One-Dimensional Packet Classification Using Pipelined Multibit Tries *

Wencheng Lu and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{wlu, sahni}@cise.ufl.edu

October 14, 2005

Abstract

We propose a heuristic for the construction of variable-stride multibit tries. These multibit tries are suitable for one-dimensional packet classification using a pipelined architecture. The variable-stride tries constructed by our heuristic require significantly less per-stage memory than required by optimal pipelined fixed-stride tries. We also develop a tree packing heuristic, which dramatically reduces per-stage memory required by fixed- and variable-stride multibit tries constructed for pipelined architecture.

Keywords
Packet classification, longest matching prefix, controlled prefix expansion, fixed-stride tries, variable-stride tries, dynamic programming.

1 Introduction

Internet packets are classified into flows based on their header fields. This classification is done using a table of rules in which each rule is a pair (F, A), where F is a filter and A is an action. If an incoming packet matches a filter in the rule table, the associated action specifies what is to be done with this packet. Typical actions include packet forwarding and dropping. A d-dimensional filter F is a d-tuple $(F[1], F[2], \cdots, F[d])$, where $F[i]$ is a range that specifies destination addresses, source addresses, port numbers, protocol types, TCP flags, etc. A packet is said to match filter F, if its header field values fall in the ranges $F[1], \cdots, F[d]$. Since it is possible for a packet to match more than one of the filters in a classifier, a tie breaker is used to determine a unique matching filter.

In one-dimensional packet classification (i.e., $d = 1$), $F[1]$ is usually specified as a destination address prefix and lookup involves finding the longest prefix that matches the packet's destination address. Data structures for longest-prefix matching have been extensively studied (see [2, 3], for surveys). In this paper, we are concerned solely with 1-dimensional packet classification. It should be noted that data structures for 1-dimensional packet classification are fundamental to the design and development of multi-dimensional packet classification (i.e., $d > 1$), since data structures for multi-dimensional packet classification are usually built on top of those for 1-dimensional packet classification.

In this paper we focus on the development of data structures suitable for ASIC-based pipelined architectures for high speed packet classification. Basu and Narlikar [7] and Kim and Sahni [6] have proposed algorithms for the

*This research was supported, in part, by the National Science Foundation under grant ITR-0326155

construction of optimal fixed-stride tries for one-dimensional prefix tables; these fixed-stride tries are optimized for pipelined architectures. Basu and Narliker [7] list three constraints for optimal pipelined fixed-stride multibit tries:

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 minimized.

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

Basu and Narliker [7] assert that constraint C3 reduces pipeline disruption resulting from rule-table updates. Although the algorithm proposed in [7] constructs fixed-stride tries that satisfy constraints C1 and C2, the constructed tries may violate constraint C3. Kim and Sahni [6] have developed faster algorithms to construct pipelined fixed-stride tries; their tries satisfy all three of the constraints C1–C3. FSTs that satisfy C1–C3 are called *optimal pipelined FSTs*.

In this paper, we propose heuristics for the construction of pipelined variable-stride multibit tries. The pipelined tries constructed by our algorithms are compared, experimentally, to those constructed by the algorithms of Kim and Sahni [6]. The variable-stride multibit tries constructed by our heuristic require significantly less per-stage memory than required by optimal pipelined fixed-stride tries. Also, we develop a tree packing heuristic, which dramatically reduces per-stage memory required by pipelined fixed- and variable-stride multibit tries.

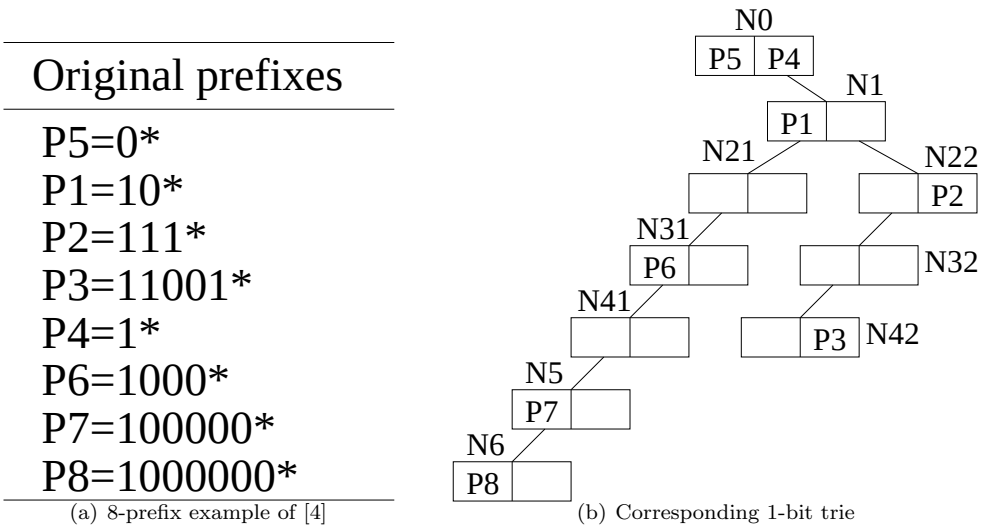
We begin, in Section 2, by reviewing basic concepts related to the trie data structure. This section also describes multibit fixed- and variable-stride tries, and controlled prefix expansion [4]. In Section 3 we develop an (heuristic) algorithm for pipelined variable-stride tries. In Section 4 we develop a tree packing (heuristic) algorithm for mapping multibit tries to a pipelined architecture. An experimental evaluation of our algorithms is conducted in Section 5.

