Fast packet classification with a varying rule set

Shashidhar Merugu[†] Ajay Gummalla[‡] Dolors Sala[‡] Ellen Zegura[†]

[†] Networking & Telecommunications Group, College of Computing, Georgia Tech.
{merugu,ewz}@cc.gatech.edu

[‡] Residential Broadband Unit, Broadcom Corporation
{ajay,dolors}@broadcom.com

Abstract

Multi-dimensional packet classification is increasingly important for applications ranging from fire-walls to traffic accounting. Fast link speeds, the desire to classify with fine granularity, and the need for agility in a dynamic environment all pose significant challenges for packet classification. We propose an approach that is capable of handling a changing set of classification rules that span multiple fields. Our approach is based on extracting a relatively small set of bits that uniquely identify the packets satisfying each rule. Changes to the rule set are handled in-line via a fast update mode that adds to the rule table, while a background process performs reoptimization of the full rule table less frequently. The classification process can be efficiently implemented using pipelined hardware and supports high packet arrival rate.

1 Introduction

Packet classification refers to the task of determining which rule(s) from a rule set are matched for a packet, based on the contents of the packet headers. For example, standard IP routing requires packet classification using the destination address field of the IP packet, with the rules stated as address prefixes. A rule matches a packet if the destination address agrees with the address prefix. Multiple rules may match; the rule with the longest prefix "wins".

Research efforts in recent years have resulted in algorithms able to classify packets for IP routing at high speeds [13, 18, 5]. Thus, interest has turned to more general packet classification problems [9, 7, 15, 19, 6, 11, 16], referred to as *multi-dimensional packet classification*, where the rules span multiple fields. Multi-dimensional packet classification is increasingly important for applications ranging from fire-walls to traffic accounting. For example, an intrusion detection database will contain rules to describe packet headers that indicate a potential attack. Depending on the type of attack, different packet header fields will be relevant. For example, a SYN-flood attack may require examining destination IP address, TCP port number and TCP flags field.

The packet classification problem [9] can be formally defined based on the structure of a rule and the definition of what it means for a packet to match a rule. Packet classification algorithms are typically evaluated on the following criteria:

- **Speed of classification.** To keep up with line rates, classifiers must be capable of handling as many packets per second as the link can deliver.
- **Size of rule tables supported.** A survey of real classifiers circa 1999 found fairly small rule tables [7]. However, the support of new applications and different degrees of QoS on a per flow basis changes the scenario. Classification on a per flow basis can increase the rule set enormously to thousands of rules. Pre-processing time for data structures, memory

requirements, and lookup speed will generally all grow with the size of the rule table, and will eventually limit the practical size of the rule table.

- **Ease of updates.** Any scheme can handle modifications to the rule table via complete recomputation of the lookup data structures, however such an approach will typically severely limit the frequency of updates. Alternatives that support incremental modification are desirable for applications with frequent changes.

- **Generality.** A spectrum of rule types exist, including simple prefixes, non-contiguous masks, operators (range, less than, greater than) and wild-cards. A more general approach supports a larger set of rule types.

The focus of this paper is multi-dimensional packet classification with relatively large tables (up to 8000 rules) that may change relatively frequently (about 20% increase in size of rule table) and must classify at high speeds (up to OC192 or 31.25 million packets per second). Our approach takes inspiration from two observations. First, the size and cost of content-addressable memories (CAMs) have improved steadily over time to become viable to consider for hardware implementation of packet classifiers[1]. Second, packet classification can be viewed as an instance of the feature selection problem [4, 3, 10] that has been studied extensively in the database, artificial intelligence and theory communities.

Our approach is based on extracting a relatively small set of bits that uniquely identify the packets satisfying each rule. Changes to the rule set are handled in-line via a fast update mode that adds to the rule table, while a background process performs reoptimization of the full rule table less frequently. The classification process can be efficiently implemented using pipelined hardware and supports high packet arrival rate.

The remainder of the paper is organized as follows. The next section describes our packet classification architecture and the algorithms that build and maintain classification rules in necessary data structures. In Section 3 we sketch the design of a hardware implementation. We evaluate the performance of the approach using both analysis and simulation in Section 4. Section 5 describes related work. Finally, we present concluding remarks in Section 6.

2 Architecture and Algorithms

2.1 Overview

One of the key observations made by Gupta and McKeown in their study of real life packet classification rule sets [7] is that the number of distinct values for a particular field in a rule set is far less than its allocated space of values. For example, the transport protocol field across all rules in a rule set contains very few distinct values (e.g. IPPROTO_TCP, IPPROTO_UDP, IPPROTO_ICMP) whereas its allocated space of 8 bits can have 256 values. This observation holds when we extend the scope from a field to a more general collection of bits called "chunk". The authors take advantage of this observation and encode the N distinct values in a particular chunk using $\lceil \log N \rceil$ bits. While this code is an optimal one, it requires pre-computation of the map from given values in rules to assigned codes. On the data path, complex logic is needed to figure out the correct encoding of a random value in packet. Further, when the rule set changes (either addition or deletion of rules), this encoding is no longer valid and cannot be corrected easily.

[1] Though the price of a CAM depends on vendor and exact specifications, it roughly costs less than $5.00 per 1024 words, with 64-bit words [1].

Our solution is based on key insights from previous work on "feature selection" in database, artificial intelligence and theory communities [4, 3, 10]. The main idea of our algorithm is to select a subset of bit positions[2] such that the bits extracted from these positions represent the entire data set. In particular, if our data set comprises of M bit vectors each of size k bits, we hope to select a subset l out of k bit positions such that the bits in these l positions for all M bit vectors best represent the entire data set. Figure 1 illustrates this idea. While the number l of representative bit locations is certainly higher than the optimal $\lceil \log N \rceil$ bits, we have a simpler bit extraction function (a bit mask operation) on the data path. Any modifications to the rule set can be performed easily by addition or deletion of representative bits. However, as always with any vector projection, the process of reduction of k bit vectors to l dimensional space leads to collisions. Our goal in this process is to minimize these collisions, ideally to none. As we shall see in our description of our algorithms and evaluation using real-life packet traces that our scheme works well in practice.

