A Modular Approach to Packet Classification: Algorithms and Results

Thomas Y.C. Woo
Bell Laboratories, Lucent Technologies
woo@research.bell-labs.com

Abstract— **The ability to classify packets according to pre-defined rules is critical to providing many sophisticated value-added services, such as security, QoS, load balancing, traffic accounting, etc. Various approaches to packet classification have been studied in the literature with accompanying theoretical bounds. Practical studies with results applying to large number of filters (from 8K to 1 million) are rare.**

In this paper, we take a practical approach to the problem of packet classification. Specifically, we propose and study a novel approach to packet classification which combines heuristic tree search with the use of filter buckets. Besides high performance and reasonable storage requirement, our algorithm is unique in the sense that it can adapt to the input packet distribution by taking into account the relative filter usage.

To evaluate our algorithms, we have developed realistic models of large scale filter tables, and used them to drive extensive experimentation. The results demonstrate practicality of our algorithms for even up to 1 million filters.

I. Introduction

Multi-dimensional packet classification with a large number of filter rules is a provably hard problem [4], [7], [8]. Specifically, previous work has cast it in terms of the range matching problem in computational geometry [3], where there are various known algorithms and theoretical results.

Most of these studies, however, focus on worst-case performance, and does not take into account actual filter usage statistics, nor the types of commonly occurring filter patterns. Moreover, they provide sparse experimental results. In practice, the asymptotic complexity does not accurately tell how the algorithms scale to large number (e.g., from 8K to 1M) of filters.

In this paper, we take a more pragmatic approach. Our interest is not as much in analytical results, but in exploring the practical limits of packet classification and understanding performance through empirical experimentations. Our algorithm is motivated by intuitive observation on the classification process, and is based on an efficient divide-and-conquer approach. Specifically, we break up the classification procedure into two main steps. In the first step, our algorithm tries to eliminates as many filters as possible by examining specific bit positions. However, instead of eliminating all but one filter, the first step terminates when the set of remaining filters is less than some pre-specified maximum. We call this set of filters a *filter bucket*. This early termination avoids the explosion that is often the result of trying to completely differentiate between a few "similar" filters. In the second step, the filter bucket is processed to find a match. Because of the limited size of a filter bucket, a completely different procedure (e.g., (hardware-based) linear or associative search) can be used. In essence, our algorithm is a modular composition of two procedures: the first to decompose large filter table into small filter buckets of a fixed maximum size (from 8 to 128), and the second to process filter buckets of limited size to find a match.

Our algorithm is also unique in the sense that it can take into account the relative usage of the individual filters in a filter table to build a more optimal search data structure. This is especially important as usage of individual filters tends to be highly unbalanced.

Our algorithm is amenable to implementation in software, hardware, or a combination of the two. We examine some of the implementation issues in this paper.

In a nutshell, our contributions are: (1) We propose and study a novel heuristic approach to packet classification that provides good average case performance, uses reasonable storage, and can adapt to the usage of individual filters. (2) We examine different issues concerning the practical implementation of our approach. (3) We identify characteristics of realistic filter tables for different classes of router devices and develop a framework for modeling them. (4) We provide benchmark results on the practical performance of our proposed algorithms based on the filter table models of different router devices.

The balance of the paper is organized as follows. In Section II, we precisely define the packet classification problem. In Section III, we present the details of our algorithms. In Section IV, we examine implementation issues. In Section V, we present our experimental results. In Section VI, we compare our approach to related work. Finally, we conclude in Section VII.

II. The Packet Classification Problem

From an algorithmic perspective, the IP packet classification problem is simply a concrete instance of the abstract classification problem. In the following, we define the latter first and specialize it to IP in the next subsection.

A. Abstract Classification Problem

A *basic filter* f is an ordered pair (b, m) of binary strings of equal length. We call b the *pattern*, and m the *mask*. m indicates the *significant* bits in b for matching purpose. For example, the basic filter (1001, 1010) means that the first and third (counting from left to right) bits of "1001" are significant for matching purpose. Equivalently, a basic filter can be represented as a ternary string in the alphabet $\{0,1,*\}$. Specifically, all the insignificant bits in b are replaced by "*," the don't care bit. The example above can be denoted as "1*0*."

Three special cases of basic filters can be defined. A basic filter, or equivalently called a *mask-based* filter, $f = (b, m)$ is called (1) *exact* if m consists of all "1"s; (2) *wildcard* if m consists of all "0"s; and (3) *prefix* if m is made up of "1"s followed by "0"s. Clearly, both exact and wildcard basic filters are special cases of prefix basic filters; and any basic filter can be represented as a collection of prefix basic filters. For example, the

basic filter "*0**" is equivalent to the collection of prefix basic filters {"00**", "10**"}.

A binary string t *matches* a basic filter $f = (b, m)$ if t and b are of equal length and are identical in all significant bit positions as indicated by m. For example, "1100" matches the basic filter "1*0*."

A *k-dimensional filter* F is a k-tuple of basic filters.[1] A *k-dimensional filter table of size* N is an ordered sequence of N k-dimensional filters. We typically denote such a table FT by the sequence $F_1, F_2, \ldots, F_N$. The size of a filter table FT is denoted by $|FT|$, i.e., $|F_1, F_2, \ldots, F_N| = N$.

Let t be a k-tuple $(t_1, \ldots, t_k)$ of binary strings, and F a k-dimensional filter denoted by $(f_1, \ldots, f_k)$. We say t *matches* F if for all $1 \leq j \leq k$, t_j matches f_j. In this case, F is called a *matching* filter for t.

Given a k-dimensional filter table FT of size N denoted by $F_1, \ldots, F_N$, a procedure for *abstract classification* takes an arbitrary input k-tuple t and returns the first F_i such that t matches F_i or **NIL** if there is no match. We call F_i the *best matching filter* for t.

An equivalent formulation of the problem is to associate each filter with a distinct *cost* or *priority*. In which case, the classification procedure should return the matching filter with the least cost or highest priority.

A simple extension to the classification problem is to associate each filter F_i with a weight W_i. The weight represents the relative match frequency of a particular filter, and is typically derived from the distribution of the input tuple t or filter usage statistics. More precisely, let t be drawn from some fixed input distribution from which the W_i's are derived. Then

$$\frac{prob(F_i \text{ is the best matching filter for } t)}{prob(F_j \text{ is the best matching filter for } t)} \approx \frac{W_i}{W_j}$$

Knowledge of the weights may help in constructing more efficient classification procedures. We call this extended problem the *weighted abstract classification problem*. In the sequel, to avoid repeated definitions, the classification problem without weights is treated as the weighted classification problem where all W_i's are 1.

Filter Covering

Given a filter table FT, not all filters can potentially be matched. For example, consider the 1-dimensional filter table 1*, 00, 11, 01, 0*, both the filters "11" and "0*" will never be returned as a match as any input matching them would have matched earlier filters, "1*" for the former and "00" or "01" for the latter.

