$O(\log W)$ Multidimensional Packet Classification [*]

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

December 5, 2006

Abstract

We use a collection of hash tables to represent a multidimensional packet classification table. These hash tables are derived from a trie-representation of the multidimensional classifier. The height of this trie is $O(W)$, where W is the sum of the maximum possible length, in bits, of each of the fields of a filter. The leaves at level i of the trie together with markers for some of the leaves at levels j such that $j > i$ are stored in a hash table H_i. The placement of markers is such that a binary search of the H_is successfully locates the highest-priority filter that matches any given packet. The number of hash tables equals the trie height, $O(W)$. Hence a packet may be classified by performing $O(\log W)$ hash-table lookups. So the expected lookup-complexity of our data structure for multidimensional packet classification is $O(\log W)$. Our proposed scheme affords a memory advantage over the $O(\log W)$ one-dimensional scheme of Waldvogel et al. [30]. For multidimensional packet classification, our proposed scheme provides both a time and memory advantage over the extended grid-of-tries scheme of Baboescu et al. [3].

Keywords: multidimensional packet classification, binary search on levels, expected complexity

1 Introduction

An Internet router classifies incoming packets into flows[1] utilizing information contained in packet headers and a table of (classification) rules. This table is called the **router table** (equivalently, **rule table**). Each router table rule is a pair of the form (f, a), where f is a filter and a is an action.

[*]©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

[1]A **flow** is a set of packets that are to be treated similarly for routing purposes.

The action component of a rule specifies what is to be done when a packet that satisfies the rule filter is received. Sample actions are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. The filter component of a rule is a k-tuple the fields of which may represent, for example, the source address of the packet, the destination address, protocol, and port number. Each field of a k-tuple may be specified as a single value, a range or a prefix. A destination address field that is specified as a range $[u, v]$ **matches** the destination address d iff $u \leq d \leq v$, while a destination address field specified by the prefix r matches all destination addresses that begin with r[2]. A filter f matches a packet p iff every field of f matches the corresponding value of p (i.e., the destination field (if any) of f matches the destination address of p, the source address field (if any) of f matches the source address of p, the port number field (if any) of f matches the port number of p, etc.). We may assume that no two rules of the router table have the same filter.

Since an Internet router table may contain several rules that match a given packet p, a tie breaker is used to select a rule from the set of rules that match p. Some commonly used tie breakers are (a) select the first rule in the table that matches p, (b) select the highest-priority rule that matches p, and (c) select the most-specific rule that matches p[3].

In the **packet classification** problem, we wish to determine which rule of the router table is to be applied to a given packet. Data structures to represent one-dimensional router tables (i.e., tables in which every filter has a single field, which is typically the destination address of the packet being classified), have been extensively studied. These structures are reviewed in [21] and [22], for example. Although 1-dimensional prefix filters are adequate for destination based packet forwarding, higher dimensional filters are required for firewall, quality of service, and virtual private network applications, for example. Two-dimensional prefix filters, for example, may be used "to represent host to host or network to network or IP multicast flows" [14] and higher dimensional filters are required if these flows are to be represented "with greater granularity." Data structures for multi-dimensional (i.e., $k > 1$) packet classification are developed in [1, 2, 3, 7, 9, 10, 11, 14, 18, 20, 25, 27, 28], for example. In the sequel we use the terms rule and

[2]For example, the prefix 10* matches all destination addresses that begin with the bit sequence 10; the length of this prefix is 2.

[3]Let f and g be two filters. f is more specific than g iff every packet matched by f also is matched by g and there is at least one packet matched by g that is not matched by f

filter interchangeably because the filters in a rule-table are distinct and, in this paper, we are not concerned with the action associated with a rule.

In this paper, we propose a binary-search-on-levels (BSOL) scheme for multidimensional filters. Our scheme, like that of Waldvogel at al. [30], relies on an underlying trie whose levels are searched using the binary search method as proposed in [8]. The difference lies in how the trie is defined. With the definition we employ, our one-dimensional structure readily extends to the multidimensional case. The strategy employed by Waldvogel et al. [30] cannot easily be extended to multidimensional packet classification. The expected lookup complexity of our BSOL scheme is $O(\log W)$, where W is the sum of the maximum possible length, in bits, of each of the fields of a filter. For one-dimensional destination-address IPv4 prefixes, $W = 32$; for 2-dimensional IPv4 (source address, destination address) prefix filters, $W = 64$; and for 4-dimensional (source address, destination address, source port, destination port) filters in which each port number as at most 16 bits, $W = 96$.

In Section 2, we describe our BSOL scheme. Experimental results are presented in Section 3. Our results are summarized in the conclusion, Section 5. Section 4 describes past research related to the topic of this paper.

2 Binary Search On Levels (BSOL)

In section 2.1, we describe the BSOL scheme for one-dimensional classifiers. In Section 2.2, we show how the one-dimensional BSOL scheme may be generalized to two or more dimensions.

2.1 One-dimensional BSOL

Let $F = \{f_0, f_1, ..., f_{n-1}\}$ be a filter set, where each filter f_i is a range $[b_i, e_i]$[4]. For definiteness, assume that each range $[b_i, e_i]$ is a range of destination addresses. Our scheme works just as well if $[b_i, e_i]$ is a range port numbers etc. Let $f_{default} = [0, 2^W - 1]$ if any. Since $f_{default}$ matches all packets, we can safely remove all filters whose priority is lower than that of $f_{default}$. $f_{default}$ does not need to be stored in BSOL. Any packet that has no matching filter in BSOL is automatically matched by $f_{default}$. From now on, we assume F does not contain $f_{default}$. We first map F into a

[4]Note that every prefix may be represented as a range.

trie. For simplicity we describe a mapping that uses a 1-bit trie. Other varieties of tries (such as a fixed-stride trie) also may be used. With each node z of the trie, we associate an interval $z.int$ in the destination address space $[0, 2^W - 1]$[5]. The interval $root.int$ associated with the root of the trie is $[0, 2^W - 1]$. The intervals associated with the children, if any, of the root are obtained by dividing $root.int$ into two equal parts, $[0, 2^{W-1} - 1]$ and $[2^{W-1}, 2^W - 1]$. The former is associated with the left child of the root, and the latter is associated with the right child of the root. In general, in a 1-bit trie, the interval associated with a node is half that associated with its parent (if any).

Each node z of the trie has a list, $z.POList$, of ranges that partially overlap $z.int$, a field $z.bmr$ that stores the best (i.e., highest priority) range that matches all of $z.int$, and a field $z.bs$ that gives the path from the trie root to z. To construct the 1-bit trie for F, we begin with a root node $z = root$, $z.int = [0, 2^W - 1]$, and $z.bs = null$. Set $root.POList = F$ and $root.bmr = null$. If the number of ranges in $z.POList$ is more than a predefined constant T, node z is split into a left and right child; the $POList$, bmr, bs and int values for the left and right children of z are determined from the information associated with node z. For example, the bs value of the left child of z is $z.bs||0$ (0 is attached to $z.bs$) and that of the right child is $z.bs||1$. The 1-bit trie is constructed by splitting nodes until each leaf has a $POList$ with at most T ranges. Once we have the 1-bit trie, we construct a collection of hash tables H_0, H_1, ..., H_h, where h is the height of the trie. H_i contains the leaves in level i of the trie together with markers for some of the leaves in levels $j > i$. Each leaf places a marker in those hash tables that lie on the path taken by a binary search (to be described later). The hash-table key used by a leaf or marker in H_i is the first i bits of the bs value of the leaf in H_i or of the leaf that placed the marker in H_i. Figure 1 gives the algorithm to construct the 1-bit trie for F as well as the collection of hash tables. $pri(r)$ returns the priority of range r. $pri(null)$ is less than $pri(r)$ for $r \neq null$. Figure 2 gives the algorithm used by each leaf to determine the levels into which it needs to place a marker.

