Succinct Representation Of Static Packet Classifiers *

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

November 14, 2006

Abstract

We develop algorithms for the compact representation of the 2-dimensional tries that are used for Internet packet classification. Our compact representations are experimentally compared with competing compact representations for multi-dimensional packet classifiers and found to simultaneously reduce the number of memory accesses required for a lookup as well as the memory required to store the classifier.

1 Introduction

An Internet router classifies incoming packets based on their header fields using a classifier, which is a table of rules. Each classifier 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 classifier, 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.

Data structures for multi-dimensional (i.e., $d > 1$) packet classification are surveyed in [1]-[17]. Our focus in this paper is the succinct representation of the 2- and higher dimensional tries that are commonly used to represent multi-dimensional packet classifiers; the succinct representation is to support high-speed packet classification. For this work, we assume that the classifier is static. That is, the set of rules that comprise the classifier does not change (no inserts/deletes). This assumption is consistent with that made in most of the classifier literature where the objective is to develop a memory-efficient classifier representation that can be searched very fast.

We begin, in Section 2, by reviewing the 1- and 2-dimensional binary trie representation of a classifier together with the research that has been done on the succinct representation of these structures. In Sections 3 through 5, we develop our our algorithms for the succinct representation of 2-dimensional tries. Experimental results are presented in Section 6.

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

2 Background and Related Work

2.1 One-Dimensional Packet Classification

We assume that the filters in a 1-dimensional classifier are prefixes of destination addresses. Many of the data structures developed for the representation of a classifier are based on the *binary trie* structure [5]. A binary trie is a binary tree structure in which each node has a data field and two children fields. Branching is done based on the bits in the search key. A left child branch is followed at a node at level i (the root is at level 0) if the ith bit of the search key (the leftmost bit of the search key is bit 0) is 0; otherwise a right child branch is followed. Level i nodes store prefixes whose length is i in their data fields. 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 binary trie. Let $Q(N)$ be the bit string defined by the path from the root to N. $Q(N)$ is the prefix that corresponds to N. $Q(N)$ is stored in $N.data$ in case $Q(N)$ is one of the prefixes to be stored in the trie.

Several strategies–e.g., tree bitmap [3], shape shifting tries [15]–have been proposed to improve the lookup performance of binary tries. All of these strategies collapse several levels of each subtree of a binary trie into a single node, which we call a *supernode*, that can be searched with a number of memory accesses that is less than the number of levels collapsed into the supernode. For example, we can access the correct child pointer (as well as its associated prefix) in a multibit trie with a single memory access independent of the size of the multibit node. The resulting trie, which is composed of supernodes, is called a *supernode trie*.

The data structure we propose in this paper also is a supernode trie structure. Our structure is most closely related to the shape shifting trie (SST) structure of Song et al. [15], which in turn draws heavily from the tree bitmap (TBM) scheme of Eatherton et al. [3] and the technique developed by Jacobson [6, 11] for the succinct representation of a binary tree. In TBM we start with the binary trie for our classifier and partition this binary trie into subtries that have at most S levels each. Each partition is then represented as a (TBM) supernode. S is the *stride* of a TBM supernode. While $S = 8$ is suggested in [3] for real-world IPv4 classifers, we use $S = 2$ here to illustrate the TBM structure.

Fig. 1 (a) shows a partitoning of a binary trie into 4 subtries W–Z that have 2 levels each. Although a full binary trie with $S = 2$ levels has 3 nodes, X has only 2 nodes and Y and Z have only one node each. Each partition is is represented by a supernode (Fig. 1 (b)) that has the following components:

1. A $(2^S - 1)$-bit bit map IBM (internal bitmap) that indicates whether each of the up to $2^S - 1$ nodes in the partition contains a prefix. The IBM is constructed by superimposing the partition nodes on a full binary trie that has S levels and traversing the nodes of this full binary trie in level order. For node W, the IBM is 110 indicating that the root and its left child have a prefix and the root's right child is either absent or has no prefix. The IBM for X is 010, which indicates that the left child of the root of X has a prefix and that the right child of the root is either absent or has no prefix (note that the root itself is always present and so a 0 in the leading position of an IBM indicates that the root has no prefix). The IBM's for Y and Z are both 100.

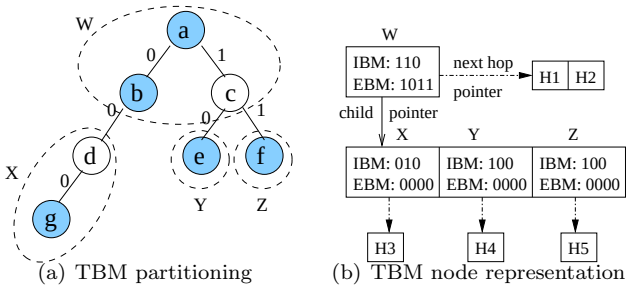


Figure 1: TBM example

2. A 2^S-bit EBM (external bit map) that corresponds to the 2^S child pointers that the leaves of a full S-level binary trie has. As was the case for the IBM, we superimpose the nodes of the partition on a full binary trie that has S levels. Then we see which of the partition nodes has child pointers emanating from the leaves of the full binary trie. The EBM for W is 1011, which indicates that only the right child of the leftmost leaf of the full binary trie is null. The EBMs for X, Y and Z are 0000 indicating that the nodes of X, Y and Z have no children that are not included in X, Y, and Z, respectively. Each child pointer from a node in one partition to a node in another partition becomes a pointer from a supernode to another aupercode. To reduce the space required for these inter-supernode pointers, the children supernodes of a supernode are stored sequentially from left to right so that using the location of the first child and the size of a supernode, we can compute the location of any child supernode.