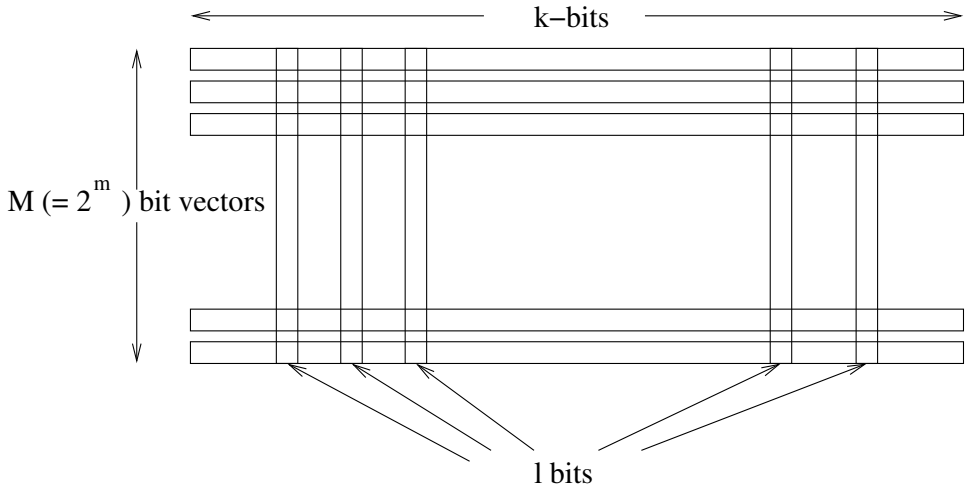


Figure 1: Selection of l representative bit locations for k-bit wide vectors

2.2 Architecture

Our proposed packet classification system is shown in Figure 2. The architecture can be divided into two main components: control and classification. The classification process operates on the data path and identifies what to do with a packet when it arrives. The outcome of the classification is an action ID, obtained from a table that contains all classification rules. The rule table is built and updated by the control process. Updates are necessary when the rule set changes, which happens on a considerably slower time scale than the packet arrival rate. Since the classification path is critical for our system, we have designed it to keep the operations at a minimum. However, the control path, which is executed as a background process, does include complex operations.

The classification process starts with the arrival of a new packet as shown in the right side of Figure 2. It includes a lookup stage and a verification stage. In the lookup stage, a bit mask ($\mathcal{H}$) is applied to the packet to extract a lookup key. A lookup algorithm uses this key to identify a rule that matches this packet. As explained earlier, collisions may occur in our design. Thus the lookup stage might identify a *bucket* of rules instead of a single rule. In this case, the verification stage determines which rule within the bucket is the one that matches this particular packet. Once a rule is identified, it contains an action to be performed on the packet. Since the execution of this

[2] We use the words "bit positions" and "bit locations" interchangeably.

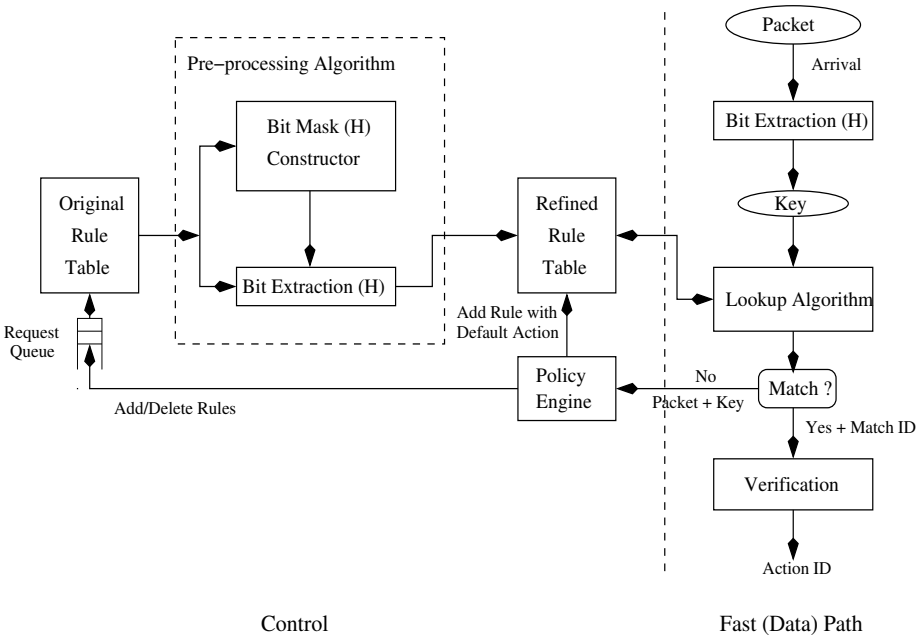


Figure 2: Architecture of our packet classification system

action is independent of classification, we do not include it in our fast path. However it is handled by subsequent stages in the overall system. Thus, the action identification is the end of one run through the classification process and the classifier returns to process the next packet.

The control process handles creation and modification[3] of the rule table used for lookups (the "refined rule table"), as well as construction of the bit mask used to extract a lookup key from a packet. The refined rule table is constructed by processing the original rule table using a bit selection algorithm. The algorithm produces a bit mask used to extract a lookup key. The same bit mask is applied to all rules in the original rule table to produce the (smaller) refined rule table.

An important characteristic of this control process are the two modes of operation: off-line-processing and fast update. The off-line-processing mode updates the rule table by reconstructing the table from scratch. On the other hand, the fast update mode is designed to add rules quickly so that the rule is available as soon as possible.

In the fast update mode, the policy engine adds a new rule by using the current bit mask. This addition could create collisions in the refined rule table and hence the bit mask may lose its optimality. Though the system can still operate under these less optimal conditions, it reconstructs the bit mask once in a while, in order to preserve optimality. The tradeoff between optimality and processing requirements is controlled by frequency of reconstruction of the bit mask.

In summary, this architecture is based on a variable bit mask coupled with a fast update policy to achieve high classification rates on data path, while reflecting any changes in the set of rules very quickly. Specific algorithms proposed for this architecture are presented in the following subsection.

[3]Addition of new rules and deletion of old rules is triggered by some policy implemented by "policy engine" module.

2.3 Bit selection algorithm

The algorithm to construct the bit mask starts with the entire set of classification rules, each of size k bits. At each iteration, we select a new bit, such that the original set of rules is divided into two subsets; one subset of rules with a zero in that chosen bit location, while the other subset of rules had a one in the same location. The choice of bit location is based on maximizing the division of the sets of rules from the previous stage.