Figure 3 shows the 1-bit trie when $W = 4$, $T = 2$ and the filter-set ranges are $\{r1, r2, r3, r4\}$ $= \{[0, 2], [3, 5], [4, 7], [9, 13]\}$. The priority of range ri is $5 - i$ (i.e. the first matching tie breaker). The int value for each node is shown just above the node. Since $W = 4$, $root.int = [0, 15]$. The

[5]For IPv4 destination addressees $W = 32$ and for IPv6 addresses $W = 128$

```
**Algorithm** $buildBSOL1D(F)\{$
    $root$ = **new** $TrieNode$;
    $root.POList = F$;
    $root.bmr = null$;
    **if**( $|root.POList| > T$) $split(root)$;
    Build hash tables $H_0$, $H_1$, ..., $H_h$ as described;
    **return** $(H_0, H_1, ..., H_h)$;
}

**Algorithm** $split(TrieNode\ z)\{$
    $z.left$ = **new** $TrieNode$;
    $z.right$ = **new** $TrieNode$;
    $z.left.bmr = z.right.bmr = z.bmr$;
    **for**(each range $r$ in $z.POList$){
        **if**($r$ contains $z.left.int$ and $pri(r) > pri(z.left.bmr)$)
            $z.left.bmr = r$;
        **if**($r$ contains $z.right.int$ and $pri(r) > pri(z.right.bmr)$)
            $z.right.bmr = r$;
    }
    **for**(each range $r$ in $z.POList$){
        **if**($r$ partially overlaps $z.left.int$ and $pri(r) > pri(z.left.bmr)$
            append $r$ to $z.left.POList$;
        **if**($r$ partially overlaps $z.right.int$ and $pri(r) > pri(z.right.bmr)$
            append $r$ to $z.right.POList$;
    }
    **if**( $|z.left.POList| > T$) $split(z.left)$;
    **if**( $|z.right.POList| > T$) $split(z.right)$;
}
```

Figure 1: Build one-dimensional BSOL

remaining values associated with a node are shown only for the leaves. The 3-field box outside each leaf shows the *bs* (top field), *bmr*, and *POList* values for the leaf. Since each node has a unique *bs*, we can identify each node by its *bs* value. For example, the left child of the root is *node*0 and the node whose *int* is [0, 3] is *node*00. Notice that $node00.bs = 00$, $node00.bmr = null$, and $node00.POList = \{r1, r2\}$.

The trie of Figure 3 is quite similar to a leaf-pushed trie. The essential differences between our trie and a leaf-pushed trie are a) our trie may be used with ranges while a leaf-pushed trie is used only with prefixes and b) our trie uses a threshold $T \geq 1$ to stop the partitioning process while,

```
**Algorithm** $leaveMarkers(leafLevel)\{$
    $left = 0;$
    $right = h - 1;$
    **while**$(left \leq right)\{$
        $i = \lfloor (left + right)/2 \rfloor;$
        **if**$(i < leafLevel)\{$
            Leave a marker at level $i$;
            $left = i + 1;$
        $\}$
        **else**
            $right = i - 1;$
    $\}$
$\}$
```

Figure 2: Determine marker levels for a leaf at level $leafLevel$

in a leaf-pushed trie, $T = 1$.

In a binary search of H_0, H_1, and H_2, H_1 is searched first, then either H_0 or H_2 is searched. Since H_1 is on the search path to H_2, leaves at level 2 of the trie place markers in H_1. The marker key is a 1-bit key, the first bit of the *bs* value of the leaf that places a marker. In our case, both $node00$ and $node01$ place a marker in H_1 with key 0; in reality only one distinct marker gets placed in H_1. The marker in H_1 is shown in Figure 3 by shading the level 1 node ($node0$) that is an ancestor of the leaf (leaves) placing the marker. For our example, $H_0 = null$, $H_1 = \{(0, marker), (1, leaf)\}$, and $H_2 = \{(00, leaf), (01, leaf)\}$. The first component of a hash-table tuple is the key and the second is either a marker or a leaf. A marker is represented by a special symbol and a leaf by the two fields bmr and $POList$.

Figure 4 shows the algorithm to search the hash tables H_0, H_1, $\cdots$, H_h for the best matching-filter for the destination d. The algorithm uses the binary-search method to determine the H_i that contains the best matching-filter for d. Its correctness follows from the construction of the H_is.

We consider two examples to illustrate the working of the BSOL search algorithm. First, to find the best matching-filter for $d = 9$ (the binary representation of 9 is 1001) in Figure 3, we start by searching H_1 using the first bit of 9. We find a matching entry $(1, leaf)$. Since it is a leaf, the binary search stops here. The bmr and $POList$ fields are examined and $r4$ is determined

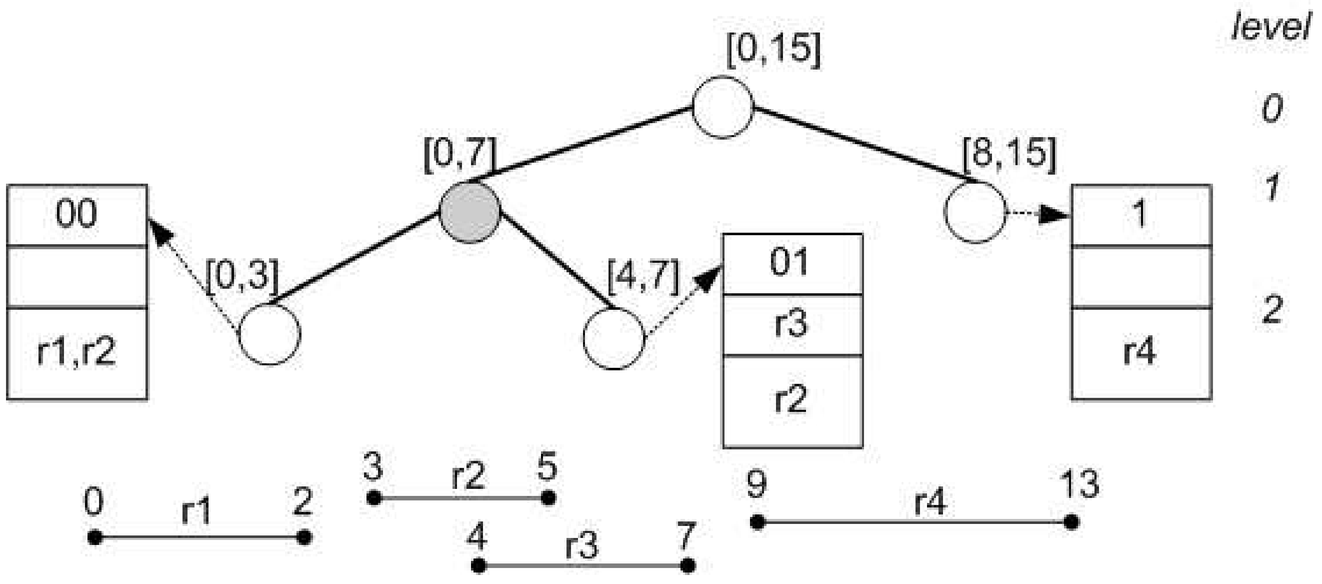

Figure 3: Example trie for one-dimensional filters ($T = 2$, $W = 4$)

```
**Algorithm** $search(d)${
    $left = 0$;
    $right = h$;
    **while**($left \leq right$){
        $i = \lfloor (left + right)/2 \rfloor$;
        Search $H_i$ using the first $i$ bits of $d$;
        **if**(searching $H_i$ returns $null$)
            $right = i - 1$;
        **else if**(searching $H_i$ returns a marker entry)
            $left = i + 1$;
        **else** //searching $H_i$ returns a leaf entry
            Search this entry and **return** the best matching-filter for $d$;
    }
}
```

Figure 4: Search for best matching-filter in one-dimensional BSOL

to be the best matching-filter. Second, to find the best matching-filter for $d = 5$ (0101 in binary), we use the first bit of d to find the entry $(0, marker)$ in H_1. Since this is a marker, we move to H_2 and search H_2 using the first two bits (01) of 5. This search returns the entry $(01, leaf)$. Searching the leaf entry returns $r2$ ($r3$ also matches 5 but the priority of $r3$ is lower than that of $r2$).

Figure 5 shows the trie for the same set of ranges as used in Figure 3. However, this time $T = 1$. For this case, $H_0 = \{(null, marker)\}$, $H_1 = \{(1, leaf)\}$, $H_2 = \{(00, marker), (01, leaf)\}$, $H_3 = \{(000, leaf), (001, marker)\}$, and $H_4 = \{(0010, leaf), (0011, leaf)\}$. To find the best matching-filter for $d = 9$, we start the binary search at H_2 using the first two bits of 9. There is no matching entry in H_2. So the binary search moves to H_0 and finds $(null, marker)$. Since it is a marker, there may be a matching leaf at a lower level. We move to H_1 and find $(1, leaf)$. Searching this entry returns $r4$.