3. A child pointer that points to the location where the first child supernode is stored.

4. A pointer to a list NH of next-hop data for the prefixes in the partition. NH may have up to $2^S - 1$ entries. This list is created by traversing the partition nodes in level order. The NH list for W is $nh(P1)$ and $nh(P2)$, where $nh(P1)$ is the next hop for prefix $P1$. The NH list for X is $nh(P3)$. While the NH pointer is part of the supernode, the NH list is not. The NH list is conveniently represented as an array.

The NH list (array) of a supernode is stored separate from the supernode itself and is accessed only when the longest matching prefix has been determined and we now wish to determine the next hop associated with this prefix. If we need b bits for a pointer, then a total of $2^{S+1} + 2b - 1$ bits (plus space for an NH list) are needed for each TBM supernode. Using the IBM, we can determine the longest matching prefix in a supernode; the EBM is used to determine whether we should move next to the first, second, etc. child of the current supernode. If a single memory access is sufficient to retrieve an entire supernode, we can move from one supernode to its child with a single access. The total number of memory accesses to search a supernode trie becomes the number of levels in the supernode trie plus 1 (to access the next hop for the longest matching prefix).

The SST supernode structure proposed by Song et al. [15] is obtained by partitioning a binary trie into subtries that have at most K nodes each. K is the *stride* of an SST supernode. To correctly search an SST, each SST

supernode requires a shape bit map (SBM) in addition to an IBM and EBM. The SBM used by Song et al. [15] is the succinct representation of a binary tree developed by Jacobson [6]. Jacobson's SBM is obtained by replacing every null link in the binary tree being coded by the SBM with an external node. Next, place a 0 in every external node and a 1 in every other node. Finally, traverse this extended binary tree in level order, listing the bits in the nodes as they are visited by the traversal.

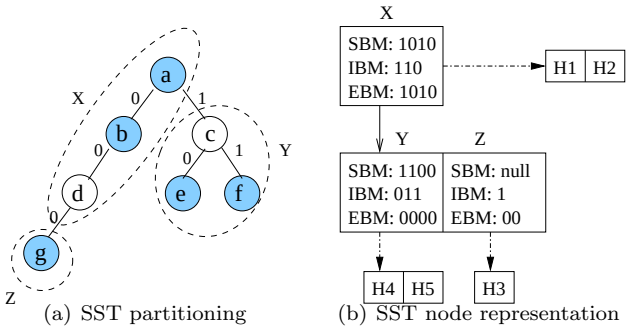


Figure 2: SST for binary trie of Fig. 1 (a)

Suppose we partition our example binary trie of Fig. 1 (a) into binary tries that have at most $K = 3$ nodes each. Fig. 2 (a) shows a possible partitioning into the 3 partitions X-Z. X includes nodes a, b and d of Fig. 1 (a); Y includes nodes c, e and f; and Z includes node g. The SST representation has 3 (SST) supernodes. The SBMs for the supernodes for X-Z, respectively, are 1101000, 1110000, and 100. Note that a binary tree with K internal nodes has exactly $K + 1$ external nodes. So, when we partition into binary tries that have at most K internal nodes, the SBM is at most $2K + 1$ bits long. Since the first bit in an SBM is 1 and the last 2 bits are 0, we don't need to store these bits explicitly. Hence, an SBM requires only $2K - 2$ bits of storage. Fig. 2 (b) shows the node representation for each partition of Fig. 2 (a). The shown SBMs exclude the first and last two bits.

The IBM of an SST supernode is obtained by traversing the partition in level order; when a node is visited, we ouput a 1 to the IBM if the node has a prefix and a 0 otherwise. The IBMs for nodes X-Z are, respectively, 110, 011, and 1. Note than the IBM of an SST supernode is at most K bits in length.

To obtain the EBM of a supernode, we start with the extended binary tree for the partition and place a 1 in each external node that corresponds to a node in the original binary trie and a 0 in every other external node. Next, we visit the external nodes in level order and output their bit to the EBM. The EBMs for our 3 supernodes are, respectively, 1010, 0000, and 00. Since the number of external nodes for each partition is at most $K + 1$, the size of an EBM is at most $K + 1$ bits.

As in the case of the TBM structure, child supernodes of an SST supernode are stored sequentially and a pointer to the first child supernode maintained. The NH list for the supernode is stored in separate memory and a pointer to this list maintained within the supernode. Although the size of an SBM, IBM and EBM varies with the partition size, an SST supernode is of a fixed size and allocates $2K$ bits to the SBM, K bits to the IBM and $K + 1$ bits to the EBM. Unused bits are filled with 0s. Hence, the size of an SST supernode is $4K + 2b - 1$ bits.

Song et al. [15] develop an $O(m)$ time algorithm, called *post-order pruning*, to construct a minimum-node SST, for any given K, from an m-node binary trie. They develop also a *breadth-first pruning* algorithm to construct, for any given K, a minimum height SST. The complexity of this algorithm is $O(m^2)$. Lu and Sahni [10] developed an improved algorithm, which reduces the complexity of minimum height SST construction to $O(m)$.