We can formalize this with a notion called *covering*. A set of filters $S = \{F_i\}$ is said to *cover* a filter F if for all input t, if t matches F, then t also matches some filter F_i in S. Given a filter table FT, a subsequence of filters $F_{i_1}, \ldots, F_{i_m}$ is said to *cover* F_ℓ if $\{F_{i_1}, \ldots, F_{i_m}\}$ covers F_ℓ and $i_m \leq \ell$.

Using the covering relation, we can divide a filter table FT into two sub-tables T and T' such that (1) filters in T and T' are subsequences of FT; (2) T and T' form a partition of FT, i.e., all filters in FT are in exactly one of T or T'; (3) $\forall F \in T', \exists F_{i_1}, \ldots, F_{i_m} \in T$ such that $F_{i_1}, \ldots, F_{i_m}$ cover F; (4) T' is a maximal subsequence satisfying (1)–(3). We can easily show that such a division produces a unique pair T and T'. We call the process of obtaining T and T' *reduction*, and denote (T, T') as the *reduct* of FT.

A filter table FT is said to be *reduced* if its reduct is $(FT, \emptyset)$.

[1] In other words, a basic filter is equivalent to a 1-dimensional filter.

B. IP Packet Classification

The IP packet classification problem can be stated as a specific instance of the abstract classification problem applied to the domain of classifying IP packets. The specific instantiation is defined as follows:

- The different dimensions of a filter correspond to the different fields of interest that can be extracted from an IP packet or its processing path.

Two forms are more popular: (1) 2-dimensional table with source and destination IP addresses; and (2) 5-dimensional table with source and destination IP addresses, protocol number, source and destination TCP/UDP port numbers.

- For IP packet filtering, a general form of filter called a *range* filter, where each dimension is specified as a range (s, f) ($s \leq f$ are integers), is sometimes used. For example, one can specify a range of port numbers to match using the range filter (6031, 8011).

A range filter is more general than a prefix filter. It is, however, not directly comparable to a mask-based filter. Specifically, some range filter (e.g., (9, 11)) can not be expressed as a single equivalent mask-based filters, and some mask-based filters (e.g., "*01*") can not be expressed as a single equivalent range filter. In general, any range basic filter can be represented by a collection (from 1 to $f - s + 1$) of mask-based filters.

Our proposed algorithm can potentially handle both mask-based and range filters because of its modular nature. Specifically, the tree search phase operates on mask-based filters, while the filter bucket search phase can process any type of filters.

- The weighted IP packet classification is similarly derived from the corresponding weighted abstract classification problem. In this case, the weights are derived from the usage counters associated with each filter; and for performance evaluation purpose, the incoming packets are assumed to be distributed in a way consistent with the weights.

The ability to adapt search to incoming traffic is especially important for IP packet classification as filter usage tends to be highly unbalanced. This distinguishes our approach from most existing approaches that can not easily take into account the relative usage of individual filters.

C. Solution Requirements

In comparing solution approaches, we first fix the complexity of filters and the number of filter rules. In this paper, we focus mostly on 2 and 5-dimensional prefix-based filters and filter table size of up to 1 million entries. Different solution approaches can then be differentiated along the following criteria.

Speed of Classification. There are at least 3 measures for the speed of classification: (1) worst case: the worst case search time possible for a packet; (2) average case: the average case search time possible for completely random collection of packets; and (3) statistical: the average case search time for packets drawn from some a priori specified packet or filter usage distribution. In this paper, we measure statistical search speed by first

assigning weights to filters,[2], then we generate random packet traffic consistent with the weight distribution, and measure the average search speed.

Amount of Storage. The amount of memory space occupied by the search data structure is an important consideration. There is a clear tradeoff between search time and search space.

Large memory space not only means extra cost for memory, but it may also force the use of slower memory (e.g., DRAM) in place of faster memory (e.g., SRAM).

Ease of Update. There are 3 possible updates: (1) full update: this refers to the initial construction of the search data structure from the filter table, or any re-construction thereafter from scratch. (2) incremental update: this refers to the incremental addition or deletion of filters from a search data structure. (3) reorganization/rebalancing: as filters are added and/or deleted over time, the search data structure may lose its efficiency. Certain packet classification approaches may include a procedure to reorganize the search data structure so as to regain its operating efficiency.

Since we do not describe the incremental update procedures in this paper due to space limitation, we do not consider this further.

III. Algorithms

Our approach consists of 4 algorithms: initial construction, (or equivalently, full update), incremental insert, incremental delete, and search. The first 3 are for construction and maintenance of the search data structure, and the last one for performing the actual classification. Due to space limitation, we will not discuss incremental update procedures in this paper.

To motivate our approach, we first make a few key observations:

1. For efficient search, a search path should seek to eliminate as many filters as possible from further consideration in the smallest number of steps. This, however, requires global optimization and can be extremely computationally intensive, due to the amount of look ahead. As an alternative, carefully designed local optimization techniques can be used to obtain reasonable search paths.
2. Prolonged search time and/or storage explosion are often the result of trying to separate "similar" filters. For example, separating the filters "0110" and "****" requires examining all 4 bits. Therefore, to avoid explosion, "similar" filters may be separated using a different technique.
3. The suitability of a search algorithm is highly dependent on the total number of filters. For large number of filters (e.g., 8K–1M), a decompositional technique with a multiplicative decrease factor can potentially yield an exponentially smaller set of filters in a linear number of steps. For small number of filters (e.g., ≤ 128), simplistic search procedures (e.g., a (pipelined) linear search) can perform as well as more sophisticated schemes. This suggests that as search progresses, i.e., as the number of remaining filters decreases, a change of the search approach may be desirable.
4. Filter usage statistics can provide useful hint on constructing efficient search data structures. Most filter implementations do keep a usage counter for each filter for statistics collection purpose. An approach that can make use of such statistics is desirable. We call such approaches *adaptive*, as they can adapt to input traffic characteristics.
5. There is a clear search speed vs storage tradeoff in most packet classification approaches. A good approach should allow flexible and tunable control between search speed and storage. Specifically, a user should have an explicit means to decrease the storage requirement if she is willing to accept a higher average search time, or vice versa.

[2]In real life, this comes from the filter usage statistics, or counters associated with each filter.

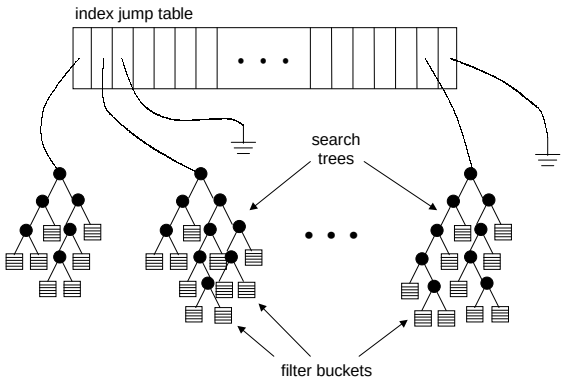


Fig. 1. Search Data Structure

Our approach addresses each of the above observations. At a very high-level, our approach organize the search space into 3 layers (Figure 1):

- Index jump table — The filters are statically divided into different groups using some initial prefixes of selected dimensions.
- Search tree — The filters in each group are then organized in a 2^m-ary search tree. The search tree is constructed by examining m bits of the filters at a time, and dividing them into 2^m groups. The particular m bits chosen for examination in each step can be drawn from any m arbitrary unexamined bit positions from any of the dimensions, and the choice is made to minimize duplication and maximize "balancedness" of the 2^m children. Many different criteria can be defined for the division. Ours takes into account the filter usage statistics, thus allowing it to adapt to the distribution of input traffic.