A pseudo code description of the algorithm to select bits is shown in Figure 3. The set of rules R is an input to this algorithm, along with two parameters: number of bits l to be selected and maximum number of collisions c allowed. While the first parameter l determines the maximum depth of the decision tree (also referred to as "set division tree", since each decision leads to a smaller set of possible matching rules), the second parameter c determines the maximum number of rules at a leaf node. In the pseudo code, r_i denotes the i^{th} rule and b_j denotes the j^{th} bit position. $\mathcal{Z}_{b_i}$ and $\mathcal{O}_{b_i}$ denote subsets of rules that have 0 and 1 respectively at bit position b_i. The algorithm proceeds in a greedy fashion selecting the best bit position at every level of depth d in the set division tree. Ω_{d_1}, Ω_{d_2}, ... Ω_{d_j} represent subsets of rules associated with internal nodes at depth d. Each outstanding bit position b_i is assigned a weight based on the extent to which the sets Ω_{d_j} would be pruned, if we were to select b_i for division at the current level of depth d. $\Omega_{d_j} \cap \mathcal{Z}_{b_i}$ and $\Omega_{d_j} \cap \mathcal{O}_{b_i}$ denote the children of Ω_{d_j} if we were to select b_i for division. The extent of division is measured in terms of difference in cardinality of these children. Thus lower the sum $\sum_{d_j} [|\Omega_{d_j} \cap \mathcal{Z}_{b_i}| - |\Omega_{d_j} \cap \mathcal{O}_{b_i}|]^2$, greater is the division. So the greedy algorithm picks the bit position that minimizes this metric in every level of decision tree. The algorithm terminates when either we have selected the required number of bit positions or the number of rules in leaf node is below the acceptable number of collisions.

	Bit Position							
Rule Index	1	**2**	3	4	5	6	7	**8**
r_1	1	**0**	0	0	1	0	1	**1**
r_2	1	**0**	0	0	1	1	0	**0**
r_3	0	**1**	1	0	1	1	0	**1**
r_4	1	**1**	0	1	1	1	0	**0**

Table 1: Example rule set to illustrate algorithm to construct Bit Mask

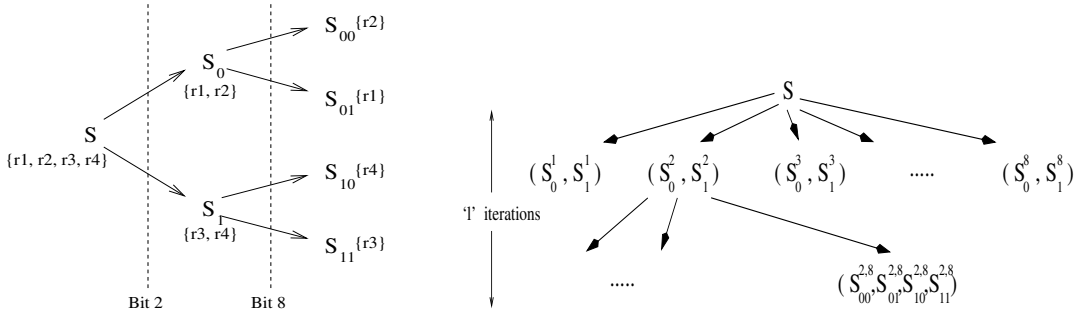
Table 1 contains a simple set of $M = 4$ rules that have $k = 8$ bits each. We select bit mask for this rule set using the algorithm described in this section. An illustration of how the rule set is divided as the algorithm proceeds is shown in Figure 4(a). In the example, bit 2 divides the original set S with four rules $\{r_1, r_2, r_3, r_4\}$ into two subsets: S_0:$\{r_1, r_2\}$ and S_1:$\{r_3, r_4\}$, containing 0 and 1 respectively in bit location: 2. Next, bit 8 divides these two sets S_0 and S_1 further into two subsets each. Finally, we have four subsets S_{00}:$\{r_2\}$, S_{01}:$\{r_1\}$, S_{10}:$\{r_4\}$, S_{11}:$\{r_3\}$, each of them with just one rule. Thus the bit locations 2 and 8 are representative of the entire of rules. Figure 4(b) describes the search tree used in the greedy algorithm to select bits. In the search tree, a pair $(S_0^{b_i}, S_1^{b_i})$ denote two subsets of S (root), if we were to select bit b_i for division. We follow similar notation for the second level of division. At every iteration, we list all possible divisions of subsets and choose the best one based on a weight function. This weight function maximizes the set division at that level. In the example, in the first iteration, the choice of (S_0^2, S_1^2) gave the best weight as it divided the set S gave exactly into half. Once a branch is chosen, we pursue in

```
SelectBits(RuleSet R, NumBits l, MaxCollisions c)
    {
        /* R has M rules, each of which is k bits wide */
        /* This algorithm attempts to select l bits such that the number of collisions is less than c */
        Ω ← {r_1, r_2, ..., r_M}; /* Ω includes all rules from R */
        K ← {b_1, b_2, ..., b_k}; /* K is a set of all bit locations */
        for each bit location b_i ∈ K do
            {
                Z_{b_i} ← {r_j | r_j has a zero at bit location b_i };
                O_{b_i} ← {r_j | r_j has a one at bit location b_i };
            }
        /* Note that ∀b_i, Z_{b_i} ∩ O_{b_i} = φ and Z_{b_i} ∪ O_{b_i} = Ω */
        L ← φ; /* L is set of bits that are selected so far */
        d ← 0; /* d is current depth of set division tree */
        Ω_0 ← Ω; /* Ω_0 is root of set division tree */
        divideFlag ← (|Ω_0| > c); /* divideFlag indicates whether to proceed with set division or not */

        while ((divideFlag) && (d < l)) do
            {
                /* Choose the next bit location */
                for each bit location b_i ∈ K − L do
                    {
                        W_{b_i} ← 0; /* W_{b_i} represents the weight of bit location b_i in this iteration */
                        for each node Ω_{d_j} at depth d do
                            W_{b_i} ← W_{b_i} + [|Ω_{d_j} ∩ Z_{b_i}| − |Ω_{d_j} ∩ O_{b_i}|]^2
                    }
                L ← L ∪ {b_min}; /* where W_{b_min} is minimum of all weights in this iteration */
                for each node Ω_{d_j} at depth d do
                    add Ω_{d_j} ∩ Z_{b_i} and Ω_{d_j} ∩ O_{b_i} as children;
                d ← d + 1; /* Extending the tree one step deeper */
                If ∀ Ω_{d_j} |Ω_{d_j}| < c then /* all nodes Ω_{d_j} at depth d have less elements than c */
                    divideFlag ← FALSE;
            } /* end of while */

        return L; /* Return the set of selected bits */
    }
```