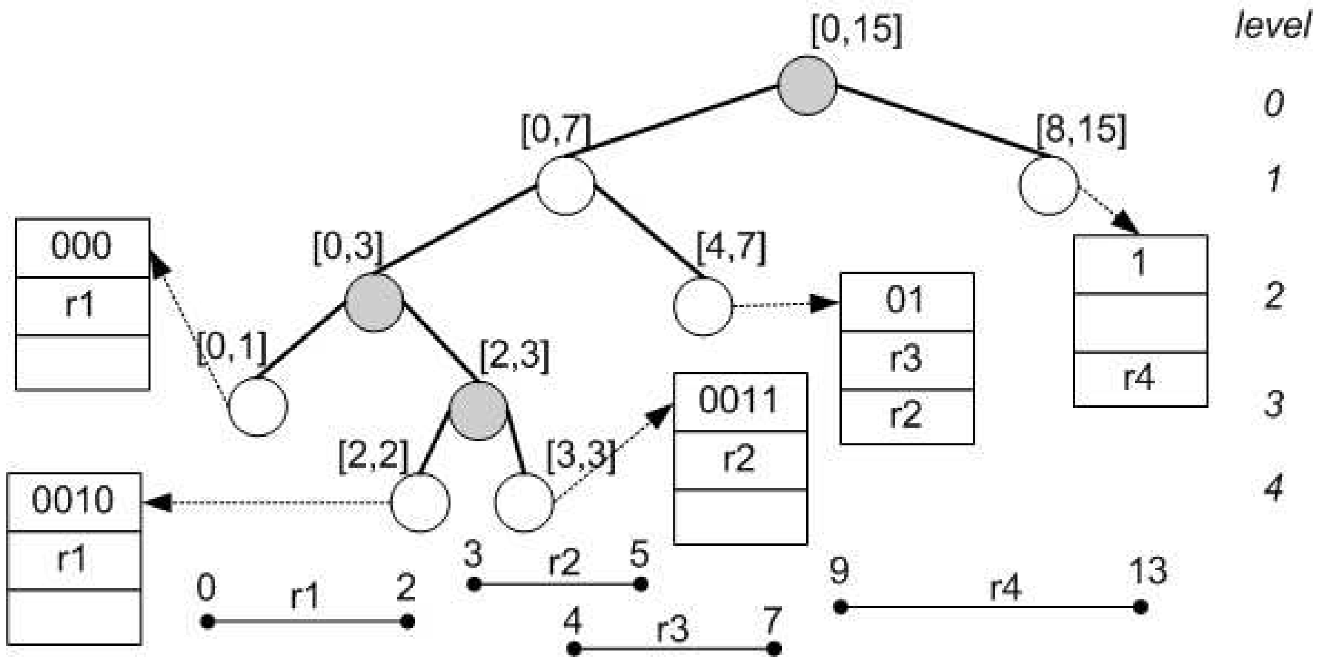

Figure 5: Example trie for one-dimensional filters ($T = 1$, $W = 4$)

It is important to note that the 1-bit trie is used only to explain the construction of the H_is. It is only the H_is that are explicitly stored in our router-table data structure.

Correctness of the scheme

First observe that a search of the underlying trie always terminates at a leaf and that this leaf contains adequate information to determine the highest priority filter that matches the given destination address. Let L be the leaf at which the serach for destination address d terminates. To prove the correctness of our search algorithm of Figure 4, we need to show that it reaches the proper leaf L of the underlying trie. At the start of the algorithm, $left = 0$ and $right = h$. So, L is at a level between $left$ and $right$. The search maintains this invariant. Note that H_i has an entry for each node at level i of the underlying trie. So, when H_i is searched using the first i bits of the destination address d, the search returns *null* iff the underlying trie has no node at level i that corresponds to the first i bits of d. In this case, the search of the underlying trie must terminate at a level j, $j < i$ (i.e., L is at a level $< i$). Hence, the search algorithm of Figure 4 correctly sets $right$ by $i - 1$. When the search of H_i returns a non-null value, the underlying trie has a node N at level i that corresponds to the first i bits of d. If the found entry in H_i is a marker, then N is not a leaf and so the level of L is greater than i. In particular, the level of L must be between $i + 1$ and $right$. If the found entry in H_i is not a marker, N is a leaf and by the invariant and the fact that N matches the first i bits of d, $N = L$.

Comparison with scheme of Waldvogel et al. [30]

The binary search on hash tables scheme differs from the BSOL scheme proposed by us in the following ways:

1. The scheme of [30] works only for prefix filters whereas ours applies to general ranges.

2. Hash table H_i in the scheme of [30] consists of prefixes whose length is i plus markers for longer-length prefixes for which H_i is on the binary-search path. The marker placement scheme is the same for our BSOL hash tables and the hash tables of [30].

3. In the scheme of [30], each marker in H_i records the longest length prefix whose length is less than i and which matches the marker. In our scheme we do not record any such information. Our markers do not need this information because, in BSOL, every packet is classified by searching for the appropriate leaf of the underlying leaf-pushed trie; this leaf contains all the information needed to classify the packet. The scheme of [30] may be thought of as based on a non-leaf-pushed trie. In such a trie, the information in nodes on the search path to a leaf is needed to properly classify a packet. For example, the longest prefix that matches a given destination address is the last prefix encountered on this path (this prefix may not be in the reached leaf). This information is encoded into the markers in the scheme of [30].

4. The scheme of [30] requires $O(n \log W)$ space. To determine the space requirements of BSOL, note that each level of our trie has $O(n)$ nodes. This follows from the observation that each filter can be in the $POList$ of at most 2 nodes on the same level. Hence, each level can have at most n nodes with a $POList$ whose size is at least 2. Since a node with a $POList$ whose size is ≤ 1 must be a leaf, each level can have at most $2n$ nodes (as there are at most n parents possible for these nodes). Since the number of levels in our trie is $O(W)$, the total number of nodes in our trie is $O(nW)$. The number of entries in H_i equals the number of nodes at level i of our trie. So, the total space needed for all of our hash tables is $O(nW)$. Even though the worst-case space complexity our our scheme is more than that of [30], we note that it has been observed that the tries for practical 1D classifiers have fewer than $2n$ nodes. Hence, for practical 1D classifiers, the space required by our scheme is $O(n)$ while that for the scheme of [30] remains $O(n \log W)$.

5. Both schemes have an $O(\log W)$ lookup complexity.

6. The BSOL scheme is easily extended to higher-dimension classifiers retaining its $O(\log W)$ complexity (this analysis regards the number of dimensions as a constant). We know of no way to extend the scheme of [30] to higher dimensions and obtain a search complexity of $O(\log W)$.

2.2 Two-dimensional BSOL

Let $F = \{f_0, f_1, ..., f_{n-1}\}$ be a set of 2-dimensional filters. Each filter $f_i = (X(f_i), Y(f_i))$ is a two-dimensional rectangle and $X(f_i)$ and $Y(f_i)$ are, respectively, the projections of f_i on the x and y axes. Table 1 shows a set of 7 filters and Figure 6 shows these 7 filters as rectangles.

We again use a trie to explain the construction of the hash tables on which a binary search is performed to find a best matching-filter. With each node z of the trie, we associate a rectangle $rect(z)$ in two-dimensional space $[0, 2^{W_1} - 1] \times [0, 2^{W_2} - 1]$. Here W_i is the maximum possible number of bits for the dimension i address. For our example of Table 1, $W_1 = W_2 = 4$. When the two fields of each filter represent the source and destination addresses of an IPv4 packet, $W_1 = W_2 = 32$.

The nodes of our trie have the same fields as for the 1-dimensional case except that the field

Filter f	$X(f)$	$Y(f)$	$pri(f)$
$f1$	[3, 14]	[13, 15]	7
$f2$	[5, 10]	[10, 12]	6
$f3$	[1, 3]	[6, 7]	5
$f4$	[4, 6]	[6, 7]	4
$f5$	[1, 5]	[4, 5]	3
$f6$	[1, 5]	[1, 2]	2
$f7$	[0, 12]	[0, 3]	1

Table 1: Two-dimensional filters

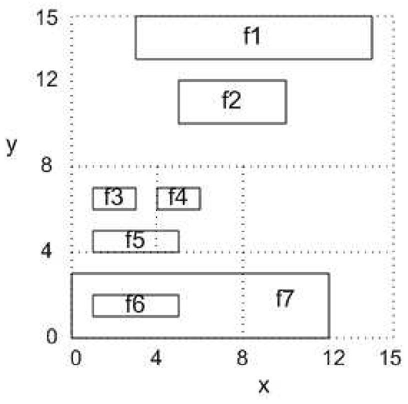

Figure 6: Rectangular representation ($W_1 = W_2 = 4$)

int is replaced by the field *rect* (rectangle). For our trie, $root.rect = [0, 2^{W_1} - 1] \times [0, 2^{W_2} - 1]$, $root.bs = null$, $root.bmr = null$, and $root.POList = F$. A trie node whose *POList* is larger than a prespecified threshold T is split into two children by splitting its rectangle in either the x direction or y direction. *For all nodes at a given level of the trie, the split direction is the same.*

Figure 7 shows the trie that results when the filters of Table 1 are mapped into a trie using the threshold $T = 2$. In this figure, the nodes at levels 0 (root) and 2 are split along the y direction while those at levels 1 and 3 are split along the x direction. The split directions from level 0 to $h - 1$ are concatenated to form a ***partition sequence***, where h is the height of the trie ($h = 4$ in Figure 7). The partition sequence for Figure 7 is $yxyx$. The partition sequence for Figure 8, which also is for the filters of Figure 6, is yyy.