2 Tries

2.1 1-bit Tries

A *1-bit trie* is a binary tree-like structure in which each node has two element fields, *le* (left element) and *re* (right element) and each element field has the components *child* and *data*. Branching is done based on the bits in the search key. A left-element child branch is followed at a node at level i (the root is at level 0) if the ith bit of the search key is 0; otherwise a right-element child branch is followed. Level i nodes store prefixes whose length is $i+1$ in their data fields. A prefix that ends in 0 is stored as $le.data$ and one whose last bit is a 1 is stored as $re.data$. The node in which a prefix is to be stored is determined by doing a search using that prefix as key. Let N be a node in a 1-bit trie and let E be an element field (either left or right) of N. Let $Q(E)$ be the bit string defined by the path from the root to N followed by a 0 in case E is a left element field and 1 otherwise. $Q(E)$ is the prefix that corresponds to E. $Q(E)$ is stored in $E.data$ in case $Q(E)$ is one of the prefixes to be stored in the trie.

Figure 1 shows a set of 8 prefixes and the corresponding 1-bit trie. The $*$ shown at the right end of each prefix is used neither for the branching described above nor in the length computation. So, the length of $P1$ is 2.



(a) 8-prefix example of [4] (b) Corresponding 1-bit trie

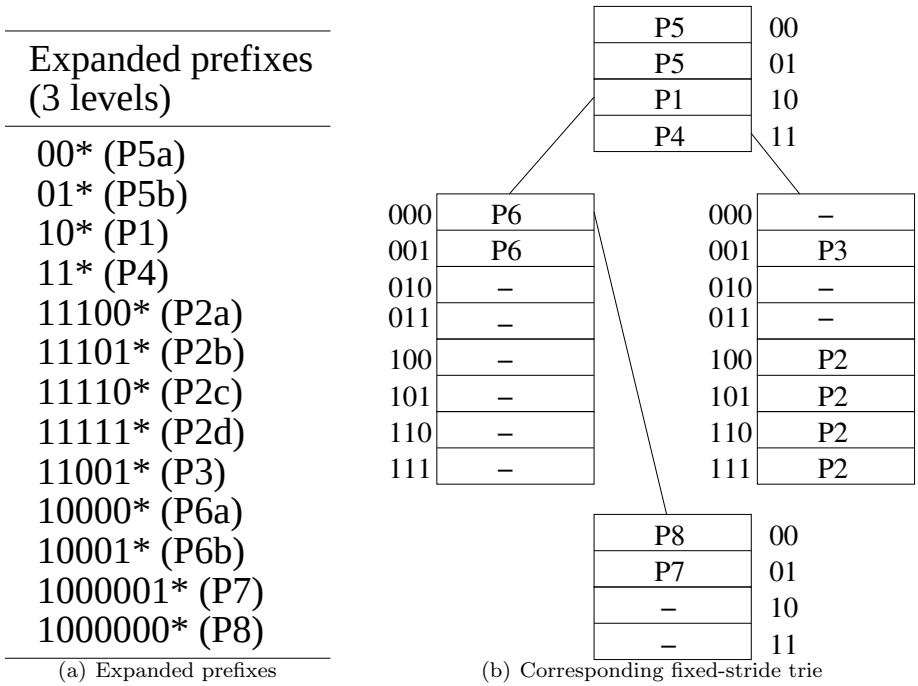
Figure 1: Prefixes and corresponding 1-bit trie [5]

2.2 Multibit Tries

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 element fields (corresponding to the 2^s possible values for the s bits that are used). Each element field has a data and a child component. A node whose stride is s requires 2^s memory units (one memory unit being large enough to accomodate an element field). Note that the stride of every node in a 1-bit trie is 1.