For dense binary tries, TBMs are more space efficient than SSTs. However, for sparse binary tries, SSTs are more space efficient. Song et al. [15] propose a hybrid SST (HSST) in which dense subtries of the overall binary trie are partitioned into TBM supernodes and sparse subtries into SST supernodes. Fig. 3 shows an HSST for the binary trie of Fig. 1 (b). For this HSST, $K = S = 2$. The HSST has two SST nodes X and Z, and one TBM node Y.

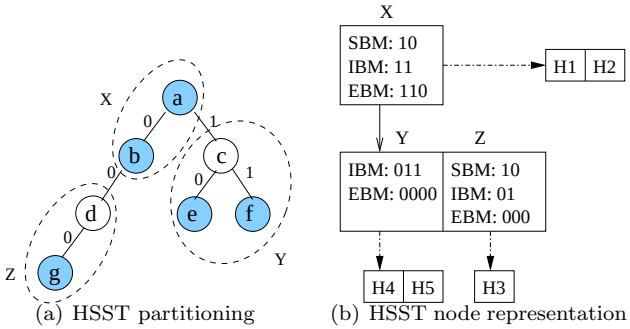


Figure 3: HSST for binary trie of Fig. 1(a)

Although Song et al. [15] do not develop an algorithm to construct a space-optimal HSST, they propose a heuristic that is a modification of their breadth-first pruning algorithm for SSTs. This heuristic guarantees that the height of the constructed HSST is no more than that of the height-optimal SST. Lu and Sahni [10] developed dynamic programming formulations for the construction of space-optimal HSSTs.

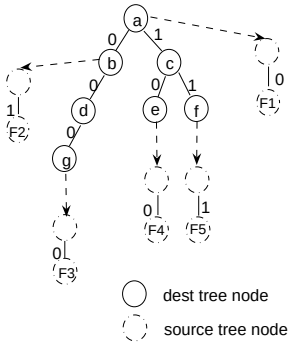
2.2 Two-Dimensional Packet Classification

We assume that the filters are of the form (D, E), where D is a destination address prefix and E is a source address prefix. A 2-dimensional classifier may be represented as a 2-dimensional binary trie (2DBT), which is a one-dimensional binary trie (called the top-level trie) in which the data field of each node is a pointer to a (possibly empty) binary trie (called the lower-level trie). So, a 2DBT has 1 top-level trie and potentially many lower-level tries.

Consider the 5-rule two-dimensional classifier of Figure 4(a). For each rule, the filter is defined by the Dest (destination) and Source prefixes. So, for example, $F2 = (0*, 1*)$ matches all packets whose destination address begins with 0 and whose source address begins with 1. When a packet is matched by two or more filters, the matching rule with least cost is used. The classifier of Figure 4(a) may be represented as a 2DBT in which the top-level trie is constructed using the destination prefixes. In the context of our destination-source filters, this top-level trie is called the *destination trie* (or simply, dest trie). Let N be a node in the destination trie. If no

Filter	Dest Prefix	Source Prefix	Cost
F1	*	0*	1
F2	0*	1*	2
F3	000*	0*	3
F4	10*	0*	4
F5	11*	1*	5

(a) 5 prefixes



(b) Corresponding two-dimensional binary trie

Figure 4: Prefixes and corresponding two-dimensional binary trie

dest prefix equals $Q(N)$, then $N.data$ points to an empty lower-level trie. If there is a dest prefix D that equals $Q(N)$, then $N.data$ points to a binary trie for all source prefixes E such that (D, E) is a filter. In the context of destination-source filters, the lower-level tries are called *source trees*.

Every node N of the dest trie of a 2DBT has a (possibly empty) source trie *hanging* from it. Let a and b be two nodes in the dest trie. Let b be an ancestor of a. We say that the source trie that hangs from b is an *ancestor trie* of the one that hangs from a. Figure 4(b) gives the 2DBT for the filters of Figure 4(a).

Srinivasan and Varghese [13] proposed using two-dimensional one-bit tries, a close relative of 2DBTs, for destination-source prefix filters. The proposed two-dimensional trie structure takes $O(nW)$ memory, where n is the number of filters in the classifier and W is the length of the longest prefix. Using this structure, a packet may be classified with $O(W^2)$ memory accesses. The basic two-dimensional one-bit trie may be improved upon by using pre-computation and switch pointers [13]. The improved version classifies a packet making only $O(W)$ memory accesses. Srinivasan and Varghese [13] also propose extensions to higher-dimensional one-bit tries that may be used with d-dimensional, $d > 2$, filters. Baboescu et al. [2] suggest the use of two-dimensional one-bit tries with buckets for d-dimensional, $d > 2$, classifiers. Basically, the destination and source fields of the filters are used to construct a two-dimensional one-bit trie. Filters that have the same destination and source fields are considered to be equivalent. Equivalent filters are stored in a bucket that may be searched serially. Baboescu et al. [2] report that this scheme is expected to work well in practice because the bucket size tends to be small. They note also that switch pointers may not be used in conjunction with the bucketing scheme.