The division process terminates when the number of filters in a node is less than some pre-defined maximum.

- Filter bucket — The set of filters left at the leaf nodes when the division process terminates is called a *filter bucket*. Essentially, a filter bucket contains a set of filters that we do not wish to further distinguish using the tree. Typically, a different algorithm is applied to search the filter bucket for a match. In other words, the filter bucket demarcates the point where the search approach switches from one to another.

A filter bucket contains at most a pre-defined maximum number (typically small from 8 to 128) of filters.

Given the search data structure, the search procedure is straightforward. A packet is first directed to a specific subtree by indexing via the jump table using the initial prefixes of certain selected dimensions. Then, it goes through a "sifting" process to place it further and further down the tree by inspecting m of its bits each step, until it lands in a filter bucket. A bucket search procedure is then invoked to match the incoming packet against the filters in the bucket.

The tree phase is optimized to allow the search to quickly nar-

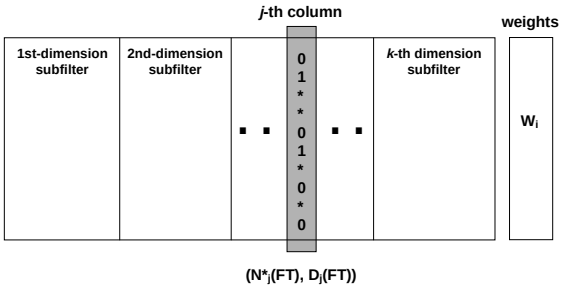


Fig. 2. Concatenated View of IP Packet Filter Table

row down to a single filter bucket among a large set of filters (up to 1 million filters in our experiments). The bucket phase is optimized to quickly search through a small set of filter (up to 128 filters in our experiments) in a filter bucket to find a match.

Strictly speaking, our approach represents a class of algorithms, rather than a specific algorithm. Specifically, by varying the criteria for selecting the m bits, m itself, and the amount of lookahead in determining the best m bits to use, one can obtain different instantiations of the algorithm. In this paper, we study the case for a specific bit selection criteria (to be described in Section III-B), $m = 1$, and a single step lookahead. The extension to the general case is straightforward, as the key ideas remain the same.

In the following, for ease of explanation, the procedures shown are not optimized. An actual implementation tries to reuse as much computation as possible at each step (see Section IV).

Notations. Before we present the algorithms, some definitions are in order. Let B be a 2-dimensional (n rows by m columns) array of ternary digits 0, 1, and *. That is each $B[i, j]$ is either 0, 1, or *. We denote the i-th row by $B[i, \cdot]$, and the j-th column by $B[\cdot, j]$. In addition, we denote by $B^{-x..y}$ the resulting n by $m-(y-x+1)$ array obtained by removing columns x through y from B. We abbreviate $B^{-x..x}$ by B^{-x}. Lastly, each row i of B has an associated weight denoted by W_i.

For each column j ($1 \leq j \leq m$), we define 3 quantities $N0_j(B)$, $N1_j(B)$ and $N*_j(B)$ as follows:

$$Nx_j(B) = \sum_{1 \leq i \leq n, B[i,j]=x} W_i$$

where x could be 0, 1, or *. In words, $Nx_j(B)$ is the total weights of all the rows whose j-th column is x. Furthermore, we define

$$D_j(B) = |N0_j(B) - N1_j(B)|$$

which gives the difference between the total weights of all the rows whose j-th column is 0 and those whose j-th column is 1.

Let $FT = F_1, \ldots, F_N$ be a k-dimensional IP packet filter table. By concatenating all the dimensions together, it can be viewed as a 2-dimensional array of ternary digits. In particular, each F_i is a fixed-length ternary digit string. Using the above definitions, we associate for each column j of this array an ordered pair $(N*_j(FT), D_j(FT))$. This is summarized in Figure 2.

We also denote by $D_{min}(FT)$ and $D_{max}(FT)$ respectively the smallest and the largest values of $D_j(FT)$ among all columns of FT. $N*_{min}(FT)$ and $N*_{max}(FT)$ are defined in a similar fashion.

```
*Table function BuildTable (FilterTable FT, int h_1, ..., int h_k)
{
      maxEntries = ∏_{h_j ≠ 0} 2^{h_j};
      T = new FilterTable (maxEntries);
      foreach Filter F_i ∈ FT {
        for j from 1 to FT.dimension {
          let p_j be the prefix in the j-th dimension of F_i;
          d_j = (h_j > numberOfBits (p_j)) ? h_j - numberOfBits (p_j) : 0;
(1.1)     s_j = first (h_j − d_j) bits of p_j ⊕ d_j bits of "0";
(1.2)     f_j = first (h_j − d_j) bits of p_j ⊕ d_j bits of "1";
        }
        foreach x_1 ∈ s_1..f_1, x_2 ∈ s_2..f_2, ..., x_k ∈ s_k..f_k
(1.3)     add F_i^{-1..h_j} to T[x_1 ⊕ x_2 ⊕ ... ⊕ x_k].filters;
      }
      return T;
}
*Node function BuildTree (FilterTable FT)
{
      n = new Node ();
(2.1) let (T, T') be the reduct of FT;
      n.reduce = (T, T');
      FT = T;
      if (|FT| ≤ BUCKETDEPTH) {
(2.2)   n.filters = FT;
        return n;
      }
      for j from 1 to FT.dimension
(2.3)   preference[j] = ComputePreference (FT[·, j]);
(2.4) b = least j such that for all x: preference[j] ≥ preference[x];
(2.5) FT_0 = sub-sequence of all filters in FT whose b-th bit is "0" or "*";
(2.6) FT_1 = sub-sequence of all filters in FT whose b-th bit is "1" or "*";
      n.bit = b;
      n.filters = NULL;
(2.7) n.left = BuildTree (FT_0^{-b}); (n.left).parent = n;
(2.8) n.right = BuildTree (FT_1^{-b}); (n.right).parent = n;
      return n;
}
*Table function BuildSearchStructure (FilterTable FT, int h_1, ..., int h_k)
{
(1)   T = BuildTable (FT, h_1, ..., h_k);
      for x from 1 to |T| {
(2)     T[x].tree = BuildTree (T[x].filters);
        (T[x].tree).parent = T[x].tree;
      }
      return T;
}
```

Fig. 3. Initial Construction

A. Filter Bucket

The basic building block in our approach is a filter bucket. A filter bucket has the following properties: (1) It contains a small pre-defined maximum number of filters; typical bucket sizes are 8, 16, 32, 64, and 128. (2) The filters in a bucket are "similar" in some way. Specifically, there is a set of bit positions such that all filters in the bucket are "similar." [3] (3) A filter may appear in multiple filter buckets. For example, a range filter typically appears in multiple filter buckets.