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 *variable-stride trie* (VST), nodes may have different strides regardless of their level.

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 [4]. For example, P5 = 0* is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, dominance rules are used to eliminate all but one occurrence of the prefix. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 2(a) shows the prefixes that result when we expand the prefixes of Figure 1 to lengths 2, 5, and 7. Duplicate prefixes, following expansion, are eliminated by favoring longer length original prefixes when longest length prefix matching is desired. Figure 2(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.



(a) Expanded prefixes (b) Corresponding fixed-stride trie

Figure 2: Prefix expansion and fixed-stride trie

Since the trie of Figure 2(b) can be searched with at most 3 memory accesses, it represents a time-performance improvement over the 1-bit trie of Figure 1(b), which requires up to 7 memory accesses to perform a search. However, the space requirements of the FST of Figure 2(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 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. Note that the 1-bit trie of Figure 1 requires only 20 memory units.

Let N be a node at level j of a multibit trie. Let s_0, s_1, $\cdots$, s_j be the strides of the nodes on the path from the root of the multibit trie to node N. Note that s_0 is the stride of the root and s_j is the stride of N. With node N we associate a pair $[s, e]$, called the *start-end* pair, that gives the start and end levels of the corresponding 1-bit trie O covered by this node. By definition, $s = \sum_{i=0}^{j-1} s_j$ and $e = s + s_j - 1$. The root of the multibit trie *covers* levels 0 through $s_0 - 1$ of O and node N covers levels 0 through $s_j - 1$ of a corresponding subtrie of O. In the case of an FST all nodes at the same level of the FST have the same $[s, e]$ values. In the FST of Figure 2(b), the $[s, e]$ values for the level 0, level 1, and level 2 nodes are $[0, 1]$, $[2, 4]$ and $[5, 6]$, respectively. Level 0 of the FST covers levels 0 and 1 of the corresponding 1-bit trie while level 2 of this FST covers levels 2, 3, and 4 of the 1-bit trie. Levels 0 and 1 of the FST together cover levels 0 through 4 of the 1-bit trie.

Starting with a 1-bit trie for n prefixes whose length is at most W, the strides for a space-optimal FST with at most k levels may be determined in $O(nW + kW^2)$ time[1] [4, 5]. For a space-optimal VST whose height[2] is

[1]The complexity of $O(kW^2)$ given in [4, 5] assumes we start with data extracted from the 1-bit trie; the extraction of this data takes $O(nW)$ time.

[2]The height of a trie is the number of levels in the trie. Height is often defined to be 1 less than the number of levels. However, the definition we use is more convenient in this paper.

constrained to k, the strides may be determined in $O(nW^2k)$ time [4, 5].

3 Pipelined Variable-stride Multibit Tries

Algorithms to construct optimal pipelined FSTs (i.e., FSTs that satisfy constraints C1–C3) have been developed in [6]. So, we consider only the construction of pipelined VSTs in this section. Optimal pipelined VSTs satisfy constraints C1–C3 (although, in C1, we replace "fixed-stride trie" with "variable-stride trie"). Although we do not develop an algorithm that constructs an optimal pipelined VST, the heuristic proposed by us, in this section, constructs an "approximately optimal" VST, which when mapped onto a pipelined architecture using constraint C1 results in a maximum per-stage memory requirement that is considerably less than that for an optimal pipelined FST for the given rule table.

Let O be the 1-bit trie for the given filter set, let N be a node of O and let $ST(N)$ be the subtree of O that is rooted at N. Let $Opt'(N,r)$ denote the approximately optimal (pipelined) VST for the subtree $ST(N)$; this VST has at most r levels. Let $Opt'(N,r).E(l)$ be the (total) number of elements at level l of $Opt'(N,r)$, $0 \leq l < r$. We seek to construct $Opt'(root(O),k)$, the approximately optimal pipelined VST for O that has at most k levels, where k is the number of available pipeline stages.

Let $D_i(N)$ denote the descendents of N that are at level i of $ST(N)$. So, for example, $D_0(N) = \{N\}$ and $D_1(N)$ denotes the children, in O, of N. Our approximately optimal VST has the property that its subtrees are approximately optimal for the subtrees of N that they represent. So, for example, if the root of $Opt'(N,r)$ represents levels 0 through $i-1$ of $ST(N)$, then the subtrees of (the root of) $Opt'(N,r)$ are $Opt'(M,r-1)$ for $M \in D_i(N)$.

When $r = 1$, $OPT'(N,1)$ has only a root node; this root represents all levels of $ST(N)$. So,

$$Opt'(N,1).E(l) = \begin{cases} 2^{height(N)} & l = 0 \\ 0 & l > 0 \end{cases} \tag{1}$$

where $height(N)$ is the height of $ST(N)$.