Figure 3: Pseudo code of algorithm to select bits



(a) Illustration of division of rule set given in table 1

(b) Search tree in algorithm to select bits

Figure 4: Trees used in bit selection algorithm

that direction, ignoring the rest of the options. Here the siblings (S_0^1, S_1^1), (S_0^3, S_1^3), ... (S_0^8, S_1^8) are not considered any more. In the second iteration, the choice of $(S_{00}^{2,8}, S_{01}^{2,8}, S_{10}^{2,8}, S_{11}^{2,8})$ was the best because it divided the sets (S_0^2, S_1^2) equally. Thus, the bit selection algorithm uses a greedy approach. Several observations can be made from this example. First, the order of bit selection is not important as $S_{10}^{2,8}$ is same as $S_{01}^{8,2}$. Second, the minimum number of bits required to identify the rules uniquely is equal to the depth of the tree at which all leaves have singleton sets. Third and last, the size of the search tree is too large (exponential in size) to be able to explore all possibilities.

2.4 Discussion

Though the description in previous section assumes that all rules contain only binary digits 0 and 1, we can extend our bit selection algorithm to fit a more general case that uses wild cards. The rules follow ternary notation $(1, 0, *)$ and certain modifications are necessary to this algorithm. When a set is partitioned, a rule with a wild card at the chosen bit location is replicated in both children. The metric that measures the extent of partition is now a weighted average of two parts: first, that computes the evenness of division of each set, second, that keeps the average cardinality of children low. While the first part ensures that the children are almost the same size, the second part avoids bit locations with wild cards, since they lead to duplication of rules. An extensive evaluation of these modifications to our algorithm is part of our future plan.

3 Suggestions for implementation in hardware

Figure 5 shows a hardware implementation of the data path for our packet classification system. The first stage is the bit extraction from the packet header. This is implemented as a cross bar which selects the relevant bits and generates a key for the lookup stage. This key generation is accomplished in one clock cycle. The lookup stage is implemented using Content Addressable Memory (CAM). The CAM matches the key against all its entries and returns a MatchID. This lookup is also performed in one clock cycle. The MatchID points to a block of memory in the SRAM where the complete rule is stored. The comparator stage compares the header of the received packet against the retrieved rule. In the case when there are collisions, the comparator sequentially compares the retrieved set of rules. This linear search takes, at most, c cycles where c is the maximum number of collisions. Hence, on average, this pipelined implementation can process one packet every c cycles. On a 125 MHz system with maximum number of collisions $c = 4$, this system can obtain a throughput of 31.25 million packets per second. In the following sections, we assume that our system operates with M ($= 2^m$ say) rules for classification. Each rule is k bits wide and our bit selection algorithm selects l representative bits.

3.1 Bit Extraction

The bit extraction module is essentially a $k \times l$ multiplexor which selects the relevant l bits among the k bit header fields. This can be easily implemented in hardware as crossbar switch with k inputs and l outputs. The l bit output is the key that is used in the lookup stage. The implementation of a crossbar has two elements. First is the switching matrix that requires $k \times l$ switches. For example, a 256×64 multiplexor requires $16K$ switches and can be implemented with current VLSI technologies. The second element is the control logic that manages the crossbar. This crossbar needs to be reconfigured only when the bit mask changes. Since the reconfiguration of the switch is a low frequency event off the fast path, this leads to a crossbar implementation that is simple and real-estate efficient.

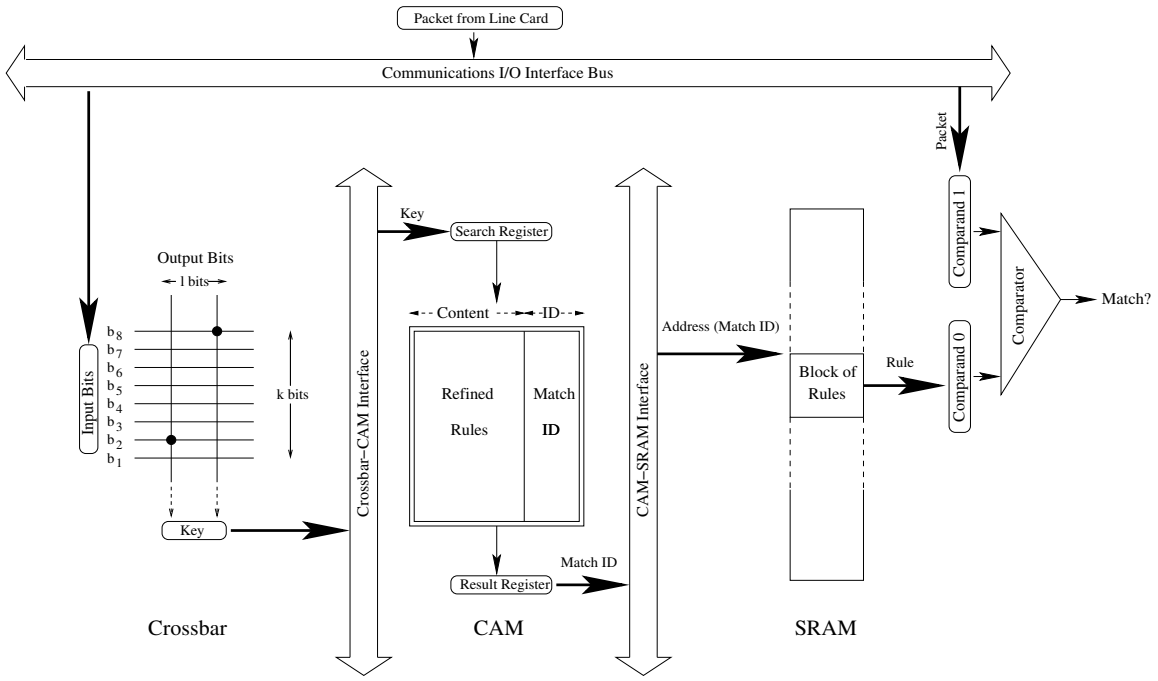


Figure 5: An illustration of the hardware implementation of the packet classifier

As an alternative design, when more delay can be tolerated through the system, this multiplexor can be implemented as a simple bit shifter. Such a system can generate the key in $(k + l)$ clock cycles. This value comes from the fact there are at least k shift operations and l store operations. However this design limits the packet processing capability to one packet every $(k + l)$ clock cycles.