Because of the small number of filters, many techniques can be used to efficiently search a filter bucket. We describe a few here:

- Linear Search — Though linear search may appear slow in software, it is a decent choice for hardware implementation. By searching each of the filters in a pipelined fashion, the throughput of a M-filter linear search equals that of a 1-filter search. The matching of each dimension in a filter can proceed in parallel by using multiple comparators. In other words, an M-stage pipeline implementation (with appropriate structuring of the filters into disjoint memory banks) can search a filter bucket of depth M in the time of a single comparison.

[3] "*" is considered to be similar to both "0" and "1," while "0" and "1" are not "similar."

• Binary Search — We can represent each dimension of a filter by an interval. A packet can be matched by first applying a binary search on all the end points in each dimension, and then combining the results from all dimensions [8].
• Hardware CAM — By using a content address memory (CAM) to store each dimension (prepended with the bucket ID) of a filter in a filter bucket, each dimension can be searched in parallel and then combined in a parallel step to obtain a match. Hardware CAM is most useful for applications where filter updates are an order of magnitude less frequent than packet forwarding rates, to avoid frequent reloading of CAM entries.

For the rest of this paper (and in particular the experimental results), we assume the use of linear search as the search procedure for filter buckets.

B. Initial Construction

For initial construction, we assume we are given a k-dimensional IP packet filter table FT, and for each dimension, $1 \leq j \leq k$, the number of bits h_j to be used in the index jump table construction.

The construction consists of two key steps: steps (1) and (2) of `BuildSearchStructure()` in Figure 3. In step (1) (`BuildTable()`), the set of filters is broadly divided into a collection of smaller filter sets by examining the first h_j bits of dimension j (steps (1.1)–(1.2)). A filter is duplicated into multiple such filter sets if the prefix length of at least one of its dimension j is less than h_j (step (1.3), $\oplus$ denotes the binary string concatenation operator).

Typically, the h_j's are chosen such that it is at most the minimal prefix length of the j-th dimension among all filters. The motivation is that the set of filters sharing the same prefixes in multiple dimensions is hopefully smaller. Both indexing or hashing can be used to map prefixes into search trees.

In step (2) (`BuildTree()`), individual subtrees are constructed for each smaller filter sets created by `BuildTable()`. Each filter set is divided recursively (steps (2.7)–(2.8)) until it can fit into a filter bucket (step (2.2)). Each tree node in a subtree logically corresponds to a set of filters that is still under consideration. Each child of a tree node contains a subset of the filters in the parent's node, and each leaf node contains a filter bucket.

The basic idea of the division is as follows: Given a particular bit position b,[4] a set of filters can be divided into 2 groups: the "0"-group containing all the filters whose b-th bit is "0" or "*," (step (2.5)) and the "1"-group containing all the filters whose b-th bit is "1" or "*." (step 2.6)). The rationale is that if the b-th bit of an input packet is "0," then it can only match the filters in the "0"-group and thus only those need to be considered further, and vice versa for the "1"-group. Thus, the key is to choose "good" bit positions so that only a small number of division is needed to reach a leaf node. We describe our bit selection scheme in the next subsection.

The reduction in step (2.1) can be critical. By "collapsing" filters in intermediate nodes, the number of nodes generated can be significantly reduced. To reduce the complexity of reduction, an incomplete but less expensive form that removes only duplicates can be used. The use of incomplete reduction can increase the size of the resulting tree.

[4]We focus only on single bit branching in this paper; our bit selection criteria extends in a straightforward manner to the multibit case.

```
int function SearchTree (Packet p, *Node T)
{
        if (T = NULL) return NIL;
        if (T.filters) /* leaf node */
(1.1)       return BucketSearch (p, T.filters);
        else /* internal node */
            if (p[T.bit] = 0) /* go left */
(1.2)           return SearchTree (p, T.left);
            else /* go right */
(1.3)           return SearchTree (p, T.right);
}
int function Search (Packet p, *Table T)
{
        let x_j be the first h_j bits in the j-th dimension of p;
(1)     return SearchTree (p, T[x_1 ⊕ ... ⊕ x_k]);
}
```

Fig. 4. Search Procedure

C. Bit Selection

The bit selected at each node determines the overall "shape" of the tree. Thus, given some global measure of the "goodness" of a search tree, the bit selected at each node should ideally "grow" the tree toward some final optimal shape. In abstract terms, we assign a preference value for each unprocessed bit position (step (2.3)), and we pick the bit with the highest preference position (step (2.4)).

For a search tree, a typical "goodness" measure is the weighted average search path length which is defined in our case as

$$wa(T) = \frac{\sum_i (\text{depth of filter bucket } i \cdot \sum_{F_j \in \text{filter bucket } i} W_j)}{\text{total number of filter buckets}}$$

This measure, though concrete and optimal, is computationally expensive to calculate, as it involves comparing fully constructed trees.

As a compromise, we try to optimize local measures in a hope that they cumulatively produce a reasonably "good" global solution. The "localness" of a measure is defined by the amount of lookaheads it uses. In this paper, we present results only for the case where a single bit is chosen at each node and our preference value is based only on 1 level of lookahead.

The preference metric we study in this paper uses a combination of 2 values: $N*_j$ and D_j. The former provides a measure of *progress*. Specifically, branching based on bit j will not eliminate more filters from consideration for the amount of traffic proportional to $N*_j$. Thus to maximize progress, the value $N*_j$ should be minimized. The latter provides a measure of *balancedness*. Specifically, a smaller value of D_j means more even branching of traffic into the next level.

In the experiments we report in this paper, we assign the preference value of column j for a filter table FT as

$$\textit{preference}[j] = \frac{D_j(FT) - D_{min}(FT)}{D_{max}(FT) - D_{min}(FT)} + \frac{N*_j(FT) - N*_{min}(FT)}{N*_{max}(FT) - N*_{min}(FT)}$$

Our construction approach is a "greedy" one in that it tries to optimize only locally. The final tree it constructs can be "skewed" by the distribution of the bits in the filter set, and may be far from optimal. However, as we will present in Section V, the results for even very large number of realistic filters (up to 1 million) are good. In addition, unlike most existing proposals, it can adapt to the actual usage of the filters.

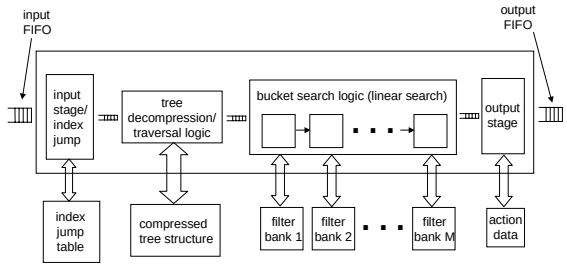


Fig. 5. An Example Packet Classification Pipeline Implementation

D. Search

The search procedure is straightforward. Its code as shown in Figure 4 is self-explanatory. First, it concatenates the leading h_j bits from each dimension j of the incoming packet to construct an index into the jump table to retrieve the root of a search tree. Then, it traverses the search tree by branching according to the value of the bit position stored in the current node, until it reaches a leaf node. Finally, the filter bucket is searched to locate a possible match.