The algorithm to construct the hash tables for our two-dimensional BSOL scheme is the same as that in Figure 1 except that *int* is replaced by *rect*.

The hash tables that correspond to Figure 7 are $H_0 = \{(null, marker)\}$, $H_1 = \{(1, leaf)\}$, $H_2 = \{(00, marker), (01, leaf)\}$, $H_3 = \{(000, leaf), (001, marker)\}$, and $H_4 = \{(0010, leaf), (0011, leaf)\}$. The hash tables for Figure 8 are $H_0 = null$, $H_1 = \{(0, marker), (1, leaf)\}$, $H_2 = \{(00, leaf), (01, marker)\}$, and $H_3 = \{(010, leaf), (011, leaf)\}$.

To determine the best matching-filter for a packet (a, b), we first construct d based on the partition sequence. For example, if the partition sequence is $yxyx$, we use the first bit of b as the first bit of d, the first bit of a as the second bit of d, the second bit of b as the third bit of d, and the second bit of a as the fourth bit of d. So $d = 1001$ for the packet $(6, 11)$ ((0110, 1011) in

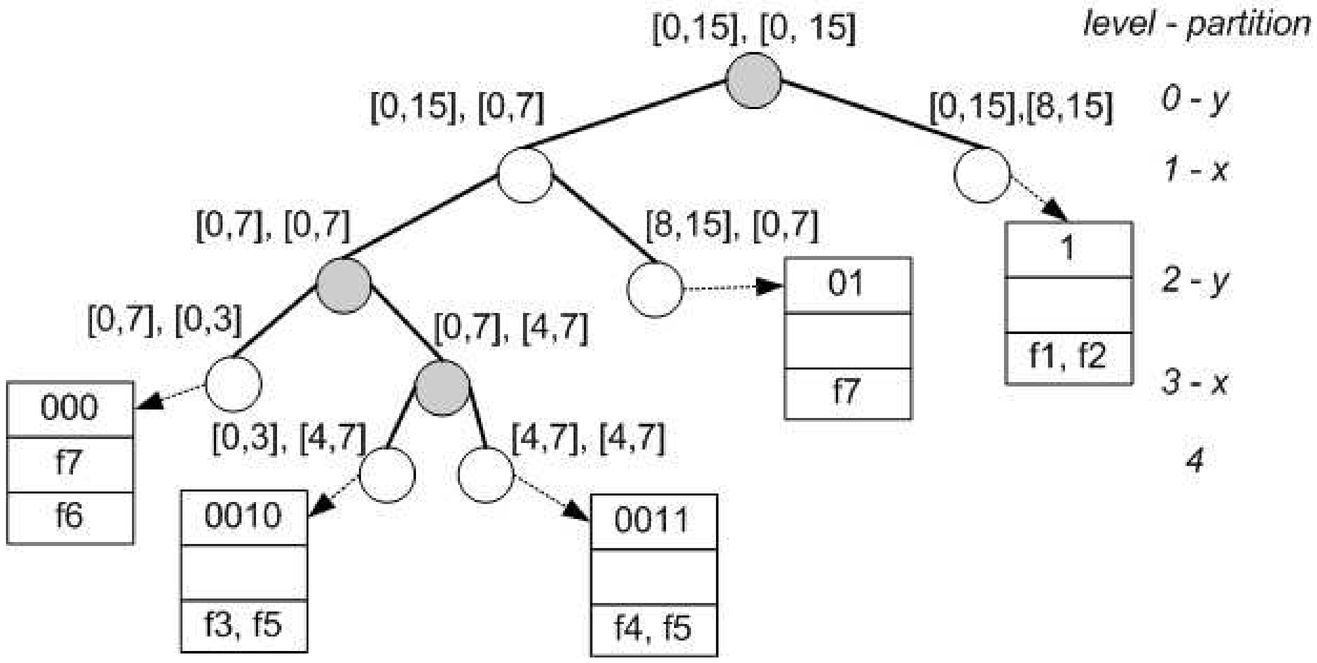

Figure 7: Trie for two-dimensional filters with partition sequence $yxyx$ and $T = 2$

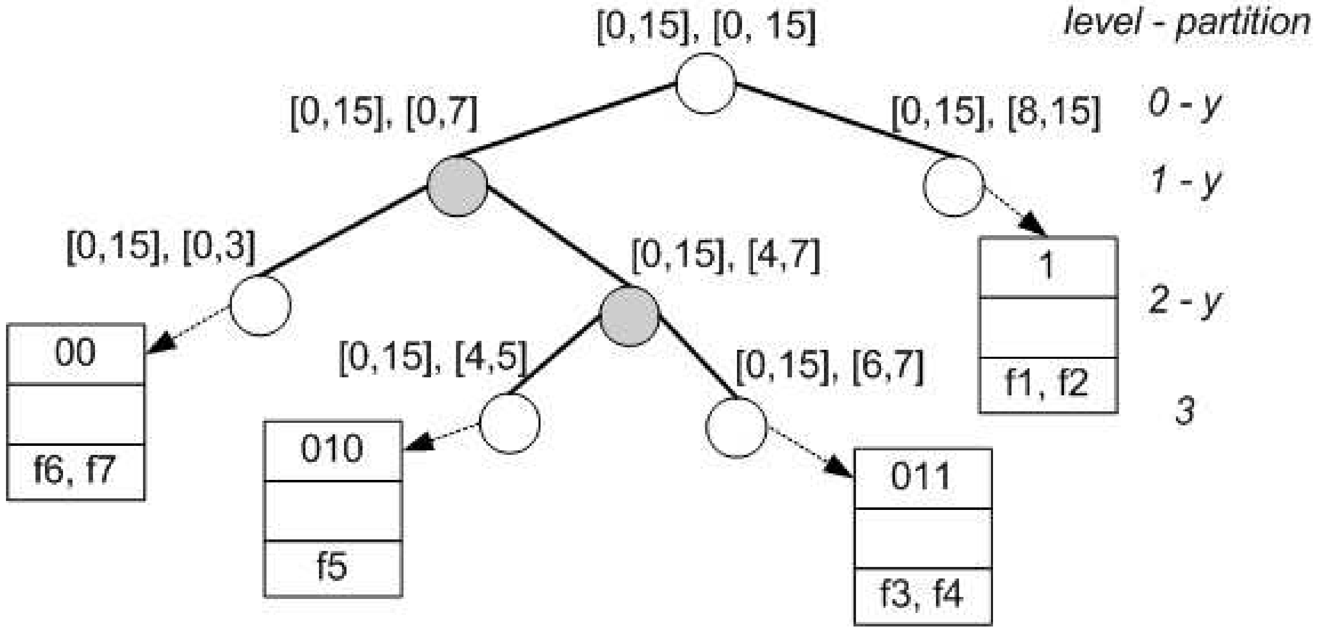

Figure 8: Trie for two-dimensional filters with partition sequence yyy and $T = 2$

binary). If the partition sequence is yyy, we use the first three bits of b as the first three bits of d. So $d = 101$ for the packet $(6, 11)$.

Once we have constructed d as described, the 1-dimensional search algorithm of Figure 4 is used to determine the best matching-filter for d. For example, to determine the best matching-filter for $(6, 11)$ using the hash tables for Figure 7, we construct $d = 1001$ and start the binary search at H_2 using the first two bits of d. The search for 10 in H_2 returns null. We move to H_0 and find $(null, marker)$. Since this is a marker, there is a possibility of finding a matching leaf at a lower level of the trie. Hence we move to H_1 and search H_1 using the first bit of d. There is a match $(1, leaf)$. Searching this leaf returns $f2$. As a second example, suppose we search for the best matching-filter for $(4, 1)$ (the binary representation is $(0100, 0001)$). We construct d as 0001 and start the binary search at H_2. Searching H_2 returns $(00, marker)$. So the search moves to H_3. The search of H_3 returns $(000, leaf)$. From the returned hash-table entry, we determine that $f6$ is the best matching-filter (although $f7$ also matches $(4, 1)$, it has a lower priority than $f6$).