When $r > 1$, the number, q, of levels of $ST(N)$ represented by the root of $Opt'(N,r)$ is between 1 and $height(N)$. From the definition of $Opt'(N,r)$, it follows that

$$Opt'(N,r).E(l) = \begin{cases} 2^q & l = 0 \\ \sum_{M \in D_q(N)} Opt'(M,r-1).E(l-1) & 1 \leq l < r \end{cases} \tag{2}$$

where q is as defined below

$$q = argmin_{1 \leq i \leq height(N)}\{\max\{2^i, \max_{0 \leq l < r-1}\{\sum_{M \in D_i(N)} Opt'(M,r-1).E(l)\}\}\} \tag{3}$$

Although the dynamic programming recurrences of Equations 1–3 may be solved directly to determine $Opt'(root(O),k)$, the time complexity of the resulting algorithm is reduced by defining auxialliary equations. For this purpose, let $Opt'STs(N,i,r-1)$, $i > 0$, $r > 1$, denote the set of approximately optimal VSTs for $D_i(N)$ ($Opt'STs(N,i,r-1)$

has one VST for each member of $D_i(N)$); each VST has at most $r-1$ levels. Let $Opt'STs(N,i,r-1).E(l)$ be the sum of the number of elements at level l of each VST of $Opt'STs(N,i,r-1)$. So,

$$Opt'STs(N,i,r-1).E(l) = \sum_{M \in D_i(N)} Opt'(M,r-1).E(l), 0 \leq l < r-1 \quad (4)$$

For $Opt'STs(N,i,r-1).E(l)$, $i>0$, $r>1$, $0 \leq l < r-1$, we obtain the following recurrence

$$Opt'STs(N,i,r-1).E(l) = \begin{cases} Opt'(LC(N),r-1).E(l) + Opt'(RC(N),r-1).E(l) & i=1 \\ Opt'STs(LC(N),i-1,r-1).E(l) + Opt'STs(RC(N),i-1,r-1).E(l) & i>1 \end{cases} \quad (5)$$

where $LC(N)$ and $RC(N)$, respectively, are the left and right children, in $ST(N)$, of N.

Since the number of nodes in O is $O(nW)$, the total number of $Opt'(N,r)$ and $Opt'STs(N,i,r-1)$ values is $O(nW^2k)$. For each, $O(k)$ $E(l)$ values are computed. Hence, to compute all $Opt'(root(O),k).E(l)$ values, we must compute $O(nW^2k^2)$ $Opt'(N,r).E(l)$ and $Opt'STs(N,i,r-1).E(l)$ values. Using Equations 1–5, the total time for this is $O(nW^2k^2)$.

4 Mapping Onto A Pipeline Architecture

When the approximately optimal VST $Opt'(root(O),k)$ of Section 3 is mapped onto a k stage pipeline in the most straightforward way (i.e., nodes at level l of the VST are packed into stage $l+1$, $0 \leq l < k$ of the pipeline), the maximum per-stage memory is

$$\max_{0 \leq l < k} \{Opt'(root(O),k).E(l)\}$$

We can do quite a bit better than this by employing a more sophisticated mapping strategy. For correct pipeline operation, we need require only that if a node N of the VST is assigned to stage q of the pipeline, then each descendent of N be assigned to a stage r such that $r > q$. Hence, we are motivated to solve the following tree packing problem:

Tree Packing (TP)

Input: Two integers $k>0$ and $M>0$ and a tree T, each of whose nodes has a positive size.

Output: "Yes" iff the nodes of T can be packed into k bins, each of capacity M. The bins are indexed 1 through k and the packing is constrained so that for every node packed into bin q, each of its descendent nodes is packed into a bin with index more than q.

By performing a binary (or other) search over M, we may use an algorithm for TP to determine an optimal packing (i.e., one with least M) of $Opt'(root(O),k)$ into a k-stage pipeline. Unfortunately, problem TP is NP-complete. This may be shown by using a reduction from the partition problem [1]. In the partition problem, we

are given n positive integers s_i, $1 \leq i \leq n$ whose sum is $2B$ and we are to detemine whether any subset of the given s_is sums to B.

Theorem 1 *TP is NP-complete.*

Proof It is easy to see that TP is in NP. So we simply show the reduction from the partition problem. Let n, s_i, $1 \leq i \leq n$, and B ($\sum s_i = 2B$) be an instance of the partition problem. We may transform, in polynomial time, this partition instance into a TP instance that has a k-bin tree packing with bin capacity M iff there is a partition of the s_is. The TP instance has $M = 2B + 1$ and $k = 3$. The tree T for this instance has three levels. The size of the root is M; the root has n children; the size of the ith child is $2s_i$, $1 \leq i \leq n$; and the root has one grandchild whose size is 1 (the grandchild may be be made a child of any one of the n children of the root).