Lu and Sahni [8], develop fast polynomial-time algorithms to construct space-optimal constrained 2DMTs (two-dimensional multibit tries). The constructed 2DMTs may be searched with at most k memory accesses, where k is a design parameter. The space-optimal constrained 2DMTs of Lu and Sahni [8] may be used for d-dimensional filters, $d > 2$, using the bucketing strategy proposed in [2]. For the case $d = 2$, switch pointers may be employed

to get multibit tries that require less memory than required by space-optimal constrained 2DMTs and that permit packet classification with at most k memory accesses. Lu and Sahni [8] develop also a fast heuristic to construct good multibit tries with switch pointers. Experiments reported in [8] indicate that, given the same memory budget, space-optimal constrained 2DMT structures perform packet classification using 1/4 to 1/3 as many memory accesses as required by the two-dimensional one-bit tries of [13, 2].

3 Space-Optimal 2DHSSTs

Let T be a 2DBT. We assume that the source tries of T have been modified so that the last prefix encountered on each search path is the least-cost prefix for that search path. This modification is accomplished by examining each source-trie node N that contains a prefix and replacing the contained prefix with the least-cost prefix on the path from the root to N. A 2DHSST may be constructed from T by partitioning the top-level binary trie (i.e., the dest trie) of T and each lower-lever binary trie into a mix of TBM and SST supernodes. Supernodes that cover the top-level binary trie use their NH (next hop) lists to store the root supernodes for the lower-level HSSTs that represent lower-level tries of T.

Figure 5 shows a possible 2DHSST for the 2DBT of Figure 4(b). The supernode strides used are $K = S = 2$. A 2DHSST may be searched for the least-cost filter that matches any given pair of destination and source addresses (da, sa) by following the search path for da in the destination HSST of the 2DHSST. All source tries encountered on this path are searched for sa. The least-cost filter on these source-trie search paths that matches sa is returned. Suppose we are to find the least-cost filter that matches (000,111). The search path for 000 takes us first to the root (ab) of the 2DHSST of Figure 5 and then to the left child (dg). In the 2DHSST root, we go through nodes a and b of the dest binary trie and in the supernode dg, we go through nodes d and g of T. Three of the encountered nodes (a, b, and g) have a hanging source trie. The corresponding source HSSTs are searched for 111 and F2 is returned as the least-cost matching filter.

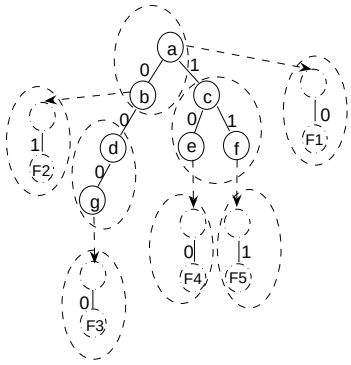


Figure 5: Two-dimensional supernode trie for Figure 4(a)

To determine the number of memory accesses required by a search of a 2DHSST, we assume sufficient memory bandwidth that an entire supernode (this includes the IBM, EBM, child and NH pointers) may be accessed with a single memory reference. To access a component of the NH array, an additional memory access is required. For each supernode on the search path for da, we make one memory access to get the supernode's fields (e.g., IBM, EBM, child and NH pointers). In addition, for each supernode on this path, we need to examine some number of hanging source HSSTs. For each source HSST examined, we first access a component of the dest-trie supernode's NH array to get the root of the hanging source HSST. Then we search this hanging source HSST by accessing the remaining nodes on the search path (as determined by the source address) for this HSST. Finally, the NH component corresponding to the last node on this search path is accessed. So, in the case of our above example, we make 2 memory accesses to fetch the 2 supernodes on the dest HSST path. In addition, 3 source HSSTs are searched. Each requires us to access its root supernode plus an NH component. in each source HSST. The total number of memory accesses is $2 + 2 * 3 = 8$.

Let $MNMA(X)$ be the *maximum number of memory accesses* (MNMA) required to search a source HSST X. For a source HSST, the MNMA includes the access to NH component of the last node on the search path. So, $MNMA(X)$ is one more than the number of levels in X. Let U be a 2DHSST for T with strides S and K. Let P be any root to leaf path in the top level HSST of U. Let the sum of the MNMAs for the lower-level HSSTs on the path P be $H(P)$. Let $nodes(P)$ be the number of supernodes on the path P. Define $2DHSST(h)$ to be the subset of the possible 2DHSSTs for T for which

$$\max_P\{H(P) + nodes(P)\} \leq h \tag{1}$$

Note that every $U, U \in 2DHSST(h)$, can be searched with at most h memory accesses per lookup. Note also that some 2DHSSTs that have a path P for which $H(P) + nodes(P) = h$ can be searched with fewer memory accesses than h as there may be no (da, sa) that causes a search to take the longest path through every source HSST on paths P for which $H(P) + nodes(P) = h$.