Heuristic for selecting partition sequence

From Figures 7 and 8 we see that different partition sequences result in tries that have a different number of leaves and in which each filter is stored in a different number of leaves. Hence different partition sequences result in different space requirement. The trie for the partition sequence yyy has four leaves, and each filter is stored only once. The trie for the partition sequence $yxyx$ has five leaves, and filter $f5$ is stored in $node0010$ and $node0011$.

The following heuristic can be used to select the partition direction at each trie level to minimize $\sum_{z \text{ is a leaf}} |z.POList|$. Let NTP be the current set of leaf nodes that need to be partitioned during the BSOL construction procedure. Initially $NTP = \{root\}$ if $|root.POList| > T$. Let $NB(z, k)$ be the number of filters that will be broken (i.e., the partition line passes through the filter) if the direction k is chosen to partition node $z \in NTP$. Let $NB(k) = \sum_{z \in NTP} NB(z, k)$. The direction k is chosen for partitioning/splitting at the current level such that $NB(k)$ is the least.

For example, in Figure 6, initially, $NTP = \{root\}$, $NB(x) = 3$ since filters $f1, f2, f7$ are broken when we split/partition along the x direction, and $NB(y) = 0$. Thus the y direction is used to partition at level 0. After partitioning at the root, we move to level 1, $NTP = \{node0\}$, $NB(x) = 1$ since only filter $f7$ is broken by an x direction partition, and $NB(y) = 0$. The y direction is chosen again. At level 2, $NTP = \{node01\}$, $NB(x) = 1$, and $NB(y) = 0$. Therefore level 2 also uses y direction for partitioning.

2.3 k-dimensional BSOL ($k > 2$)

Extending BSOL to $k > 2$ dimensions is straightforward. We omit the details here. Once again we obtain a collection of hash tables based upon a trie-mapping of the filters. A search key d is constructed from data obtained from the packet that is to be classified as well as from the partitioning sequence used to construct the trie. The hash tables are searched using d and the binary search algorithm of Figure 4.

Lemma 1 *Using the BSOL scheme multidimensional packet classification can be done performing $O(\log h)$ hash-table searches plus a search in at most one POList. The number of hash tables is $h + 1$, where h is the height of the trie used to construct the hash tables.*

Proof True since there are $h + 1$ levels and a binary search on levels is performed. ■

Notice that $h \leq W = \sum_{i=1}^{d} W_i$, where W_i is the number of bits in ith dimension. For 4-dimensional filters (IPv6 source address range, IPv6 destination address range, source port range, destination port range) with 16-bit port numbers, $h \leq 288$ and at most 9 hash-table searches are needed to reach a leaf. The reached leaf may be searched serially in $O(T)$ time.

Lemma 2 *Let m be the number of leaves in the trie for BSOL. The space requirement of BSOL is $O(m(T + \log h))$, where h is the height of the trie.*

Proof Each leaf leaves at most $\log h$ markers and each leaf requires $O(T)$ space to store. ■

Reduce the number of hash tables

Before the markers are generated, the trie levels with no leaf nodes can be removed. We call the level with at least one leaf node ***non-empty levels***. Let l be the number of non-empty levels. We only need l hash tables instead of $h + 1$, and multidimensional packet classification can be done by performing $O(\log l)$ hash-table searches plus a search in at most one *POList*. Other improvement similar to that in [26, 17] may be applied to further reduce the value of l.

3 Experimental Results

3.1 Comparison with the 1D scheme of Waldvogel et al. [30]

Waldvogel et al. [30] have proposed a scheme for one-dimensional packet classification. Like our scheme, their scheme employs hash tables that are searched using a binary search. We first compare the memory requirements of their scheme and that of our proposed scheme BSOL1D for one-dimensional classification. For this comparison we use use four IPv4 prefix databases obtained from [19]. The databases MaeWest and Aads were obtained on Nov 22, 2001 and Pb and Paix were obtained Sept. 13, 2000. In our memory analysis we assume that memory is bit addressible; so we count the number of bits required.

For the scheme of Waldvogel et al. [30], let Hi be the hash table for the prefixes and markers whose length is $pLen(i)$. Each entry in Hi is a triple $(key, flag, action)$, where key may be a marker or a prefix. The length of key is $pLen(i)$ bits and that of $action$ is assumed to be 8 bits. The $flag$ field is one bit ($flag = 0$ for a prefix and 1 for a marker). For a marker, $action$ is the action that corresponds to the longest matching prefix of the marker. Thus, each entry of Hi requires $pLen(i) + 1 + 8$ bits. So the total number of bits required by Hi is $size(Hi) * (pLen(i) + 9)/loadFactor$, where $loadFactor$ is the hash table loading density.

For BSOL1D, each entry of Hi is a pair $(key, leafIndex)$. The length of key is still $pLen(i)$ bits. $leafIndex$ is $null$ for a marker. For a leaf, $leafIndex$ serves as an index into a list of the leaves of Hi. When bmr is not $null$, the leaf is a 4-tuple $(1, bAction, POListSize, POList)$, where $bAction$ is the action that corresponds to the bmr for this leaf. When bmr is $null$, the leaf is a triple $(0, POListSize, POList)$. It is enough to give $POListSize$ $\lceil \log_2 T \rceil$ bits. Each entry of $POList$ is a triple $(prefixLength, prefixValue, action)$. $prefixLength$ is 5 bits for IPv4 We only need $POList[j].length - pLen(i)$ bits for $prefixValue$ of the jth prefix in the $POList$ since the key field gives the first $pLen(i)$ bits. Let $leafStructSize$ be the total number of bits required to store the leaves of Hi. Then $\lceil \log_2 leafStructSize \rceil$ bits are enough for $leafIndex$. Hi also needs a 32 bit pointer pointing to the leaf structure and 32 bits for $leafStructSize$.

Table 2 gives the memory required for the scheme of [30] and BSOL1D for different values of T. As noted in [30], hash tables need be constructed only for those lengths for which there is at least one prefix. Once the lengths for which hash tables will be constructed are determined, the binary

IPv4 Prefix Database		MaeWest	Aads	Pb	Paix
Num of Prefixes		28889	31827	35303	85988
Scheme of [30] (Kbits)		3829	4536	4983	11186
BSOL1D (Kbits)	T=1	4228	4307	5159	13254
	T=2	2699	2841	3401	8634
	T=3	2167	2342	2663	6745
	T=4	1742	1919	2124	5276
	T=5	1534	1702	1883	4697
	T=10	1050	1157	1277	3137

Table 2: Memory requirement (in Kbits) of BSOL1D and [30]. $loadFactor = 0.5$.

search paths[6] (and hence the location of markers) are well defined. The same observation applies to BSOL, hash tables need be maintained only for those trie levels that have at least one leaf node. Once these levels are determined, the binary search paths (and hence the location of markers) are well defined. Table 2 accounts for the elimination of hash tables for the stated prefix lengths and trie levels. The reported memory requirements are for the case when $loadFactor = 0.5$, i.e., each hash table is half-full and the longest-matching tie breaker is used. With $T = 1$, BSOL1D takes about 10% more memory than does the scheme of [30] for MaeWest, 18% more for Paix, 3.5% more for Pb, and 5.3% less for Aads. With $T = 3$ or 4, BSOL1D takes about half as much memory as taken by [30].

With respect to memory accesses, BSOL1D with $T = 1$ makes the same number of hash-table lookups as does the scheme of [30]. When $T > 1$, BSOL1D needs an extra memory access to get to the relevant leaf of Hi (for small T and modestly large cache lines, this leaf may be searched with a single memory access). Note that if the reduced memory required by BSOL1D (for $T > 1$) enables one to store all the hash tables in fast memory, say SDRAM, versus using slower memory for the structure of [30], BSOL1D would provide faster lookup.

3.2 Memory requirement of BSOL as a function of T

We experimented with two four-dimensional classifiers, e.l1 and e.l2, available from [12] [7]. The filter component of a rule is a 4-tuple (w, x, y, z) = (source address prefix, destination address prefix, protocol, port number range). The classifiers e.l1 and e.12, respectively, have 473 and 508

[6]Now the binary search is done over the set of prefix lengths for which we have a hash table.