Each phase of search, namely, index jump, bit branching, and bucket search, are simple and are amenable to highly efficient implementation in software or hardware.

IV. Implementation Considerations

In a typical implementation, initial construction and update are software procedures that run on a standard CPU. They build and maintain the search data structure in memory, which is in turn accessed by the search procedure.[5] The search itself can be implemented either as customized hardware (e.g., FPGAs or ASICs) or as part of the data path software, depending on the particular design approach.

Mutual exclusion is critical during update of the search data structure. This can be achieved by double buffering and some form of atomic switch.

A. Initial Construction

If weights are not available, the initial construction should be run with all weights set to unity. Then every so often, the construction can be re-run using actual usage statistics. The re-run can also be triggered by some measure of "balancedness," the number of updates, etc.

The two time consuming steps in the initial construction are the reduction and the preference computation. The former can be performed in $O(N \log N)$ time where N is the number of filters to be reduced. The computation can be reused from the parent to the children.

Preference computation can be sped up by bounding the number of columns to be examined. This is straightforward for non-weighted case, as $N*_j()$ is an increasing function of the column number j within a single dimension.

Fortunately, even though the number of nodes expands at each layer, the tree construction gets more efficient, as both the number of filters and the number of columns decrease at each layer. The former could decrease geometrically while the latter linearly.

[5] After appropriate downloading of the structure into the memory of the search engine.

B. Search

There are 3 keys to optimizing search performance: (1) reduce the complexity of the basic search step; (2) reduce the number of memory accesses; and (3) consider pipeline implementation. We describe each below.

Basic Search Step. Our approach has 3 kinds of basic search steps, namely, indexing to select the correct subtree to search, tree traversal based on a particular bit position, and the matching of a packet to a single filter. The first two are extremely primitive and map directly to hardware instructions even with a software implementation. The third requires a number of comparison proportional to the number of dimensions. In hardware though, parallel comparators can be used to perform the match in a single step.

Memory Organization. Careful memory organization is critical to improving search performance. To reduce data access time: (1) Data that are needed in the immediate future should be stored close together (e.g., in the same memory page) so that they are available without further fetching. For example, a child node should be stored close to its parent, as that it is available as soon as the branching decision is made. We have developed a novel scheme for compressing and storing tree nodes such that the nodes lying on a frequently visited tree path are stored closely together in a memory page. We describe our scheme below under Tree Compression in Section IV-C. (2) Multiple separate memory banks should be used for pipelining (see below).

Pipelining. By dividing the search steps into different stages, and pipelining through the stages, throughput (or the number of classifications per second) can be significantly improved.[6]

An example hardware pipelined implementation of our approach is shown in Figure 5. There are 4 stages in the pipeline: (1) input/index jump, which retrieves a packet header from the input FIFO, and look up the starting address of the appropriate search tree from the index jump table; (2) tree traversal, which reads in tree nodes by pages and makes branching decision; (3) bucket search, which has its own internal pipeline. Each stage of this M-stage internal pipeline handles **BUCKETDEPTH**/M filters. For each stage to operate in parallel, each has its own memory bank for storing the selected section of the filters; (4) output stage, which retrieves the action data corresponding to the match.

Using a novel way to partition the search tree into disjoint sections, it is possible to further pipeline the tree traversal stage. We omit the details here due to space limitation. The tree traversal stage can have variable completion time, internal FIFOs are used to absorb the variation.

C. Storage

The bucket size provides an effective control for the amount of storage used. A bucket size of 1 means the search tree must distinguish every single filters from one another, thus a large tree is needed. A bucket size equal to the size of the filter table requires the minimal storage, and is equivalent to a linear search.

[6] Though latency may increase slightly because of the transitions between stages.

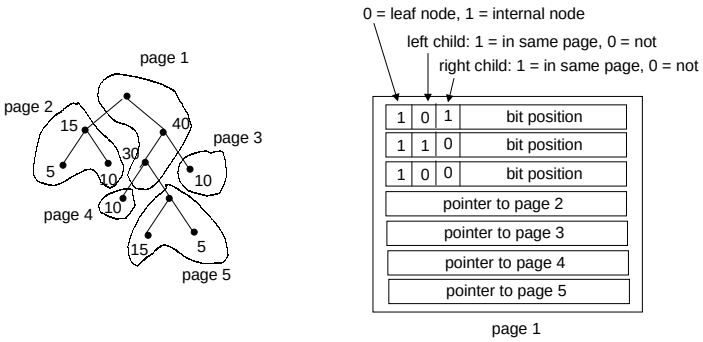


Fig. 6. Mapping of Search Tree to Memory Pages

Our algorithm does not provide a non-trivial worst-case storage bound as it depends on the distribution of the "0", "1", and "*" in the given filter table. It is possible to construct a highly skewed, but unrealistic, filter table that will require a large number of nodes.

Tree Compression. The search tree can be stored in a highly compressed form with a pointerless representation [9]. The compression follows 2 steps: (1) The tree is first segmented into subtrees of a maximum size (number of nodes), say X, where X is chosen based on the size of a memory page. We use a Huffman-encoding [1] like procedure for the segmentation. Specifically, we label each node with a weight: a leaf node's weight is the sum of the weights of all filters in its bucket, an internal node's weight is the sum of weights of its children. Then we start from the root collecting nodes into a page by selecting the node with the largest weight from all the nodes adjacent to some nodes already in the page.[7] This process continues until the page is full. Then a new page collection is started with one of the adjacent nodes. Figure 6 shows an example. (2) Each internal tree node can be encoded using a 3-bit type together with $\log W$ bits (W is total number of bits in all dimensions) of node information (i.e., the bit position to be examined). For source and destination IP address filters, each internal node can be stored in 9 bits. Each leaf node need a 1-bit type together with a bucket ID. As we will see in Section V, a 24-bit bucket ID is far more than enough.

An example encoding of page 1 is shown in Figure 6. We note that for a page with X nodes, there are at the most $X+1$ external pointers.

Using our tree compression, a search tree with even up to 1 million nodes take up at most a few Mbytes of memory, which is well within acceptable limits in a modern high-end router.

Filter Bucket Compression. Filters common to many filter buckets need not be stored multiple times. For example, a filter cache for the M most frequently occurring filters can be kept in on-chip memory. The cached filters can be represented by a cache index in the filter buckets they appear.

Wildcard Separation. Wildcard filters are the main contributors toward storage explosion. If a dimension contains a large majority of wildcard filters, it may be better off to separate them out in another table and construct two different search trees that must both be searched to find a match.