We consider the construction of a space-optimal 2DHSST V such that $V \in 2DHSST(H)$. We refer to such a V as a space-optimal $2DHSST(h)$. Let N be a node in T's top-level trie, and let $2DBT(N)$ be the 2-dimensional binary trie rooted at N. Let $opt1(N, h)$ be the size (i.e., number of supernodes) of the space-optimal $2DHSST(h)$ for $2DBT(N)$. $opt1(root(T), H)$ gives the size of a space-optimal $2DHSST(H)$ for T. Let $g(N, q, h)$ be the size (excluding the root) of a space-optimal $2DHSST(h)$ for $2DBT(N)$ under the constraint that the root of the $2DHSST$ is a TBM supernode whose stride is q. So, $g(N, S, h) + 1$ gives the size of a space-optimal $2DHSST(h)$ for $2DBT(N)$ under the constraint that the root of the $2DHSST$ is a TBM supernode whose stride is S. We see that, for $q > 0$,

$$g(N, q, h) = \min_{m(N) \leq i \leq h}\{g(LC(N), q-1, h-i) + g(RC(N), q-1, h-i) + s(N, i)\} \tag{2}$$

where $m(N)$ is the minimum possible value of MNMA for the source trie (if any) that hangs from the node N (in case there is no source trie hanging from N, $m(N) = 0$), $g(N, 0, h) = opt1(N, h-1)$, $g(null, t, h) = 0$, and $LC(N)$

and $RC(N)$ respectively, are the left and right children (in T) of N. $s(N,i)$ is the size of the space-optimal HSST for the source trie that hangs off from N under the constraint that the HSST has an MNMA of at most i. $s(N,i)$ is 0 if N has no hanging source trie.

Let $opt1(N,h,k)$ be the size of a space-optimal $2DHSST(h)$ for $2DBT(N)$ under the constraint that the root of the $2DHSST$ is an SST supernode whose utilization is k. It is easy to see

$$opt1(N,h) = \min\{g(N,S,h)+1, \min_{0<k\leq K}\{opt1(N,h,k)\}\} \tag{3}$$

Suppose that $k>0$ and $h>0$. If N has no child,

$$opt1(N,h,k) = 1 + s(N,h-1) \tag{4}$$

When N has only one child a,

$$opt1(N,h,k) = \min_{m(N)\leq i<h}\{f(a,h-i,k-1)+s(N,i)\} \tag{5}$$

where $f(N,h,k)$ is the size of a space-optimal $2DHSST(h)$ for $2DBT(N)$ plus the parent (in T) of N (but excluding the lower-level source trie (if any) that hangs from N) under the constraint that the root of the 2DHSST is an SST supernode whose utilization is $k+1$. For example, when $k=0$, the root of the constrained 2DHSST has a utilization 1 and contains only the parent of N; the remaining supernodes of the 2DHSST represent $2DBT(N)$. Thus $f(N,h,k) = opt1(N,h,k)$ when $k>0$ and $1+opt1(N,h-1,0)$ when $k=0$.

When N has two children a and b,

$$opt1(N,h,k) = \min_{m(N)\leq i<h}\{\min_{0\leq j<k}\{f(a,h-i,j)+f(b,h-i,k-j-1)-1\}+s(N,i)\} \tag{6}$$

For $h\leq 0$

$$opt1(N,h,*) = \infty \tag{7}$$

When we have n filters and the length of the longest prefix is W, the number of nodes in the dest trie of T is $O(nW)$ and the number of source tries in T is $O(n)$. The time to compute all $s(N,h)$ values using the algorithm described in [10] to compute space-optimal HSSTs is $O(n^2WHK^2)$ time. Using Equation 2 and previously computed g values, $O(H)$ time is needed to compute each $g(*,*,*)$ value. Using Equation 3, each $opt1(*,*)$ value may be computed in $O(K)$ time. Using Equations 4-7, we can compute each $opt1(*,*,*)$ value in $O(KH)$ time. Since there are $O(nWH)$ $opt1(*,*)$, $O(nWHK)$ $opt1(*,*,*)$, and $O(nWSH)$ $g(*,*,*)$ values to compute, the time to determine $opt1(root(T),H)$ is $O(n^2WHK^2+nWHK+nWH^2K^2+nWSH^2) = O(n^2WHK^2)$ (as, in typical applications, $n>H$).

4 2DHSSTs With Prefix Inheritance (2DHSSTP)

Let T be the 2DBT of Figure 4. Consider the dest-trie supernode ab of Figure 5. This supernode represents the subtrie of T that is comprised of the binary nodes a and b. A search in this subtrie has three exit points–left child

of b, right child of b and right child of a. For the first two exit points, the source tries that hang off of a and b are searched whereas for the third exit point, only the source trie that hangs off of a is searched. We say that the first two exit points *use* the source tries that hang off of a and b while the third exit point uses only the source trie that hangs off of a. If the source trie that hangs off of b is augmented with the prefixes in the source trie that hangs off of a, then when the first two exit points are used, only the augmented source trie that hangs off of b need be searched.

In *prefix inheritance*, each non-empty source trie in a partition is augmented with the prefixes in all source tries that hang off of ancestors in the partition. When this augmentation results in duplicate prefixes, the least-cost prefix in each set of duplicates is retained. The resulting augmented source tries are called *exit tries*. In a 2DHSST with prefix inheritance (2DHSSTP), prefix inheritance is done in each supernode. Figure 6 gives the 2DHSSTP for the 2DHSST of Figure 5.

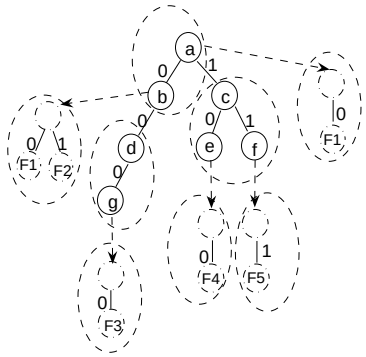


Figure 6: 2DHSSTP for Figure 5

Notice that to search a 2DHSSTP we need search at most one exit trie for each dest-trie supernode encountered– the last exit trie encountered in the search of the partition represented by that dest-trie supernode. So, when searching for $(da, sa) = (000, 111)$, we search the exit tries that hang off of b and g for 111. The number of memory accesses is 2 (for the two supernodes ab and dg) + 2 (1 to access the supernode in each of the two source tries searched) + 2 (to access the NH arrays for the source trie supernodes) = 6. The same search using the 2DHSST of Figure 5 will search three source tries (those hanging off of a, b, and g) for a total cost of 8 memory accesses.