3.2 Lookup

The next step in the fast path is the lookup stage. The lookup in our algorithm is to match the input key l bit string against the refined rule table. Such fixed length string comparisons can be implemented in hardware using Content Addressable Memory (CAM) which allow fast and parallel comparisons. CAMs, which were once expensive, are becoming increasingly ubiquitous and cheaper. Commercial CAMs of sizes $64K$ words $\times$ 128 bits/word are available today [12, 14].

The refined table is stored in a binary CAM. Each entry in the CAM has two parts: first part is an l bit vector obtained by extracting l bits from a rule and second part is a Match ID (x bits in width) of the bucket containing the corresponding rule. The CAM required in our implementation is M words long and $l + x$ bits wide. The key obtained from the bit extraction is used in the CAM lookup. The lookup is a one cycle operation and the result is a success or failure with the appropriate MatchID. For buckets of size 1, CAM lookup returns the MatchID associated with the rule. On the other hand, if the buckets are of size greater than 1, the MatchID returns an identifier of the bucket and the verification stage has to continue the classification further.

3.3 Verification

The purpose of verification is to confirm if the lookup stage is correct, and to classify further if buckets have more than one rule in them. We use the MatchID obtained from lookup stage to locate

the candidate rule(s) that could potentially match the packet in consideration. The candidate rule(s) accessed from memory can be compared with k bits in the packet either in sequence or in parallel. Sequential comparison needs one k-bit-wide comparator. The candidate rule(s) are searched in sequence, hence this takes as many as c cycles. In cases where c is a small value, a group of c such comparators can be provided for faster parallel comparison. As an alternative to comparator logic, the candidate rules could be stored in another CAM prefixed with the MatchID of the bucket. However, this would require a CAM with large width to store an entire rule along with the bucket ID as a prefix. Though this option is good in terms of clock cycles, it is not cost effective.

3.4 Bit Selection Algorithm

Since the bit selection algorithm does not fall in the fast path, it can be executed without the need for a hard deadline. Hence it can be run off-line on a separate processor or a multi-processor, preferably with spare *cpu* cycles and capable of executing mathematical operations (e.g. set intersection) efficiently. Though the execution of the bit selection algorithm does not have any direct effect on the fast path, its frequency of execution determines the optimality of refined rule table. Therefore it is imperative to choose good time-out period that controls the frequency of optimization.

4 Evaluation

We evaluate the packet classification system both analytically and through simulations. We start with a mathematical analysis that assumes each rule is equally likely. The simulation results described in the second half of this section evaluate the different parameters of our classifier system under rule sets taken from packet traces.

4.1 Mathematical Analysis

We derive statistical results that are based on the assumption that each rule (k-bit vector) in the given set of M ($= 2^m$ say) rules (bit vectors) follows a uniform random distribution. In other words, each bit has equal probability of being zero or one. This assumption represents an ideal case.

The goal of this analysis is to determine how large l must be so that l bits are sufficient to search the rule set. As we mentioned in section 2, dimensionality reduction leads to collisions. Our goal is to minimize these collisions, ideally to none, by selecting the right set of l bits. We derive the probability that there exists an l bit vector that uniquely identifies each original k bit vector. The higher the probability, the better are the chances to find a right set of l bits.

We use some basic results from combinatorics in this analysis. Given l bit locations (i.e. the bit locations are fixed), the probability that any set of 2^m bit patterns, each of size k bits, has no collision at all is given by:

$$P_{nc} = \frac{\binom{2^l}{2^m} \times \binom{2^{(k-l)}}{1}^{2^m}}{\binom{2^k}{2^m}} = \frac{\binom{2^l}{2^m} \times 2^{(k-l)\times 2^m}}{\binom{2^k}{2^m}}$$

The denominator represents the total number of ways to construct a set of 2^m bit patterns, each of size k bits. The numerator represents the number of ways to construct good sets (without collision in l bits). To construct a good set, we first fill in the given l bit locations from the 2^l vector space, and the rest of $(k-l)$ bits in any order.

Given l bit locations, the probability that in any set of 2^m bit patterns, each of size k bits, has one or more collisions is given by: $P_c = 1 - P_{nc}$

The probability that all chosen subsets of l bit locations (out of a possible k bit locations), results in at least one collision is: $P_c^{\binom{k}{l}}$. The exponent represents the number of ways to select l bit locations from a possible k locations. The probability that there is at least one subset of l bit locations, such that there is no collision at all is given by:

$$\mathcal{P} = 1 - P_c^{\binom{k}{l}} = 1 - [1 - P_{nc}]^{\binom{k}{l}} = 1 - \left[1 - \frac{\binom{2^l}{2^m} \times 2^{(k-l)\times 2^m}}{\binom{2^k}{2^m}}\right]^{\binom{k}{l}}$$

As detailed in Appendix A the probability $\mathcal{P}$ can be approximated as

$$\mathcal{P} \approx 1 - 2^{[(2m-l-1)\times\binom{k}{l}]}$$

Note that the exponent of the second term $[(2m - l - 1) \times \binom{k}{l}]$ is negative for $l > (2m - 1)$ and $\binom{k}{l}$ is a very large value when l is close to $k/2$. From the above expression, we can observe that $\mathcal{P}$ is close to 1 as $l > (2m - 1)$.

4.2 Simulation Results

In this section, we present results of experiments to study the bit selection algorithm and classification. All the algorithms for bit selection and classification are written in C and executed on a 440 MHz Sun UltraSPARC-IIi running the SunOS 5.7 operating system. The timing measurements are gathered using a Solaris nanosecond resolution timer[4].

The rule sets used in all our experiments are constructed from real packet traces collected at an access router providing service to residential users. Due to the unavailability of rule sets of the magnitude and dimensions that we are interested in this work, we had to work our way backwards from the packet traces to generate the rule sets. Specifically, our approach was to fix the protocol headers and fields that we are interested in, and examine every packet in the trace to collect information on valid protocol headers/fields. The rules, thus generated, were as random as the packets in a trace can be, but were certainly more correlated to each other than those that can be constructed using a pseudo-random algorithm. By default, each rule comprises of five fields: $[ip\ src, ip\ dst, ip\ proto, xport\ sport, xport\ dport]$. The following experiments include both a study of the bit selection algorithm as well as the performance of our classifier (as implemented in software) on a stream of packets.