As a concrete example, consider a filter table with source and destination IP addresses as the 2 dimensions. Instead of constructing a single search structure for all the filters, one can partition the table up into 4 subtables: Table 1: all filters in which both the source and destination addresses are not wildcards; Table 2: all filters in which the source address is a wildcard; Table 3: all filters in which the destination address is a wildcard; Table 4: all filters in which both the source and destination addresses are wildcards, which contains at most 1 filter. Corresponding to these 4 subtables, 4 search trees can be built and searched to find a match.

[7] This is in a way similar to Dijkstra's algorithm for computing the shortest path.

V. Experimental Results

Evaluating a heuristic algorithm is tricky, as there are always pathological cases that do not perform well. Evaluating a heuristic approach for packet classification is even worse, as there does not yet exist real filter table with large number of filter rules.[8]

Thus, in this paper, we resort to evaluating our algorithms by modeling what a large filter table would look like. Our model is based on practical observations on existing filter table, and projecting on future applications for packet classification. Interestingly, the modeling of packet filter table can be considered a research topic in its own right, as there are many potential applications for packet classification with diverse requirements. A commonly agreed model can serve as a benchmark for the many packet classification algorithms that have been proposed. Because applications of packet classification are not yet fully developed, our model should best be understood as a first attempt in capturing the potential complexity of a filter table.

In a nutshell, in our modeling approach, network elements are classified into distinct types, some examples are workstation hosts, server hosts, subnet border routers, enterprise core routers, enterprise edge routers, ISP edge routers, ISP core routers, and ISP peering routers. For each class, the applications and characteristics (e.g., distribution of values in each dimension, filter specificity, etc.) of their filter tables are identified. A summary of their key characteristics is presented in Figure 7. These characteristics in turn are "codified" in filter specification files of our design. Specific filter table instances can then be obtained by running a filter generation tool that we have built on the filter specification file.

Our particular modeling approach will certainly be scrutinized. It does, however, serve a purpose in establishing the feasibility and baseline performance of our proposed algorithm.

The results we report below are for the case of an ISP edge router, which, in our opinion, would need to support the largest number of packet filters. Briefly, our ISP edge router filter specification consists of 4 sections: (1) VPN filters that has fully specified source and destination addresses, and port numbers; (2) ingress filters that apply to sources in a single subnet; (3) ingress filters that apply to sources in multiple subnets; and (4) ingress filters that apply to souces in an entire domain.

Our results are obtained by randomly generating a large number of filter table instances of varying sizes using our ISP edge router filter specification. For each experiment, we collect the statistics (e.g., number of filter buckets (leaves), tree depth) of the tree structure by explicitly constructing it, and the search performance by running a large number of randomly generated (obeying the weights) packets through it.

[8] Most existing ones consist of at most hundreds of rules, and they mostly have a bias toward firewall applications.

Type	# of Filters	Use	Address Characteristics
Workstation Host	8K	QoS (by destination IP, application)	fixed source IP to any, most traffic goes to a small number of fixed destination hosts, the rest is uniformly distributed over a large number of random destinations, about even ingress/egress traffic
Server Host	8K-64K	QoS (by source IP)	from any to fixed destination IP, highly unbalanced ingress/egress traffic ratio, about equal traffic goes to a large number of destination hosts with a fixed domain prefix (Intranet server) or random (Internet server)
Subnet Border Router	8K-128K	QoS (by L2 label, subnet), security (by subnet)	from fixed subnet prefix to any, or from fixed subnet prefix to fixed subnet prefix
Enterprise Core Router	8K-64K	QoS (by physical port, L2 label, application, TOS)	from fixed subnet prefix to fixed subnet prefix (Intranet), or from fixed subnet prefix to any (Internet)
Enterprise Edge Router	64K-256K	QoS (by destination IP, application), security (by source IP), tunneling (by source and destination IP)	from fixed domain prefix to fixed destination prefix (VPN), or from fixed domain prefix to any (Internet), segmented by subnet
ISP Edge Router	512K-1M	QoS (by source IP, application, TOS), consistency (by source IP), tunneling (by source and destination IP)	from large set of domain prefixes to another large set of domain prefixes (VPN), or from large set of domain prefixes to any (Internet), segmented by customers' logical interfaces
ISP Core Router	8K-128K	QoS (by application, TOS, MPLS label)	transit traffic vs traffic that originates and terminates in the ISP
ISP Peering Router	8K-128K	peering agreement enforcement (by physical port, source IP), consistency (by physical port, source IP)	transit traffic vs traffic that originates and terminates in the ISP

Fig. 7. Filter Characteristics by Location

We have focused most of our experiments on 2 cases: (1) 5 dimensions including source and destination IP addresses, protocol number, and source and destination port numbers; and (2) 2 dimensions with only source and destination IP addresses. The trends in both cases are similar. For brevity, we present only the results for the 5 dimension case.

All results are obtained from pure application-level software implementations of the algorithms in C on a standard desktop Pentium II 400MHz with 512K L2 cache. The implementation is not optimized (e.g., we use 1 byte to represent each filter bit), especially in regard to the use of linear search for filter buckets.

The results are summarized in Figures 8–10. Figure 8 shows the general trend as filter table size increases. Figure 9 shows the effect of filter bucket size on the storage and search performance. Figure 10 shows the benefits of the first level index jump table. The notation AxB refers to the use of A and B bits respectively from the first 2 dimensions (source and destination IP addresses) to form the jump table. We elaborate on the results below.

We do caution that our results here represent average case performance using our filter table models. It is possible for one to construct artificial examples whose results deviate significantly from our average case. Though, as our results show, the filter bucket size and the index jump table can serve as effective tunable controls to temper potential "bad" filter tables.

A. Tree Statistics

From Figure 8(a), we see that the number of filter buckets grows linearly with the filter table size, while the depth grows about logarithmically with the filter table size. Even for 1 million filters, less than 200,000 filter buckets are used.

As expected, the weighted case has a larger tree (number of filter buckets and depth) than the non-weighted case in general. The percentage increase is not significant though, and is below 10% in all cases.

From Figure 9(a), we see that the filter bucket size provides an effective control for storage requirements. A doubling of the filter bucket size about halves the number of filter buckets.

From Figure 10(a), we see that the use of jump table decreases tree depth proportionally, though it does increase the total number of filter buckets. This apparent contradiction is explained by the fact that certain filters (e.g., a wildcard filters) are duplicated into multiple subtrees. For example, a wildcard filter is duplicated into $2^{(6+6)} = 4096$ search subtrees with a 6x6 jump table.