A node N in a dest-trie partition is a *dominating node* iff there is an exit trie on every path from N to an exit point of the partition. Notice that if N has two children, both of which are dominating nodes, then the exit trie (if any) in N is never searched. Hence, there is no need to store this exit trie.

We are interested in constructing a space-optimal 2DHSSTP that can be searched with at most H memory accesses. Although we have been unable to develop a good algorithm for this, we are able to develop a good algorithm to construct a space-optimal constrained 2DHSSTP for any 2DBT T. Note that the 2DHSSTP for T is comprised of supernodes for the dest-trie of T plus supernodes for the exit tries.

Let $2DHSSTPC(h)$ be a 2DHSSTP that is constrained so that (a) it can be searched with at most h memory accesses and (b) the HSST for each exit trie is a minimum height HSST for that exit trie. Our experimental studies suggest that the space required by an HSST is somewhat insensitive to the height constraint placed on the HSST. So, we expect that the space required by a space-optimal $2DHSSTPC(h)$ is close to that required by a space-optimal $2DHSSTP(h)$.

Let N be a node in the dest-trie of the 2DBT T and let $opt2(N, h)$ be the size of a space-optimal $2DHSSTPC(h)$ for the subtree, $ST(N)$, of T rooted at N. The supernode strides are K and S. Notice that $opt2(root(T), H)$ gives the size of a space-optimal $2DHSSTPC(H)$ for T. The development of a dynamic programming recurrence for $opt2$ follows the pattern used for our earlier dynamic programming recurrences. Let $2DHSSTPC(N, h, k, p)$ be a $2DHSSTPC(h)$ for $ST(N)$ under the constraints (a) the root of the 2DHSSTPC is an SST node whose utilization is k and (b) for the root, prefix inheritance is not limited to the partition of T represented by the root of the 2DHSSTPC; rather prefix inheritance extends up to the p nearest ancestors of N in T and let $opt2(N, h, k, p)$ be the size of a space-optimal $2DHSSTPC(N, h, k, p)$. The recurrence for $opt2(N, h)$ is (a detailed development appears in [10]):

$$opt2(N, h) = \min\{\min_{0<k\leq K}\{opt2(N, h, k, 0)\}, 1 + ss(N) + \sum_{Q\in D_S(N)} opt2(Q, h - 1 - h(Q))\} \quad (8)$$

When we have n filters and the length of the longest prefix is W, $opt2(root(T), H)$ may be computed in $O(n^2W^2HK^2 + nWK + nWHK + nW^2HK^2) = O(n^2W^2HK^2)$ time [10].

5 Implementation Considerations

We use the enhanced base implementation [10] of an HSST for both the dest and source tries of a 2DHSST and a 2DHSSTPC. End node optimization [10] is done on each source trie of a 2DHSST and a 2DHSSTPC. For the dest trie, however, we do the following:

1. Cut off the leaves of the dest binary trie prior to applying the equations of Sections 3 and 4 to construct space-optimal 2DHSSTs and 2DHSSTPCs. Following the construction, identify the parent dest-trie supernode for each leaf that was cut off.

2. In the case of 2DHSSTPCs, each source trie that hangs off of a leaf of the dest binary trie, inherits the prefixes stored along the path, in the parent dest-trie supernode, to this leaf.

3. Each cut-off leaf is replaced by the HSST for its source trie (this source trie includes the inherited prefixes of (2) in case of a 2DHSSTPC). The root of this HSST is placed as the appropriate child of the parent dest-trie supernode. (This requires us to use an additional bit to distinguish between dest-trie supernodes and source HSST roots.)

By handling the leaves of the binary dest-trie as above, we eliminate the need to search the source tries that are on the path, in the dest-trie parent, to a leaf child.