It is easy to see that T may be packed into 3 capacity M bins iff the given s_is have a subset whose sum is B.

■

All VSTs have nodes whose size is a power of 2 (more precisely, some constant times a power of 2). The TP construction of Theorem 1 results in node sizes that are not necessarily a power of 2. Despite Theorem 1, it is possible that TP restricted to nodes whose size is a power of 2 is polynomially solvable. However, we have been unable to develop a polynomial-time algorithm for this restricted version of TP. Instead, we propose a heuristic, which is motivated by the optimality of the First Fit Decreasing (FFD) algorithm to pack bins when the size of each item is a power of a, where $a \geq 2$ is an integer. In FFD [1], items are packed in decreasing order of size; when an item is considered for packing, it is packed into the first bin into which it fits; if the item fits in no exisiting bin, a new bin is started. Although this packing strategy does not guarantee to minimize the number of bins into which the items are packed when item sizes are arbitrary integers, the strategy works for the restricted case when the size of each item is of the form a^i, where $a \geq 2$ is an integer. Theorem 2 establishes this by considering a related problem–restricted max packing (RMP). Let $a \geq 2$ be an integer. Let s_i, a power of a, be the size of the ith item, $1 \leq i \leq n$. Let c_i be the capacity of the ith bin, $1 \leq i \leq k$. In the *restricted max packing* problem, we are to maximize the sum of the sizes of the items packed into the k bins. We call this version of max packing restricted because the item sizes must be a power of a.

Theorem 2 *FFD solves the RMP problem.*

Proof Let a, n, c_i, $1 \leq i \leq k$ and s_i, $1 \leq i \leq n$ define an instance of RMP. Suppose that in the FFD packing the sum of the sizes of items packed into bin i is b_i. Clearly, $b_i \leq c_i$, $1 \leq i \leq k$. Let S be the subset of items not packed in any bin. If $S = \emptyset$, all items have been packed and the packing is necessarily optimal. So, assume that $S \neq \emptyset$. Let A be the size of smallest item in S. Let x_i and y_i be non-negative integers such that $b_i = x_i A + y_i$ and $0 \leq y_i < A$, $1 \leq i \leq k$. We make the following observations:

(a) $(x_i + 1) * A > c_i$, $1 \leq i \leq k$. This follows from the definition of FFD and the fact that S has an unpacked item whose size is A.

(b) Each item that contributes to a y_i has size less than A. This follows from the fact that all item sizes (and hence A) are a power of a. In particular, note that every item size $\geq A$ is a multiple of A.

(c) Each item that contributes to the x_iA component of a b_i has size $\geq A$. Though at first glance, it may seem that many small items could collectively contribute to this component of b_i, this is not the case when FFD is used on items whose size is a power of a. We prove this by contradiction. Suppose that for some i, $x_iA = B + C$, where $B > 0$ is the contribution of items whose size is less than A and C is the contribution of items whose size is $\geq A$. As noted in (b), every size $\geq A$ is a multiple of A. So, C is a multiple of A. Hence B is a multiple of A formed by items whose size is smaller than A. However S has an item whose size is A. FFD should have packed this item of size A into bin i before attempting to pack the smaller size items that constitute B.

The sum of the sizes of items packed by FFD is

$$FFDSize = \sum_{1 \leq i \leq k} x_iA + \sum_{1 \leq i \leq k} y_i \tag{6}$$

For any other k-bin packing of the items, let $b'_i = x'_iA + y'_i$, where x'_i and y'_i are non-negative integers and $0 \leq y'_i < A$, $1 \leq i \leq k$, be the sum of the sizes of items packed into bin i. For this other packing, we have

$$OtherSize = \sum_{1 \leq i \leq k} x'_iA + \sum_{1 \leq i \leq k} y'_i \tag{7}$$

From observation (a), it follows that $x'_i \leq x_i$, $1 \leq i \leq k$. So, the first sum of Equation 7 is $\leq$ the first sum of Equation 6.