In our first set of experiments, we started with rule sets of different sizes varying from 128 rules to 8015 rules. For each rule set, our bit selection algorithm constructed a bit mask to classify rules as a function of the maximum number of collisions (c as defined in section 2) that are acceptable. The first observation to be made in Figure 6 is the importance of allowing collisions: the larger the number of collisions allowed, the fewer bits are needed. If collisions are not allowed at all, the rules need to be uniquely identified, hence the number of bits needed is quite large. For example, a rule set of 1024 rules needs 23 bits to operate with no collisions. On the other hand, if one more collision is allowed this number drops to 18. Further, 4, 8, and 16 collisions need 15, 12, and 8 bits respectively. Note that while the number of bits decreases rapidly as number of collisions increase, this increase is less once the number of collisions is relatively large. This indicates that there is no advantage in designing the hardware to handle a very large number of collisions.

[4]Please see `gethrtime(3c)` manual page on SunOS 5.5+.

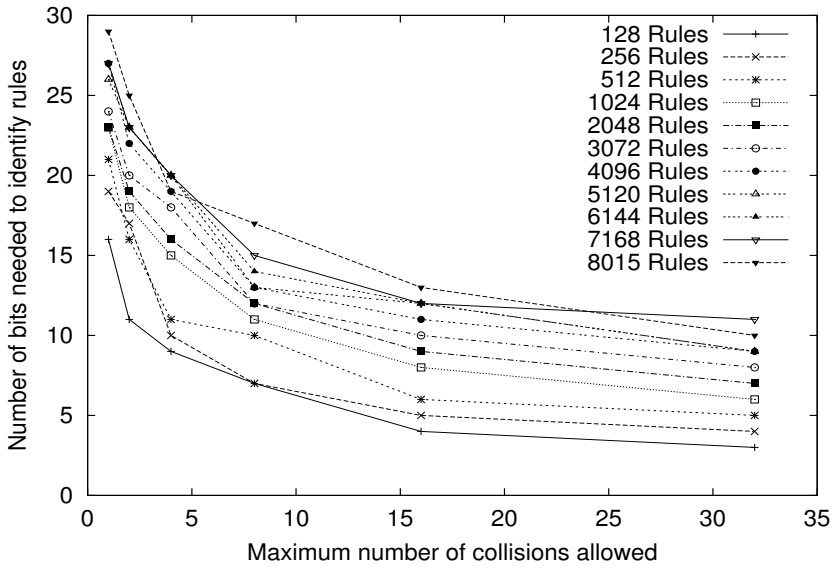


Figure 6: Study of bit selection algorithm

The second observation in this figure is the effect of the size of the rule set. As expected, the number of bits required is more for a large rule set. But this difference becomes almost constant if the number of collisions is relatively large. Hence we can say that the effect of the maximum number of collisions is fairly independent of the size of the rule set.

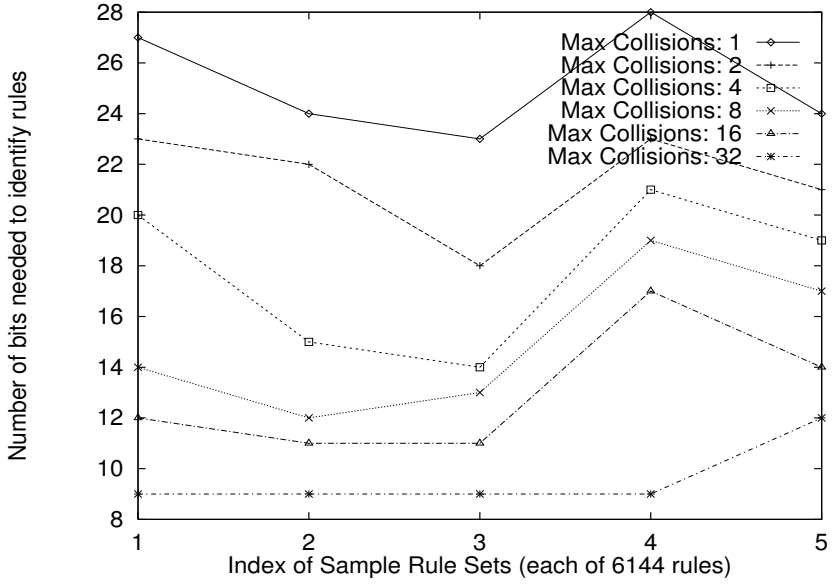


Figure 7: Dependence on rule set

Figure 7 evaluates the statistical variation of the number of bits needed, by conducting experiments on five different sample rule sets. Our sample rule sets (each of size 6144 rules) are generated from different portions of the the packet trace. The relative variation in the results of these experiments is fairly flat indicating that the actual rules in these rule sets certainly have an effect on the bit selection algorithm, but the effect (at least in rules inferred from a packet trace)

is fairly minimal.

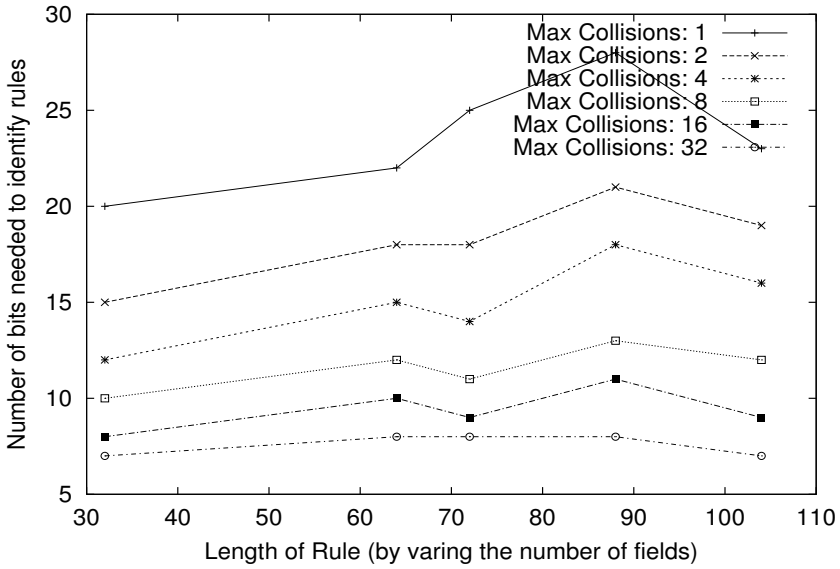


Figure 8: Effect of change in length of a rule

An interesting property of this algorithm is that the number of bits needed to identify the rule set is fairly independent on the number of fields the rule is comprised of. In this next set of experiments, we have kept the size of the rule set constant at 2048 rules, while the length of rules has been varied across the rule sets. We start with rules that have only one field $[ip\ src] = 32$ bits, and increment the length of the rule by adding successive fields $[ip\ dst]$(32 bits), $[ip\ proto]$(8 bits), $[xport\ sport]$(16 bits), $[xport\ dport]$(16 bits), in that order. The algorithm selects different bits if the length of the rule set is different, but since it still needs to distinguish the same number of rules, it does not necessarily require more bits as illustrated by the flat trend in the Figure 8.