B. Search Performance

From Figure 8(b), we see that the search performance decreases sublinearly with increase in filter table size. The performance range is from about 261,000 classifications for filter table size of 8K to 113,000 classifications for filter table size of 1M. Since these results are from an application-layer program, we expect many-fold increase in performance in an embedded software implementation running on a dedicated CPU, and even better performance with a customized hardware implementation.[9]

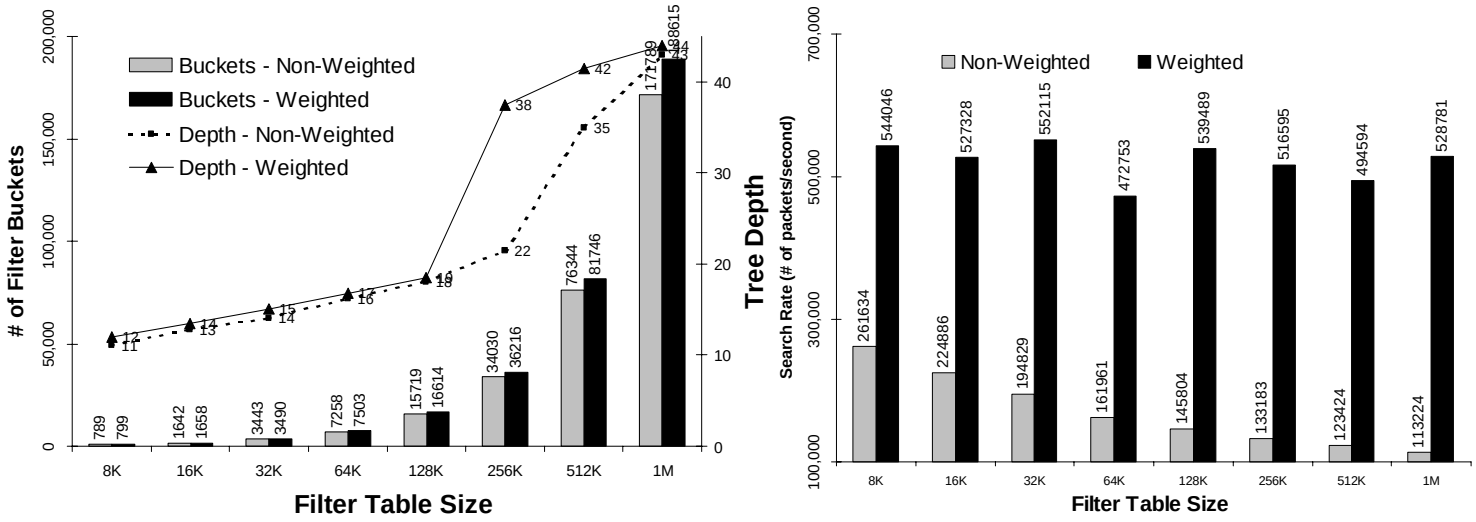
We note that the specific search time we obtained is highly skewed by the time spent in the linear search phase. Specifically, the tree traversal time is directly determined by the tree size, it decreases proportionally as the tree reduces in size.

We also observe that the weighted case performs significantly better than the non-weighted case. In fact, the performance of the weighted case ranges from 2 to 4 times better than their non-weighted counterpart, and is relatively insensitive to the filter table size. This provides evidence to the importance of collecting filter usage statistics and the potential benefits for a packet classification algorithm to exploit them.

From Figure 9(b), as expected with a linear search of filter buckets, we see that as the filter bucket sizes increases, the search rate decreases. The decrease is, fortunately, sub-linear. A double of filter bucket size does not come close to halving the search rate.

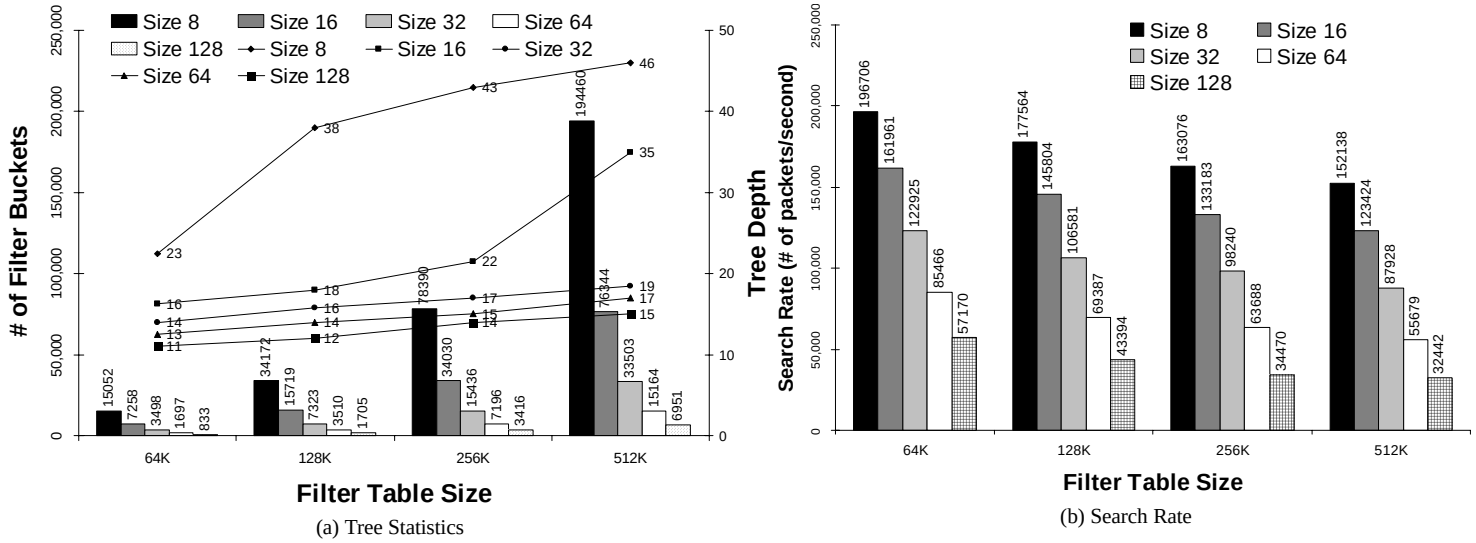
From Figure 10(b), we see the benefits of using a jump table. The search rate is higher with the use of jump table. In particular, the use of an 6x6 jump table can provide a speedup of more than 15% over the case where no jump table is used.

[9] A hardware pipeline implementation using parallel comparators can remove the key bottlenecks: bit examination and the linear filter bucket search.