6 Experimental Results

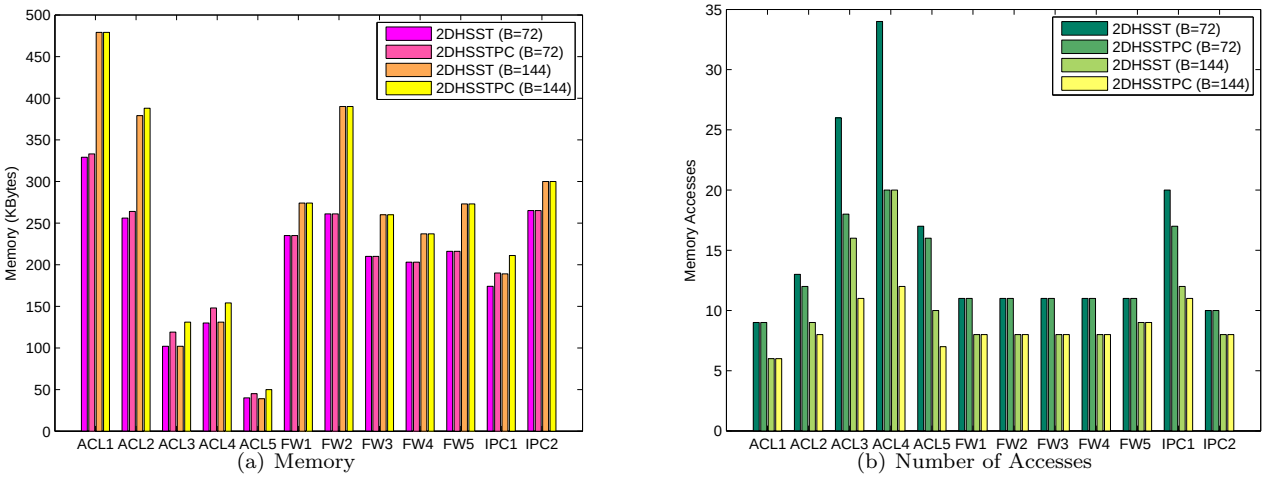
C++ codes for our algorithms for space-optimal 2-dimensional supernode tries were compiled using the GCC 3.3.5 compiler with optimization level `O3` and run on a 2.80 GHz Pentium 4 PC. Our algorithms were benchmarked against recently published algorithms to construct space-efficient data structures for multi-dimensional packet classification [8, 12]. The benchmarked algorithms seek to construct lookup structures that (a) minimize the worst-case number of memory accesses needed for a lookup and (b) minimize the total memory needed to store the constructed data structure. As a result, our experiments measured only these two quantities. Further, all test algorithms were run so as to generate a lookup structure that minimizes the worst-case number of memory accesses needed for a lookup; the size (i.e., memory required) of the constructed lookup structure was minimized subject to this former constraint. Let B be the number of bits that may be retrieved with a single memory access. For benchmarking purposes we assumed that the classifier data structure will reside on a QDRII SRAM, which supports both B=72 bits (dual burst) and B=144 bits (quad burst). For our experiments, we used 22 bits for a pointer (whether a child pointer or a pointer to a next-hop array) and 18 bits for the priority and action associated with a filter.

We evaluated the performance of our proposed data structures using both 2-dimensional and 5-dimensional data sets. We used twelve 5-dimensional data sets that were created by the filter generator of [16]. Each of these data sets actually has 10 different databases of rules. So, in all, we have 120 databases of 5-dimensional rules. The data sets, which are named ACL1 through ACL5 (Access Control List), FW1 through FW5 (Firewall), IPC1 and IPC2 (IP Chain) have, respectively, 20K, 19K, 19K, 19K, 12K, 19K, 19K, 18K, 17K, 17K, 19K, and 20K rules, on average, in each database. Our 2-dimensional data sets, which were derived from these 5-dimensional data sets, have, respectively, 20K, 19K, 10K, 13K, 5K, 19K, 19K, 18K, 17K, 17K, 16K and 20K rules on average in each database. The 2-dimensional rules were obtained from our 5-dimensional rules by stripping off the source and destination port fields as well as the protocol field; the dest and source prefix field were retained. Following this stripping process, duplicates were deleted (i.e., two rules are considered duplicate if they have the same dest prefix and the same source prefix).

First, we compare the space-optimal minimum-access 2DHSST and 2DHSSTPC structures. Figure 7 show the results from our experiment. For 5 of our 12 data sets–ACL2-5, and IPC1–2DHSSTPCs reduce the number of accesses at the expense of increased memory requirement. For the remaining data sets, 2DHSSTPCs and 2DHSSTs require almost the same number of accesses and the same amount of memory.

Across all our data sets, 2DHSSTPCs required between 0% and 29% more memory than required by 2DHSSTs (the mean increase in memory required was 6% and the standard deviation was 9%). As noted earlier, although 2DHSSTPCs required more memory, they required a smaller number of memory accesses for a lookup. The reduction in number of memory accesses afforded by 2DHSSTPCs was between 0% and 41% (the mean reduction was 11% and the standard deviation was 13%).

When B is increased from 72 to 144, for both 2DHSSTs and 2DHSSTPCs, the number of memory accesses



(a) Memory (b) Number of Accesses

Figure 7: Total memory and number of memory accesses required by 2DHSSTs and 2DHSSTPCs

required is reduced, but the total memory required is generally increased. For 2DHSSTs, the total memory required when $B = 144$ normalized by that required when $B = 72$ is between 0.98 and 1.50 (the mean and the standard deviation are 1.21 and 0.19); the number of memory accesses reduces by between 28% and 41% (the mean reduction is 30% and the standard deviation is 9%). For 2DHSSTPCs, the normalized memory requirement is between 1.04 and 1.49 (the mean and standard deviation are 1.23 and 0.16); the reduction in number of memory accesses ranges from 18% to 56% (the mean reduction and the standard deviation are 31% and 11%).

Since our primary objective is to reduce the number of memory accesses, we use 2DHSSTPCs with $B = 144$ for further benchmarking with 2DMTSas and 2DMTds [8]. The 2DMTSas and 2DMTds employed by us used the compression techniques *packed array* [14] and *butler node* [7]. These two techniques are very similar; both attempt to replace a subtrie with a small amount of actual data (prefixes and pointers) by a single node that contains these data. We note that 2DMTds and 2DMTSas are the best of the structures developed in [8], and using these two compression techniques, [9] has established the superiority of 2DMTds and 2DMTSas over other competing packet classification structures such as Grid-of-Tries [13], EGT-PCs [2], and HyperCuts [12]. For this further benchmarking, we constructed space-optimal 2DHSSTPCs with the minimum possible number, H, of memory accesses for a worst-case search. This minimum H was provided as input to the 2DMTSa (2DMTd) algorithm to construct a 2DMTSa (2DMTd) that could be searched with H memory accesses in the worst case. Because of this strategy, the worst-case number of memory accesses for 2DHSSTPCs and 2DMTSas (2DMTd) is the same.

