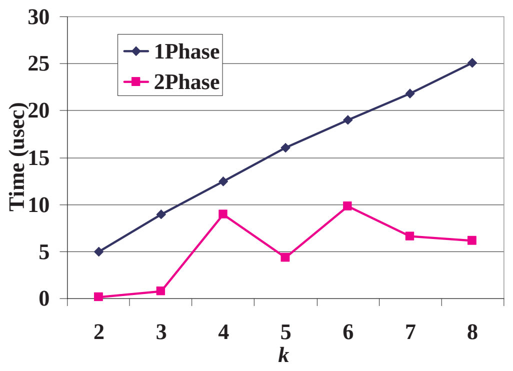


Figure 20: Execution time for our composite 2-phase algorithm and the 1-phase algorithm of Basu and Narlikar [1] for RRC04

phase strategies to correctly determine fixed stride tries that minimize total memory subject to the constraint that maximum per-stage memory is minimized. Our 2-phase algorithms not only generate better pipelined FSTs than generated by the 1-phase algorithm of [1] but also are considerably faster. On our largest database RRC04, our 2-phase composite algorithm took 1/38th the time taken by the algorithm of [1] when $k = 2$ and less than 1/4th the time when $k = 8$.

Additionally, we have proposed a node pullup scheme that guarantees that the maximum per-stage memory is not increased; the pullup scheme of [1] does not make this guarantee (in fact, on one of our test sets, the pullup scheme of [1] increased the maximum per-stage memory). We also have proposed a partitioning heuristic that generated pipelined multibit tries with smaller maximum per-stage memory than required by the best tries produced using any other scheme.

References

[1] A. Basu and G. Narlikar, Fast incremental updates for pipelined forwarding engines, *IEEE INFOCOM*, 2003.

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

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

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

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

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

[7] G. Frederickson, Optimal algorithms for tree partitioning, *Second ACM-SIAM Symposium on Discrete Algorithms*, 1991, 168-177.

[8] G. Frederickson, Optimal parametric search algorithms in trees I: Tree partitioning, Technical Report CS-TR-1029, Purdue University, West Lafayette, IN, 1992.

[9] G. Frederickson and D. Johnson, Finding kth paths and p-centers by generating and searching good data structures, *Jr. of Algorithms*, 4, 1983, 61-80.

[10] G. Frederickson and D. Johnson, Generalized selection and ranking: Sorted matrices, *SIAM Jr. of Computing*, 13, 1984, 14-30.

[11] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of data structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[12] E. Horowitz, S. Sahni, and S. Rajasekaran, Computer Algorithms, W.H. Freeman, NY, 1997, 769 pages.

[13] K. Kim and S. Sahni, IP lookup by binary search on length, *Journal of Interconnection Networks*, 3, 2002, 105-128.

[14] B. Lampson, V. Srinivasan, and G. Varghese, IP Lookup using multi-way and multicolumn search, *IEEE INFOCOM*, 1998.

[15] H. Lu and S. Sahni, O(log n) dynamic router-tables for ranges. *IEEE Symposium on Computers and Communications*, 2003, 91-96.

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

[17] Merit, IPMA statistics, http://nic.merit.edu/ipma, 2003.

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

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

[20] Ris, Routing information service raw data, http://data.ris.ripe.net, 2003.

[21] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[22] S. Sahni and K. Kim, Efficient construction of multibit tries for IP lookup, *IEEE/ACM Transactions on Networking*, 11, 4, 2003, pp. 650-662.

[23] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[24] S. Sahni and K. Kim, O(log n) dynamic router-table design, *IEEE Transactions on Computers*, to appear.

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

[26] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.

[27] S. Suri, G. Varghese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *GLOBECOM 2001*.

[28] V. Thanvantri and S. Sahni, Optimal folding of standard and custom cells, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, No. 1, 1996, 123-143.

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