[7]Despite significant effort, we were unable to get the large real classifiers used by others in their research.

filters. The remaining classifiers used in [12] have less than 50 rules and were considered too small for our experiments.

Tables 3 and 4 give, for different values of T, the memory requirement, the number of leaves, trie height, and the number of levels l that have at least 1 leaf node. Clearly the choice of T greatly affects the memory requirement of BSOL (T plays an important role in determining the extent of the splitting of the same rule into multiple leaves.)

T	7	8	9	10	11	15	20
Num of leaves	49,533	12,173	394	325	313	229	177
Trie height	71	68	63	63	59	58	58
l	63	59	55	55	51	50	49
Memory(Kbits)	19,387	5,185	162	140	137	107	93

Table 3: Classifier e.l1 (473 filters). Load factor of hash table is 0.5.

T	7	8	9	10	11	15	20
Num of leaves	74	73	73	71	70	21	19
Trie height	59	58	58	58	58	25	25
l	51	50	50	50	50	17	18
Memory(Kbits)	22	22	22	22	22	9	9

Table 4: Classifier e.l2 (508 filters). Load factor of hash table is 0.5.

Let B be the maximum number of filters that match any packet when only the source and destination components of a filter are considered. Baboescu et al. [3] have observed that in real-world classifiers, $B < 20$. A possible heuristic to select T is to initially set $T = B$, and then increase (decrease) T if the memory requirement is higher (lower) than the memory budget.

Since $T = 20$ normally works fine, we use $T = 20$ for the rest of our experiments.

Note that to classify a packet, we make $\log_2 l$ hash-table searches plus a linear search in a leaf node.

3.3 Comparison with the kD scheme of Baboescu et al. [3]

Baboescu et al. [3] have proposed the EGT-PC (Extended Grid-of-Tries with Path Compression) scheme for multidimensional packet classification. The code for EGT-PC is available from [16].

To obtain a fair comparison of the memory requirements of BSOL and EGT-PC, we discarded the fields of the trie nodes of EGT-PC that are not needed for lookups. The node structure used by us to determine the memory requirement of EGT-PC is given in Figure 9. Each trie node uses left (zero) and right (one) pointers to locate its children. Since we know the total number of trie nodes, $numNodes$, in the trie, we allocate $\lceil \log_2 numNodes \rceil$ bits to each child pointer rather than the standard 32-bits per pointer. The field *pdimList* points to the filters stored in the node of a bottom-level trie. Although the code of [16] uses a linked list to store the rules in a node of the bottom-level trie, for our space analysis we use a more efficient representation–an array of rules together with the number of rules in the array. For each rule, only the protocol and port range fields plus the action need to be stored. The protocol field takes 8 bits and the port number range takes 32 bits (16 bits for start point, and 16 bits for finish point).

```
//Total bits: 88 + 3 * $\lceil \log_2 numNodes \rceil$ + max{$\lceil \log_2 numNodes \rceil$ , $\lceil \log_2 pdimListSize \rceil$}.
//dest points to the bottom-level trie and is needed for top-level trie nodes.
//pdimList points to the filter list and is needed for bottom-level trie nodes.
struct TRIENODEC{
    unsigned char level; //8 bits
    unsigned int zmask; //32 bits
    unsigned int omask; //32 bits
    unsigned char zmaskLen; //8 bits
    unsigned char omaskLen; //8 bits
    struct TRIENODEC *zero; //$\lceil \log_2 numNodes \rceil$ bits
    struct TRIENODEC *one; //$\lceil \log_2 numNodes \rceil$ bits
    struct TRIENODEC *fail; //$\lceil \log_2 numNodes \rceil$ bits
    union{
       struct TRIENODEC *dest; //$\lceil \log_2 numNodes \rceil$ bits
       struct TRIESUBNODE* pdimList; //$\lceil \log_2 pdimListSize \rceil$ bits
    };
};
```

Figure 9: EGT-PC node structure. Search-unrelated fields are removed from the code at [16]

For a multidimensional BSOL, the data structure is the same as that for the one-dimensional BSOL except that each entry of *POList* is a filter plus the associated action. Since *key* already gives us some number of leading bits of the filter, we can reduce the space required by a filter by not storing these bits again. Suppose that *key* uses *kProt* leading bits of the protocol field, *kSrcAddr* leading bits of the source address, $kDstAddr$ leading bits of the destination address, and $kPort$ leading bits of the port number. For the jth filter in $POList$, the protocol field needs $8 - kProt$ bits, the source address prefix takes 5 bits for prefix length and $\max\{0, POList[j].srcAddrLen - kSrcAddr\}$ bits for the remaining bits of the prefix, the destination address prefix uses 5 bits for length and $\max\{0, POList[j].dstAddrLen - kDstAddr\}$ bits for the remaining bits of the prefix, and the port number range needs $2*(16-kPort)$ bits. BSOL makes $\log_2(l)$ hash-table lookups to reach a leaf, then a linear search of the leaf is performed. It is fair to use $T = 20$ for the memory requirement comparison between BSOL and EGT-PC.

Experiment on two 4D classifiers

Table 5 shows the memory required by our BSOL scheme and the EGT-PC scheme of [3] on our two four-dimensional classifiers, e.l1 and e.l2, available from [12]. M is the maximum number of filters that share the same source-destination prefix pair. The tie breaker used by us is the first matching rule. The hash tables of BSOL use a loading factor of 0.5 (half full). For the e.l1 classifier our BSOL structure takes half as much memory as required by the EGT-PC scheme; for e.l2, the memory required by BSOL is 3.6% that required by EGT-PC.

To estimate lookup performance, we use the center of each filter in each classifier as the query point and record the number of memory accesses during lookup. The maximum, minimum, average, and standard deviation of the number of memory accesses per lookup are given in Table 6. The cache line size affects lookup performance. Each EGT-PC node can be loaded with one cache miss. Depending on its size, searching the $pdimList$ stored in EGT-PC node may result in several cache misses. Since M is equal to 6 for both e.l1 and e.l2 and one 64 byte cache line can load 10 filters, changing cache line size from 64 bytes to 128 bytes does not affect the lookup performance of EGT-PC. However, BSOL lookup benefits from larger cache line sizes as BSOL may store up to T ($T = 20$ in test) filters in a leaf node. The number of memory accesses required by BSOL is 32% to 38% that required by EGT-PC on average. The variation and maximum number of memory accesses are also much smaller.

Classifier	# of filters	M	EGT-PC (Kbits)	BSOL (Kbits)
e.l1	473	6	186	93
e.l2	508	6	248	9

Table 5: Memory requirement of EGT-PC and BSOL. Load factor of hash table is 0.5. $T = 20$. M is the maximum number of filters who share the same source-destination prefix pair.

Cache Line Size	Classifier	Scheme	max	min	average	std
64 bytes	e.l1	BSOL	10	3	7.7	1.6
		EGT-PC	43	5	20.3	4.7
	e.l2	BSOL	9	5	6.2	0.7
		EGT-PC	39	7	17.5	4.6
128 bytes	e.l1	BSOL	8	2	6.6	1.3
		EGT-PC	43	5	20.3	4.7
	e.l2	BSOL	6	4	6.0	0.3
		EGT-PC	39	7	17.5	4.6

Table 6: The number of memory accesses per lookup of EGT-PC and BSOL. Load factor of hash table is 0.5. $T = 20$.

Experiment on 5D synthetic random classifiers

We experimented with several random five-dimensional classifiers generated by ClassBench [29]. Table 7 shows the memory required by our BSOL scheme and the EGT-PC scheme of [3]. Our BSOL scheme takes about 20% of the memory required by the EGT-PC scheme. On a Pentium 4 1.5GHz PC, the pre-processing time (i.e., the time to construct the necessary hash tables) of BSOL is less than 60 milliseconds for each of these five random classifiers. Table 8 (9) shows the number of memory accesses per lookup required by BSOL and EGT-PC when cache line size is 64 bytes (128 bytes). The number of memory accesses required by BSOL is 9% to 14% that required by EGT-PC, on average, when the cache line size is 128 bytes.

Experiment on 5D synthetic seed-based classifiers

We also experimented with several five-dimensional classifiers generated by ClassBench [29] based on the input seed files. The input seed file defines the characteristics of the original small classifiers. The large synthetic classifiers were generated using the command line *dbgenerator -bc seed 1000 2 0.5 -0.1*. Table 10 shows the memory required by our BSOL scheme and the EGT-PC scheme

# of filters	M	EGT-PC(Kbits)	BSOL(Kbits)
1,000	2	670	148
2,000	2	1,455	295
3,000	4	2,140	446
6,000	4	4,717	907
10,000	9	8,089	1,534

Table 7: Memory requirement of EGT-PC and BSOL for five-dimensional synthetic random classifiers generated using ClassBench [29]. Load factor of hash table is 0.5. M is the maximum number of filters who share the same source-destination prefix pair. $T = 20$.