From observations (b) and (c) and the fact that A is the smallest size in S, it follows that every item whose size is less than A is packed into a bin by FFD and contributes to the second sum in Equation 6. Since all item sizes are a power of a, no item whose size is more than A can contribute to a y'_i. Hence, the second sum of Equation 7 is $\leq$ the second sum of Equation 6. So, $OtherSize \leq FFDSize$ and FFD solve the RMP problem. ■

The optimality of FFD for RMP motivates our tree packing heuristic of Figure 3, which attempts to pack a tree into k bins each of size M. It is assumed that the tree height is $\leq k$. The heuristic uses the notions of a ready node and a critical node. A *ready node* is one whose ancestors have been packed into prior bins. Only a ready node may be packed into the current bin. A *critical node* is an, as yet, unpacked node whose height[3] equals the number of bins remaining for packing. Clearly, all critical nodes must be ready nodes and must be packed into the current bin if we are to successfully pack all tree nodes into the given k bins. So, our heuristic ensures that critical nodes are ready nodes. Further, it first packs all critical nodes into the current bin and then packs the remaining ready nodes in decreasing order of node size. We may use the binary search technique to determine the smallest M for which the heuristic is successful in packing the given tree.

[3]The height of a leaf is 1; the height of a non-leaf is 1 more than the maximum of the heights of its children.

Step 1: [Initialize]
$currentBin = 1$; $readyNodes$ = tree root;

Step 2: [Pack into current bin]
Pack all critical ready nodes into $currentBin$;
if bin capacity is exceeded **return failure**;
Pack remaining ready nodes into $currentBin$ in decreasing order of node size;

Step 3 [Update Lists]
if all tree nodes have been packed **return** success;
if $currentBin == k$ **return failure**;
Remove all nodes packed in Step 2 from $readyNodes$;
Add to $readyNodes$ the children of all nodes packed in Step 2;
$currentBin++$;
Go to Step 2;

Figure 3: Tree packing heuristic

5 Experimental Results

Our algorithms for pipelines multibit tries were programmed in C++ and compiled using the GCC 3.3.5 compiler with optimization level `O3`. The compiled codes were run a 2.80 GHz Pentium 4 PC. Our algorithms for pipelined multibit tries were benchmarked against the best multibit trie algorithms of [6].

Basu and Narliker [7] and Kim and Sahni [6] have proposed algorithms for the construction of pipelined multibit tries. Since the algorithms of [6] are superior to those of [7], we focus on the algorithms of [6]. [6] develops an algorithm PFST-2, which is a 2-stage algorithm that results in pipelined FSTs that miminize total memory subject to minimizing the maximum per-stage memory. Sahni and Kim [6] also propose two algorithms PU-2n and PART that are based on FSTs but result in VSTs that are superior for pipeline applications than the optimal FSTs generated by PFST-2. In particular, the PU-2n tries require smaller total memory than do the tries of PFST-2; the maximum per-stage memory no more than that for PFST-2. The PART tries have a smaller maximum per-stage memory requirement than the tries of PFST-2 but require more total memory. Henceforth, we abbreviate PFST-2 to PFST. Let VST denote the algorithm of Sahni and Kim [5], which constructs VSTs with minimum total memory and let PVST be our algorithm of Section 3. So, in all, we have 5 algorithms–PFST, PU-2n, PART, VST and PVST–for the construction of pipelined multibit tries. Only one of these, PFST, results in an FST and the others result in VSTs.

We first determine the effectiveness of our tree packing heuristic of Figure 3 relative to the straightforward mapping (i.e., nodes at level l of the multibit trie are packed into stage $l+1$, $0 \leq l < k$ of the pipeline). For our experiment, we use the 6 data sets–RRC04, RRC03b, RRC01, MW02, PA, and ME02–used in [6]. The number of prefixes in these data sets is 109600, 108267, 103555, 87618, 85987, and 70306, respectively. Table 1 gives the reduction in maximum per-stage memory when we use our tree packing heuristic rather than the straightforward mapping. For example, on our 6 data sets, the tree packing heuristic reduced the maximum per-stage memory

required by the multibit trie generated by PVST by between 0% and 31%; the mean reduction was 11% and the standard deviation was 10%. The reduction obtained by the tree packing heuristic was as high as 44% when applied to the tries constructed by the algorithms of [6].