We observe that there is a significant difference between the maximum number of collisions the system can handle versus the number of collisions that actually happen during operation. Figure 9 measures this difference for several sizes of rule sets.

The results indicate that a large number of rules can be added without reaching the maximum number of collisions. This helps in keeping the frequency of re-optimization of classification rule set low.

Number of Original Rules	Random Addition of New Rules		Restricted Addition of New Rules	
	Number	Percentage	Number	Percentage
128	230.5	180.08	60.3	47.11
256	248.7	97.15	54.5	21.29
512	542.6	105.98	130.3	25.45
1024	8665.8	846.26	550.4	53.75
2048	11951.2	583.55	397.8	19.42

Table 2: Addition of new rules to the rule table

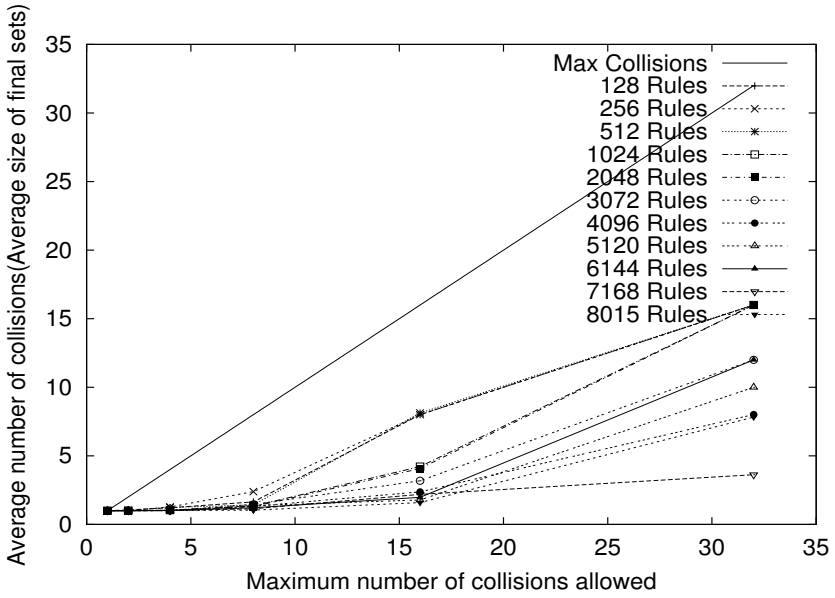


Figure 9: Variation of actual number of collisions

A significant number of random rules can be added to an existing rule set because the actually number of rules is very small compared to the total possible number of rules that is determined by the length of the chosen bit mask[5]. However, as illustrated in Table 2, the correlation between the original rules and the new rules certainly has an effect on rule addition. It is highly likely that a new rule does not conflict with the current one if there is no correlation (as indicated by the *random addition* columns). However, if the new rule is highly correlated to the existing ones, the collisions happen more frequently (*restricted addition* columns). This is the case when the rules come from the same trace. Even in the highly correlated case, the rule set can be increased by about 20% in size before it needs to be re-optimized.

Figure 10 shows the processing time required to construct the refined rule table. The computation required to build a collision free table is large, since the algorithm must go to the lowest depth of the tree. However as more collisions are allowed, the algorithm can stop at higher depth levels, hence there is an exponential reduction in the number of nodes to be explored in the search tree. This translates to the sharp exponential decrease in processing time shown in the figure for all rule set sizes. Obviously, the larger the rule set, the larger the processing time required.

The classification time for packets along the data path, as implemented in software, is evaluated in Table 3. In this experiment we have used rule tables of different sizes from 128 rules to 7168 rules. We have kept the size of each rule constant at 5 fields (104 bits). The average classification time taken for 100,000 randomly generated packets is listed in the right column. For a rule set of size 1024 rules, it takes about 967 nano-seconds to classify a single packet. This corresponds to a throughput of about 1 million packets per second. This classification in software is performed as a traversal of the set division tree constructed during bit selection. However, when this lookup is actually implemented in a CAM as suggested in Section 3, the system can handle up to 31.25 million packets per second along the data path.

[5] A bit mask of length l can potentially handle 2^l rules.

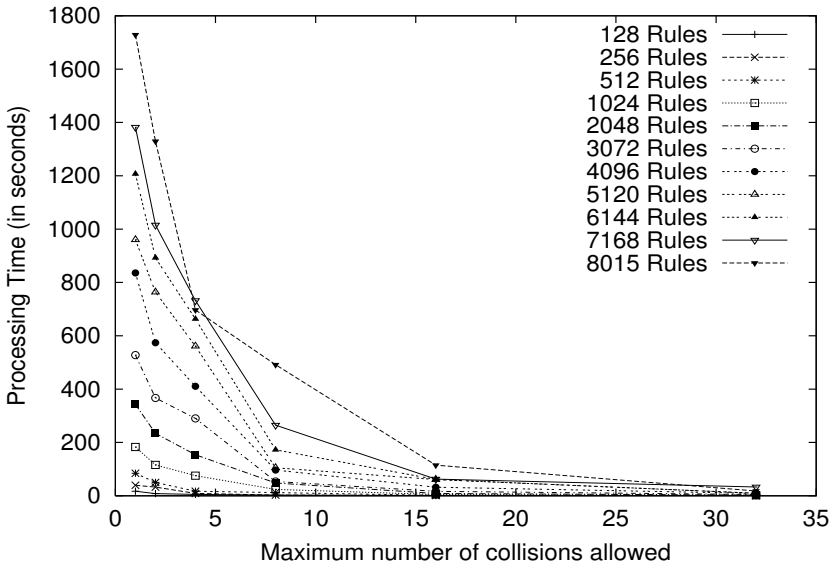


Figure 10: Variation of processing time of bit selection algorithm

Number of Rules	Avg. Classification Time (nanoseconds)
128	714
256	798
512	843
1024	967
2048	1100
3072	1212
4096	1304
5120	1435
6144	1229
7168	1598