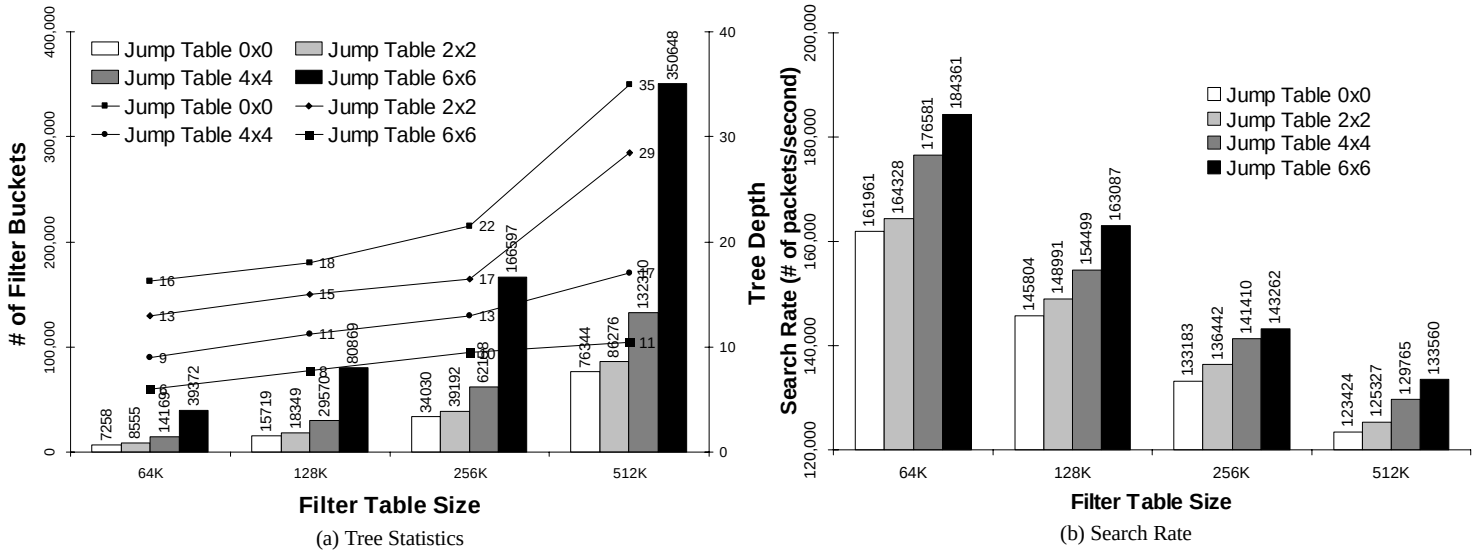
(a) Tree Statistics (b) Search Rate

Fig. 8. Trends under various Filter Table Sizes (no jump table, filter bucket size = 16)



(a) Tree Statistics (b) Search Rate

Fig. 9. Trends under various Filter Bucket Sizes (no jump table)



(a) Tree Statistics (b) Search Rate

Fig. 10. Trends under various Filter Table Sizes with Jump Table (filter bucket size = 16)

VI. Related Work

Existing approaches to packet classification can be grouped into 2 broad categories: geometry-based and non-geometry-based. The former refers to algorithms that interpret filters as geometric shapes, and map the packet classification problem to some form of geometric point-location problem. The latter refers to any other approach based on regular data structures such as trees and graphs. [8] and [2] belongs to the former, while [7],

[4], [6], [5] and ours belong to the latter.

[8] presents 2 algorithms. The first algorithm admits a hardware implementation but does not readily scale to a large number of filters. The second algorithm applies only to 2 dimensions, and it does not appear to easily generalize to higher dimensions.

[2] is based on space decomposition, it has worst-case search time similar to the best existing schemes, but improves on the worst-case update time.

[7] also presents 2 algorithms, namely, *Grid of Tries* and *Crossproducting*. The construction of the former appears to be complicated, and updates can not be easily performed. A cascade update of *switching pointers* may be triggered by a single update. Crossproducting, as the authors admit, can suffer from memory blowup. They introduce an on-demand scheme, but did not provide extensive results. It would be interesting to compare the average performance of crossproducting with our approach.

[4] uses an approach based on a *directed acyclic graph* (DAG). The approach is simple, but it requires $O(N^2)$ storage. Though both our scheme and the DAG scheme use bit positions to construct a search tree, the details are significantly different. The two key components of our algorithm, heuristic bit selection and the concept of a filter bucket, are unique to ours. Since no formal description was given for the DAG algorithm[10], it is not clear how the algorithm scales to large number of filters. Again, the average case comparison with ours should be interesting.

Both [6] and [5] represent new interesting approaches to packet classification. Recognizing the inherent difficulty of the problem, both try to exploit structure within a filter table to improve search performance. Specifically, [6] makes use of the observation that there is typically only a small number of prefix lengths in a filter table, while [5] exploits the observation that "overlaps" of filter rules occur much rarer than suggested by the worst case. Similar to ours, [5] is heuristic-based, while [6] proposes the use of a heuristic to improve its searches.

Overall, most existing studies focus more on the worst-case bounds. Our emphasis is on practical approach with good average case performance and tunable controls to deal with "bad" cases.

VII. Conclusion

We proposed and studied a new approach to packet classification. Our approach combines two search procedures: a heuristic tree search for separating a large set of filters into fixed size filter buckets, and another search procedure for searching through fixed size filter buckets (we use linear search in our study). This modular construction is motivated by the practical observation that a single search approach may not be optimal for filter table of all sizes.

Through experiments with a large number of filter tables, we demonstrated that our approach yielded good search speed (around 200K classifications per second for 128K filter table using a pure software implementation) and reasonable storage. Though we caution that the experimental results do not constitute a complete validation of our approach, we do believe they demonstrated its general feasibility. As in all heuristic approach though, cases can be constructed where our approach do not work well. In those case, the filter bucket depth can be used as an effective tunable control to reduce memory usage at the expense of increased search time. We do not claim optimality nor universal applicability of our approach, but the key ideas in our approach represent novel tools for tackling the problem of packet classification. We would have liked to try our algorithms on real filter tables, this have proven to be very difficult. For privacy reasons, real filter tables are not easy to obtain. For the few ones that we do have, they are too small (hundreds of rules) to create any stress on our algorithm.

Our approach is also the only one we know of that can adapt to the input traffic distribution (i.e., relative filter usage). As the results demonstrate, taking into account the relative usage of each filter can dramatically improve search performance, by as much as 400% in some of our results.

Unlike most existing studies, our emphasis is on the average case performance and practical results. To this end, we have identified the different characteristics of filter tables for different classes of router devices, and proposed and implemented a framework to model filter tables. We feel that this is an important research area in its own right. Our effort should be considered a first step in this (right) direction, it is necessarily controversial as different people tend to have different projections on how large-scale packet classification will be applied. Better models can be defined as more real-life applications of packet classification are proposed.

For ongoing work, we are exploring variations of our basic approach. Specifically, we are looking at different bit selection criteria and multibit tree construction.

[10] It was introduced in the paper using an example.

References

[1] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, 1990.

[2] M. Buddhikot, S. Suri, and M. Waldvogel. Space decomposition techniques for fast layer-4 switching. In *Proceedings of IFIP Workshop on Protocols for High Speed Networks*, Salem, Massachusetts, August 25–28 1999.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

[4] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proceedings of ACM Sigcomm*, pages 191–202, Vancouver, Canada, August 31–September 4 1998.

[5] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM Sigcomm*, pages 147–160, Cambridge, Massachusetts, August 30–September 3 1999.

[6] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of ACM Sigcomm*, pages 135–146, Cambridge, Massachusetts, August 30–September 3 1999.

[7] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM Sigcomm*, pages 191–202, Vancouver, Canada, August 31–September 4 1998.

[8] D. Stiliadis and T.V. Lakshman. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM Sigcomm*, pages 203–214, Vancouver, Canada, August 31–September 4 1998.

[9] H.-Y. Tzeng. Longest prefix search using compressed trees. In *Proceedings of IEEE Globecom*, Sydney, Australia, November 8–12 1998.