Algorithm	Min	Max	Mean	Standard Deviation
PFST	0%	41%	18%	15%
PU-2n	0%	41%	17%	15%
PART	0%	44%	15%	12%
VST	0%	20%	7%	7%
PVST	0%	31%	11%	10%

Table 1: Reduction in maximum per-stage memory resulting from tree packing heuristic

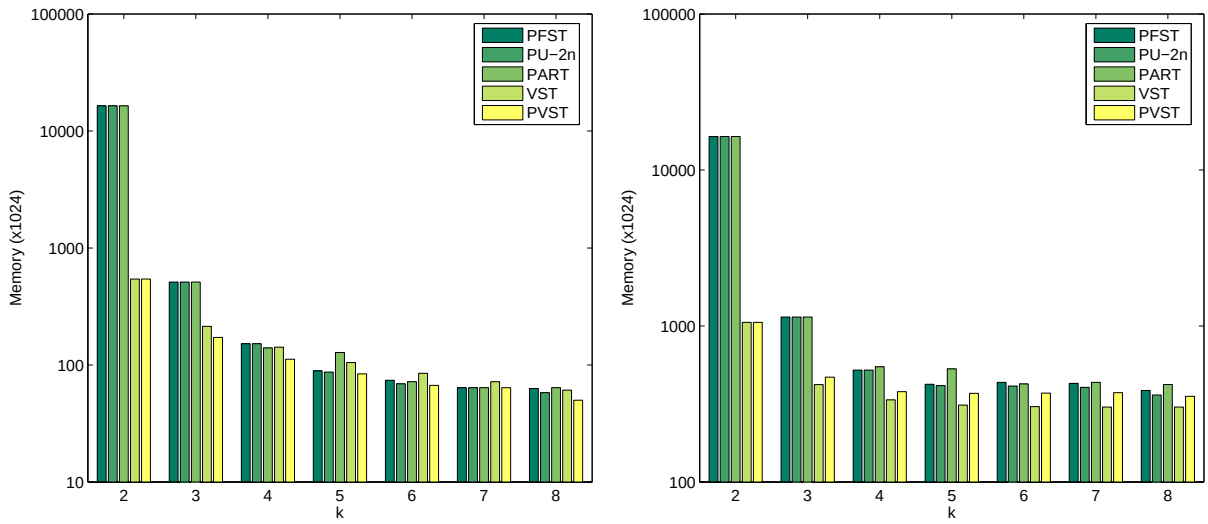
Tables 2 and 3, respectively, give the maximum per-stage memory and total memory requirements for the multibit tries resulting from our 5 algorithms. In each case, the tries were mapped into k, $2 \leq k \leq 8$, pipeline stages using our tree packing heuristic. Figure 4 plots this data for RRC01.

k		2	3	4	5	6	7	8
RRC04	PFST	16384	512	256	146	136	64	64
	PU-2n	16384	512	256	136	136	64	64
	PART	16384	512	198	141	103	73	64
	VST	890	235	147	108	88	72	62
	PVST	890	188	119	89	71	64	53
RRC03b	PFST	16384	512	256	154	151	64	64
	PU-2n	16384	512	256	151	151	64	64
	PART	16384	512	205	128	102	76	64
	VST	927	236	147	109	88	73	62
	PVST	927	189	118	89	70	64	56
RRC01	PFST	16384	512	152	89	74	64	63
	PU-2n	16384	512	152	87	69	64	58
	PART	16384	512	140	128	72	64	64
	VST	543	214	142	105	85	72	61
	PVST	543	172	112	84	67	64	50
MW02	PFST	16384	512	137	128	68	64	61
	PU-2n	16384	512	137	128	64	64	54
	PART	16384	512	128	93	70	64	64
	VST	657	179	125	88	71	**58**	50
	PVST	512	144	94	71	64	64	45
PA	PFST	16384	512	127	75	64	64	52
	PU-2n	16384	512	127	73	64	64	48
	PART	16384	512	128	81	64	64	64
	VST	588	178	125	88	72	**59**	53
	PVST	512	142	93	70	64	64	46
ME02	PFST	15258	512	141	118	64	64	41
	PU-2n	15258	512	141	118	64	64	39
	PART	15258	512	141	91	73	64	43
	VST	610	155	105	74	59	47	42
	PVST	610	128	78	64	49	41	38

Table 2: Maximum per-stage memory (KB)

k		2	3	4	5	6	7	8
RRC04	PFST	16520	1286	799	558	551	449	443
	PU-2n	16520	1286	798	549	542	425	417
	PART	16520	1286	685	631	571	471	442
	VST	1402	461	361	330	321	319	318
	PVST	1402	504	400	387	389	391	371
RRC03b	PFST	16535	1296	811	571	564	446	440
	PU-2n	16535	1296	810	562	555	423	414
	PART	16535	1296	698	640	556	488	447
	VST	1439	463	361	329	320	317	317
	PVST	1439	506	399	386	387	389	357
RRC01	PFST	16399	1142	522	423	435	429	386
	PU-2n	16399	1142	522	415	412	404	361
	PART	16398	1142	548	531	425	435	422
	VST	1055	421	336	311	304	302	302
	PVST	1055	470	379	370	371	374	354
MW02	PFST	16429	1035	477	480	407	400	355
	PU-2n	16429	1035	477	472	388	378	333
	PART	16429	1035	510	413	407	381	367
	VST	913	363	284	261	255	254	253
	PVST	961	415	328	320	321	323	290
PA	PFST	16397	1011	447	365	375	368	323
	PU-2n	16397	1011	447	358	356	347	302
	PART	16395	1224	478	380	370	370	359
	VST	844	360	282	260	255	254	253
	PVST	923	411	327	318	320	322	282
ME02	PFST	23450	1015	480	413	317	327	283
	PU-2n	23450	1015	480	411	315	312	268
	PART	23450	1015	516	417	394	290	335
	VST	1122	324	245	221	215	213	213
	PVST	1122	375	285	277	256	256	238