# of filters	max		min		average		std	
	EGT-PC	BSOL	EGT-PC	BSOL	EGT-PC	BSOL	EGT-PC	BSOL
1,000	60	8	19	4	29.7	6.5	5.0	1.1
2,000	62	9	22	4	33.3	7.3	5.1	1.3
3,000	67	9	25	3	37.8	6.2	5.2	1.3
6,000	77	9	29	3	42.9	6.5	5.6	1.3
10,000	86	9	31	4	46.7	6.9	5.9	0.9

Table 8: The number of memory accesses per lookup of EGT-PC and BSOL for 5D synthetic random classifiers generated using ClassBench [29] when cache line size is 64-bytes. $T = 20$.

# of filters	max		min		average		std	
	EGT-PC	BSOL	EGT-PC	BSOL	EGT-PC	BSOL	EGT-PC	BSOL
1,000	60	5	19	3	29.7	4.1	5.0	0.6
2,000	62	6	22	3	33.3	4.8	5.1	0.8
3,000	67	6	25	2	37.8	3.9	5.2	0.8
6,000	77	6	29	2	42.9	3.9	5.6	1.0
10,000	85	6	30	3	45.7	4.1	5.9	0.5

Table 9: The number of memory accesses per lookup of EGT-PC and BSOL for 5D synthetic random classifiers generated using ClassBench [29] when cache line size is 128-bytes. $T = 20$.

of of [3]. On a Pentium 4 1.5GHz PC, the pre-processing time of BSOL is less than 1 second for all tested classifiers other than fw2 with 9,629 filters (2.5 seconds), ipc1 with 5,735 filters (6.2 seconds), and ipc1 with 9,505 filters (3.5 seconds). For synthetic classifiers based on the seed acl1, our BSOL scheme takes about 51% to 77% of the memory required by the EGT-PC scheme. For synthetic classifiers based on the seeds fw2 and ipc1, our BSOL scheme takes less memory than does the EGT-PC scheme on the first classifier, and more memory on the remaining classifiers (in fact, much more on the classifiers of size about 6,000 and 10,000). We looked into the classifiers based on seeds fw2 and ipc1 and found that these classifiers contain a significant number of mesh-like filters, i.e., filters whose source/destinaiton IP prefix pairs are (*, destination IP prefix) or (source IP prefix, *). Since BSOL recursively splits space, mesh-like filters cause significant memory increase. This is a weakness of space subdivision-based classification schemes such as as HiCuts and BSOL. EGT-PC does not suffer from such a space explosion when mesh-like filters are present because EGT-PC stores each filter only once.

One way to deal with the memory explosion of BSOL when mesh-like filters are present is to increase the value of T. An alternative is to divide the classifier into two subsets. BSOL2 divides the classifiers into two subsets. The first subset contains filters with source IP prefix length less than 5, and the second subset contains the remaining filters. BSOL2 then builds separate hash-table collections for each subset. The memory requirement of BSOL2 is shown in Table 10.

Table 11 (12) shows the number of memory accesses per lookup required by EGT-PC, BSOL and BSOL2 when the cache line size is 64 bytes (128 bytes). We list only the average and standard deviation due to space limitations. BSOL2 always outperforms EGT-PC. The number of memory accesses required by BSOL2 to classify a packet is less than twice that required by BSOL. This is because the leaves of BSOL are often full and require more memory accesses, whereas the leaves of BSOL2 are often not full. The performance of BSOL and BSOL2 benefits more from a large cache line than does that of EGT-PC. When hardware support to search the hash tables of BSOL and BSOL2 in parallel is available, BSOL2 will outperform BSOL on both memory requirement and lookup.

4 Related Work

Waldvogel et al. [30] have proposed a packet classification scheme for 1-dimensional prefix filters. Their scheme performs a binary search on hash tables organized by prefix length. Let W be the maximum possible length, in bits, of a prefix in the router table and let l be the number of different prefix lengths. Note that $W = 32$ for IPv4, $W = 128$ for IPv6, and $l \leq W + 1$. This binary search scheme proposed by Waldvogel et al. [30] has an expected complexity of $O(\log l) = O(\log W)$ for lookup. Srinivasan and Varghese [26] and Kim and Sahni [17] have proposed ways to improve

Seed	# of filters	M	EGTPC(Kbits)	BSOL(Kbits)	BSOL2(Kbits)
acl1	964	13	323	173	112
	1,914	23	581	299	253
	2,728	54	704	546	345
	5,543	35	1,508	1,037	635
	9,803	23	3,474	1,822	1,170
fw2	978	4	1,438	974	144
	1,943	4	3,237	3,707	289
	2,887	4	4,705	9,116	430
	5,800	4	9,121	35,342	845
	9,629	4	14,815	99,329	1,412
ipc1	977	5	678	375	144
	1,921	7	1,207	1,695	278
	2,863	8	1,822	4,301	422
	5,735	21	3,803	83,576	977
	9,505	20	6,371	83,786	1,935

Table 10: Memory requirement of EGT-PC, BSOL, BSOL2 for five-dimensional synthetic classifiers generated using ClassBench [29]. Load factor of hash table is 0.5. M is the max number of distinct source-destination prefix pairs matching a packet.

the performance of the binary-search-on-lengths scheme by using controlled prefix expansion to reduce the value of l. Braun et al. [4] implement the scheme of [30] in hardware. Broder and Mitzenmacher [6] proposed using multiple hash functions (two hash functions) to improve the lookup performance.

We note that Waldvogel's scheme is very similar to the k-ary search-on-length scheme developed by Berg et al. [5] and the binary search-on-length schemes developed by Willard [31]. Berg et al. [5] used a variant of stratified trees [8] for one-dimensional point location in a set of n disjoint ranges. Willard [31] modified stratified trees and proposed the y-fast trie data structure to search a set of disjoint ranges. By decomposing filter ranges that are not disjoint into disjoint ranges, the schemes of [5, 31] may be used for longest-prefix matching in static router tables. The asymptotic complexity for a search using the schemes of [5, 31] is the same as that of Waldvogel's scheme [30]. The decomposition of overlapping ranges into disjoint ranges is feasible for static router tables but not for dynamic router tables because a large range may be decomposed into $O(n)$ disjoint small ranges.

Seed	# of filters	average			std		
		EGTPC	BSOL	BSOL2	EGTPC	BSOL	BSOL2
acl1	964	33.4	8.3	12.5	11.2	1.8	1.8
	1,914	51.2	8.3	12.2	10.0	1.8	1.8
	2,728	42.8	8.7	13.8	11.0	1.9	1.9
	5,543	53.4	8.2	14.1	12.8	2.2	1.7
	9,803	49.0	8.6	14.8	11.4	2.2	3.0
fw2	978	31.4	8.2	12.1	8.1	1.3	2.7
	1,943	28.5	8.3	12.1	8.0	1.5	2.1
	2,887	33.2	7.4	12.9	8.9	1.4	2.6
	5,800	32.7	8.5	11.3	9.2	1.7	1.9
	9,629	29.3	8.1	10.9	6.8	1.3	1.8
ipc1	977	39.8	9.3	14.1	15.8	1.6	2.9
	1,921	42.2	9.9	14.8	16.3	1.8	2.9
	2,863	50.6	9.8	15.0	16.7	1.9	2.5
	5,735	54.6	9.8	16.6	17.2	1.7	2.8
	9,505	53.8	9.8	16.6	16.3	1.9	2.7

Table 11: The number of memory accesses per lookup of EGT-PC, BSOL and BSOL2 for five-dimensional synthetic classifiers generated using ClassBench [29]. The seeds are acl1, fw2 and ipc1. Cache line size is 64-bytes

It is challenging to design data structures for multidimensional router tables. The problem of point location in a static set of n non-overlapping k-dimensional hyper-rectangles requires $O(\log n)$ time with $O(n^k)$ memory requirement or $O((\log n)^{k-1})$ time with $O(n)$ memory requirement [13]. Multidimensional classification is no easier than the point location problem since filters can overlap.