Figure 8 plots the memory memory required by 2DHSSTPCs, 2DMTds, and 2DMTSas. We see that, on the memory criterion, 2DHSSTPCs outperform 2DMTSas by an order of magnitude, and outperform 2DMTSas by an order of magnitude on 4 of our 12 data sets. The memory required by 2DMTds normalized by that required by 2DHSSTPCs is between 1.14 and 624, the mean and standard deviation being 56 and 179. The normalized numbers for 2DMTSas were 9, 49, 17, 11. We also observed that when 2DMTds are given up to 60% more memory

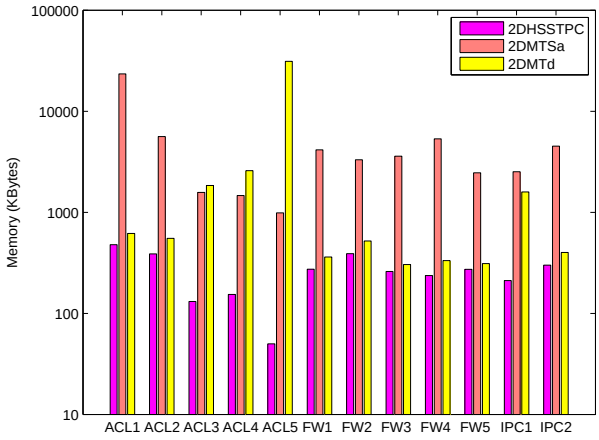


Figure 8: Total memory required by 2DHSSTPCs, 2DMTSas, and 2DMTds

than required by space-optimal DHSSTPCs with the minimum possible H, we can construct 2DMTds that can be searched with 1 or 2 fewer accesses for our data sets FW1-5 and IPC2.

For 5-dimensional tables, we extend 2DHSSTPCs using the bucket scheme proposed in [2]. In [10], we report the observed improvement of extended 2DHSSTPCs over HyperCuts [12], which is one of the previously best known algorithmic schemes for multidimensional packet classification.

7 Conclusion

We have developed succinct representations, 2DHSSTs and 2DHSSTPCs, for two-dimensional IPv4 data sets. 2DHSSTPCs result in fewer memory accesses but more memory requirement than do 2DHSSTs. Since, memory accesses per lookup is the primary optimization criterion, we recommend 2DHSSTPCs over 2DHSSTs. Given the same budget for the number of memory accesses, 2DHSSTPCs require between 0.2% and 88% of the memory required by 2DMTds of [9], and between 2% and 11% of the memory required by 2DMTSas of [9]. For 5-dimensional classifiers, extended 2DHSSTPCs require significantly less memory and significantly less memory accesses, both on average and in the worst-case, than required by HyperCuts [12].

References

[1] F.Baboescu and G.Varghese, Scalable packet classification, *ACM SIGCOMM*, 2001.

[2] F.Baboescu, S.Singh and G.Varghese, Packet Classification for Core Routers: Is there an alternative to CAMs? *INFOCOM*, 2003.

[3] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, *Computer Communication Review*, 34(2): 97-122, 2004.

[4] A.Hari, S.Suri, G.Parulkar, Detecting and resolving packet filter conflicts, *INFOCOM*, 2000.

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

[6] G. Jacobson, Succinct Static Data Structure, *Carnegie Mellon University Ph.D Thesis*, 1998.

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

[8] W. Lu and S. Sahni, Packet Classification Using Two-Dimensional Multibit Tries, *IEEE Symposium on Computers and Communications*, 2005.

[9] W. Lu and S. Sahni, Packet Classification Using Two-Dimensional Multibit Tries, University of Florida, 2006, *http://www.cise.ufl.edu/∼wlu/papers/2dtries.pdf*.

[10] W. Lu and S. Sahni, Succinct Representation of Packet Classifiers, University of Florida, 2006, *http://www.cise.ufl.edu/∼wlu/papers/hsst.pdf*.

[11] J. Munro and S. Rao, Succinct representation of data structures, in *Handbook of Data Structures and Applications*, D. Mehta and S. Sahni editors, Chapman & Hall/CRC, 2005.

[12] S. Singh and F. Baboescu and G. Varghese and J. Wang, Packet Classification Using Multidimensional Cutting, *Proceedings of ACM Sigcomm*, 2003.

[13] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, Scalable Algorithms for Layer four Switching, *Proceedings of ACM Sigcomm*, 8, 1998.

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

[15] H. Song, J. Turner, and J. Lockwood, Shape Shifting Tries for Faster IP Route Lookup, *Proceedings of 13th IEEE International Conference on Network Protocols*, 2005.

[16] D. Taylor and J. Turner, ClassBench: A Packet Classification Benchmark, *INFOCOM*, 2005.

[17] D. Taylor and J. Turner, Scalable Packet Classification using Distributed Crossproducting of Field Labels, *INFOCOM*, 2005.