Table 3: Total memory (KB)



(a) Maximum per-stage memory

(b) Total memory

Figure 4: Maximum per-stage and total memory (KB) for RRC01

In all but two of the 42 tests (MW02 and PA with $k = 7$), PVST results in the least maximum per-stage memory requirement. Tables 4 and 5 give the maximum per-stage and total memory required by the 5 algorithms normalized by the requirements for PVST. The maximum per-stage memory requirement for the algorithms of [6] are up to 32 times that of PVST while the requirement for VST is up to 35% more than that of PVST. On two of our test cases, VST required up to 9% less per-stage memory than did PVST. On average, the total memory required by the multibit tries produced by VST was 13% less than that required by the PVST tries; the tries generated by the algorithms of [6] required, on average, about 3.5 times the total memory required by the PVST tries.

Algorithm	Min	Max	Mean	Standard Deviation
PFST	1.00	32.00	5.16	8.90
PU-2n	1.00	32.00	5.14	8.91
PART	1.00	32.00	5.11	8.92
VST	0.91	1.35	1.18	0.11
PVST	1	1	1	0

Table 4: Maximum per-stage memory normalized by PVST's maximum per-stage memory

Table 6 shows the time taken by the various algorithms to determine the pipelined multibit tries for the case $k = 8$. The shown time includes the time for the tree packing heuristic. Figure 5 plots this data. VST has an execution time comparable to that of the algorithms of [6] but produces significantly superior pipelined tries. Although PVST takes about 3 times as much time as does VST, it usually generates tries that require significantly less maximum per-stage memory.

Algorithm	Min	Max	Mean	Standard Deviation
PFST	1.09	20.90	3.57	5.22
PU-2n	1.02	20.90	3.54	5.23
PART	1.13	20.90	3.60	5.20
VST	0.79	1	0.87	0.06
PVST	1	1	1	0

Table 5: Total memory normalized by PVST's total memory

Data Set	PFST	PU-2n	PART	VST	PVST
RRC04	303	361	296	450	1256
RRC03b	302	355	379	441	1253
RRC01	372	410	311	428	1196
MW02	244	306	174	350	995
PA	274	308	167	334	993
ME02	1373	206	1394	296	838

Table 6: Execution time (msec) for computing 8-level multibit-stride trie

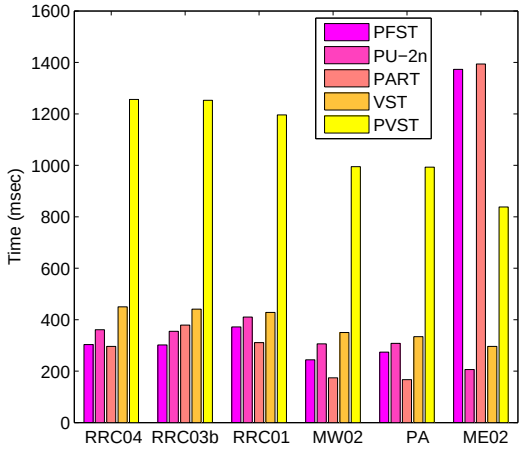


Figure 5: Execution time (msec) for computing 8-level multibit-stride trie

6 Conclusion

We have developed a tree packing heuritic that reduces the maximum per-stage memory required by the multibit tries of [6] by as much as 44%. Our PVST algorithm results in VST multibit tries, which when mapped into a pipelined architecture, have a maximum per-stage memory requirement that is up to 1/32 that required by the tries of [6].

References

[1] E.Horowitz, S.Sahni, and S.Rajasekeran, Computer Algorithms/C++, W. H. Freeman, NY, 1997.

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

[3] 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.

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

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

[6] K. Kim and S. Sahni, Efficient construction of pipelined multibit-trie Router-Tables, *Paper in Review.*

[7] A.Basu and G. Narlikar Fast Incremental Updates for Pipeline Forwarding Engines, *InfoCom*, 2003.