Gupta et al. [13] review data structures for multidimensional router tables. Hierarchical tries requires $O(W^k)$ time for lookup with $O(kWn)$ memory requirement. Hierarchical tries can support update in $O(kW)$ time. Set pruning trees [24] support search in $O(kW)$ time with $O(n^k)$ memory requirement. Cross-producting [24] decomposes the lookup into k one-dimensional classifications and thus supports lookup in $O(kS)$ time, where S is the lookup time for one-dimensional classification. Cross-producting requires $O(n^k)$ memory, and is also suitable for range filters if the data structure used for one-dimensional classification also supports range filters. HiCuts [11] supports lookup in $O(W+T/C)$ time with $O(n^k)$ memory requirement, where T is the maximum bucket size and C is the cache line size.

Seed	# of filters	average			std		
		EGTPC	BSOL	BSOL2	EGTPC	BSOL	BSOL2
acl1	964	33.3	6.7	10.0	11.2	1.4	1.3
	1,914	51.0	7.0	9.9	9.8	1.2	1.3
	2,728	42.3	7.1	10.4	10.7	1.5	1.6
	5,543	53.0	6.7	10.7	12.6	1.6	1.3
	9,803	49.0	7.1	10.9	11.4	1.4	1.5
fw2	978	31.4	5.5	8.4	8.1	0.9	1.4
	1,943	28.5	5.6	8.4	8.0	0.9	1.0
	2,887	33.2	4.7	8.9	8.9	1.1	1.6
	5,800	32.7	5.7	8.2	9.2	1.2	1.1
	9,629	29.3	5.3	8.0	6.8	0.9	1.2
ipc1	977	39.8	6.6	9.7	15.8	1.2	1.8
	1,921	42.2	7.4	10.4	16.3	1.2	1.8
	2,863	49.6	7.3	11.2	16.7	1.4	1.6
	5,735	53.5	7.2	12.4	17.2	1.2	1.7
	9,505	52.8	7.2	12.7	16.3	1.6	1.7

Table 12: The number of memory accesses per lookup of EGT-PC, BSOL and BSOL2 for five-dimensional synthetic classifiers generated using ClassBench [29]. The seeds are acl1, fw2 and ipc1. Cache line size is 128-bytes

Baboescu et al. [3] have proposed a general strategy along with a specific data structure for multidimensional packet classification. The scheme of Baboescu et al. [3] relies on the following characteristic of real-world classifiers: "the number of distinct source-destination prefix pairs matching a packet is even less than 20" [3]. The general strategy proposed by Baboescu et al. [3] uses any two-dimensional search algorithm to find all the rules whose source-destination prefix pairs match the source-destination fields of the incoming packet. Then a linear search is performed on these rules to determine which match the remaining fields. Thus the time needed to determine the best matching filter for an incoming packet is the time to find all rules that match the source and destination fields of the packet plus the time to check these rules against the remaining fields.

The specific data structure proposed in [3] is the EGT-PC (Extended Grid-of-Tries with Path Compression). EGT-PC is a modified grid-of-tries [24] that employs path compression to reduce the search time and memory requirement. Let B is the number of distinct source-destination prefix pairs matching a packet. EGT-PC takes $O(W^2)$ time to search grid-of-trie to get all source-destination prefix pairs that match a packet's source-destination IP addresses. There are at most B pairs. Each source-destination prefix pair corresponds to a bucket. The bucket contains filters who share source-destination prefix pair. Let M the maximum number of filters who share the same source-destination prefix pair. Note that B and M is determined by the classifier, while T used in BSOL and HiCuts is selected by the user. EGT-PC then search B buckets for the best matching filter. Therefore, the time complexity of a lookup using EGT-PC is $O(W^2 + BM/C)$, where and C is the size of cache line size. EGT-PC has $O(n)$ memory requirement since it uses path-compressed trie and saves each filter only once.

Therefore, EGT-PC guarantees linear memory requirement but depends on the characteristics of real classifiers (i.e., M and B being a small value) to achieve reasonable lookup performance. In contrast, HiCuts guarantees $O(W + T/C)$ lookup performance and but depends on the characteristics of real classifiers to achieve a reasonable memory requirement. In this regard, our BSOL scheme is similar to HiCuts. BSOL also has $O(n^k)$ memory requirement. BSOL guarantees $O(\log W + T/C)$ lookup performance and but depends on the characteristics of real classifiers to achieve a reasonable memory requirement. Both HiCuts and BSOL uses space subdivision method to partition classifier into many small subsets of filters.

5 Conclusion

We have developed an $O(\log W)$ lookup scheme for multidimensional packet classification. Unlike the $O(\log W)$ scheme proposed by Waldvogel et al. [30] for one-dimensional prefix tables, our scheme works for both prefixes and ranges and multidimensional classifiers. For one-dimensional IPv4 tables, our scheme uses less memory than is used by the scheme of [30] provided $T > 1$. For multidimensional classifiers, our BSOL scheme provides a time advantage over the EGT-PC scheme proposed in [3]. BSOL scheme provides a memory advantage over the EGT-PC scheme on two four-dimensional classifiers available from [12], on random five-dimensional synthetic classifiers generated using ClassBench [29], and on some of the five-dimensional synthetic seed-based classifiers generated using ClassBench [29]. In [3] it is shown that the EGT-PC scheme has a memory advantage over other competing schemes for multidimensional classifiers.

References

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

[2] F.Baboescu and G.Varghese, Fast and Scalable Conflict Detection for Packet Classifiers, *10th IEEE International Conference on Network Protocols (ICNP'02)*,2002.

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

[4] F.Braun, J.Lockwood, M.Waldvogel, Reconfigurable Router Modules Using Network Protocol Wrappers, *Field-Programmable Logic and Applications* (FPL 2001), Lecture Notes in Computer Science 2147, pp. 254-263, August 2001.

[5] M.Berg, M.Kreveld, and J.Snoeyink, Two- and three-dimensional point location in rectangular subdivisions, *Journal of Algorithms*, 18(2):256-277, 1995.

[6] A.Broder, M.Mitzenmacher, Using Multiple Hash Functions to Improve IP Lookups, *IEEE Infocom 2001.*

[7] M.Buddhikot, S.Suri and M.Waldvogel, Space decomposition techniques for fast layer-4 switching, *Conference on High Speed Networks*, 1998.

[8] P.V.Emde Boas, R. Kass, and E.Zijlstra, Design and implementation of an efficient priority queue, *Mathematical Systems Theory*, 10, 99-127, 1977.

[9] D.Eppstein and S.Muthukrishnan, Internet packet filter management and rectangle geometry, *12th ACM-SIAM Symp. on Discrete Algorithms*, 2001, 827-835.

[10] A.Feldman and S.Muthukrishnan, Tradeoffs for packet classification, *INFOCOM*, 2000.

[11] P.Gupta and N.McKeown, Packet classification using hierarchical intelligent cuts, *ACM SIGCOMM*, 1999.

[12] P.Gupta, http://klamath.stanford.edu/∼pankaj/rfcv1.0.tar.gz

[13] P. Gupta and N. Mckeown, Algorithms for packet classification, *IEEE Network*, 15(2):24-32, 2001.

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

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

[16] Internet algorithms lab, http://www.ial.ucsd.edu/classification/.

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

[18] T.Lakshman and D.Stiliadis, High speed policy-based packet forwarding using efficient multi-dimensional range matching, *ACM SIGCOMM*, 1998.

[19] Merit, Ipma statistics, http://nic.merit.edu/ipma.

[20] L.Qiu, G.Varghese and S.Suri, Fast firewall implementation for software and hardware based routers. *9th International Conference on Network Protocolos ICNP*, 2001.

[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, 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.

[23] H.Samet, Application of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison-Wesley, 1989.

[24] V.Srinivasan, G.Varghese, S.Suri, M.Waldvogel, Fast and scalable layer 4 switching, *ACM Sigcomm'98.*

[25] V.Srinivasan, S.Suri, and G.Varghese, Packet classification using tuple space search, *ACM SIGCOMM*, 1999.

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

[27] V. Srinivasan, A packet classification and filter management system, *INFOCOM*, 2001.

[28] X.Sun and Y.Zhao, Packet classification using independent sets, *IEEE Symposium on Computers & Communications*, 2003, 83-90.

[29] Filter Set Generator. http://www.arl.wustl.edu/∼det3/

[30] M.Waldvogel, G.Varghese, J.Turner, and B.Plattner, Scalable High-Speed Prefix Matching, *ACM Transactions on Computer Systems*, Volume 19, Number 4, Pages 440-482, November 2001.

[31] D.E.Willard, Log-logarithmic worst-case range queries are possible in space $\theta(N)$, *Information Processing Letters*, 17, 81-84, 1983.