Table 3: Variation of average packet classification time vs. number of rules

5 Related Work

Recent advances in non-best effort services, such as QoS, accounting, intrusion detection, clearly indicate a need to distinguish and classify traffic into different flows for suitable processing. This increase in emphasis has led to a variety of proposals for packet classification, especially to solve the multi-dimensional version. In their exhaustive survey [9] of current research in packet classification, Gupta and McKeown examine both hardware and software solutions to this problem. The taxonomy suggested in this survey categorizes different solutions into four groups:

- Simple techniques like linear search, caching and those that use basic data structures like hierarchical trees, set-pruning tries [17]
- Geometry based approaches such as grid-of-tries [16], area-based quad-tree [2], fat inverted segment tree [6]
- Heuristic based solutions such as recursive flow classification [7], hierarchical cuttings [8], tuple-space search [15]
- Finally, those solutions that are designed with special hardware such as Ternary CAMs and FPGAs [11]

Our work presented in this paper falls into the third category of solutions based on heuristics, yet focused on being amenable to a hardware implementation. Though we highlight the solutions in third group briefly next, we encourage the reader to refer this survey for a comparative study of various approaches.

Gupta and McKeown develop a multi-stage algorithm called Recursive Flow Classification (RFC) [7]. RFC recursively reduces the S bits of the packet header to T bits of classifier ID, by essentially extracting the unique values of fields and representing those values using fewer bits than in the original headers. They use heuristics to determine the number of phases of recursion and ways to combine packet header fields in each phase. The basic RFC algorithm has two limitations. First, the space and pre-processing time become problematic for large rule tables. Second, the scheme does not support incremental updates and therefore is suitable only if the rule set changes relatively slowly.

Srinivasan *et. al.* propose an interesting scheme called Tuple Space Search algorithm [15] that builds on hash tables. The algorithm constructs d-tuple for every d-dimension rule such that the i^{th} component is equal to the prefix length in i^{th} dimension of the rule. Two rules are mapped on to the same tuple if the prefixes in every dimension are fixed and equal in length. All the rules that are mapped on to a single tuple are entered into a hash table associated with that tuple. A classification operation is broken into exact match operations on each of the hash tables. Incremental updates are very simple and require only one insertion operation in the correct hash table. This solution works very well for multiple dimensions if the number of tuples is small on an average. However the number of tuples could be very large in the worst case, with practically no benefit in transition to tuple space from original space of rules. This scheme works only for prefix specification, thus limiting the generality of the rules. Thirdly, the time taken for a search along each hash table is non-deterministic since there are variable number of entries in hash tables.

Hierarchical Intelligent Cuttings (HiCuts) [8] and Modular Approach by Woo [19] bear a strong similarity to our work. These three schemes, including ours, use a decision tree to narrow down from a large original set of rules at its root to a small set of possible matching rules (filter bucket) at a leaf. This is followed by a search phase within the filter bucket for the exact matching rule. The decision tree is constructed while pre-processing the original set of rules. However there are

subtle differences between these three schemes. The local decision in HiCuts at an internal node is based on equal sized cuts along a dimension. While in Woo's modular approach and our work, it is based on a clever selection of bit(s) irrespective of dimensions. In our work, the bit selected to make a local decision is the same for all internal nodes at the same level of depth. The optimization strategy used in our decision tree construction examines all internal nodes at a level together to select the best bit position. On the other hand, in both HiCuts and Woo's modular approach, the local decision (i.e. dimension to cut, selected bit) at an internal node is different from that of other internal nodes at the same level. While their scheme leads to a decision tree with a short overall depth, it has certain disadvantages. Since the local decision at a node affects the local decision at a child node, it requires a sequential examination and hence a linear traversal of the decision tree. Such a sequence cannot be pipelined well, unless we incorporate multiple levels of look-ahead. Also since the length of the path to a leaf is variable, it takes variable amount of time for each classification operation, thus limiting the maximum speed of classification. Though our scheme results in decision trees of longer overall depth, we have the advantage of parallel execution of decisions, each corresponding to a level of depth in decision tree. Since each local decision in our scheme is to examine a bit position, we can leverage CAM technology to perform the decision tree traversal in a single clock cycle.

6 Conclusions

For a variety of applications, classifying packets requires matching in multiple dimensions (fields) of the packet headers. The problem of multi-dimensional packet classification becomes even more complex when the set of these rules change with time. We observed a simple analogy of the classification problem with classical feature selection problem, thus realizing a hash function to extract bits from a packet using a mask. We have presented a hardware-friendly framework and algorithms based on bit selection. Our approach has been backed by a mathematical analysis and a practical study using a trace of packets. Our framework for classification includes modules for optimization of rule set, bit extraction, lookup and variation in rule set. Finally, we proposed ways to implement various modules of our architecture and algorithms efficiently in hardware.

A Approximating $\mathcal{P}$

Recall that the function we are interested in is:

$$f(l) = 1 - \left[1 - \frac{\binom{2^l}{2^m} \times 2^{(k-l)\times 2^m}}{\binom{2^k}{2^m}}\right]^{\binom{k}{l}}, \quad where \quad m \leq l \leq k, \quad and \quad m, k \quad are \quad constants.$$

Remark: A quick look at the value of the function $f(l)$ at the boundaries gives:

$$f(m) = 1 - \left[1 - \frac{2^{(k-m)\times 2^m}}{\binom{2^k}{2^m}}\right]^{\binom{k}{m}} \quad and \quad f(k) = 1$$

Breaking this function $f(l)$ into parts and examining the behavior separately:

$$f(l) = 1 - [1 - g(l)]^{\binom{k}{l}}$$

$$where \quad g(l) = \frac{\binom{2^l}{2^m} \times 2^{(k-l)\times 2^m}}{\binom{2^k}{2^m}}$$

Simplifying the expression for $g(l)$, we get:

$$g(l) = \frac{2^l!}{(2^l - 2^m)! \times 2^m!} \times \frac{(2^k - 2^m)! \times 2^m!}{2^k!} \times 2^{(k-l)\times 2^m}$$

$$g(l) = \frac{2^l.(2^l - 1)\ldots(2^l - 2^m + 1)}{2^k.(2^k - 1)\ldots(2^k - 2^m + 1)} \times 2^{(k-l)\times 2^m}$$

$$g(l) = \frac{1}{2^{(k-l)\times 2^m}} \times \left[\frac{1.(1 - \frac{1}{2^l})(1 - \frac{2}{2^l})\ldots(1 - \frac{2^m-1}{2^l})}{1.(1 - \frac{1}{2^k})(1 - \frac{2}{2^k})\ldots(1 - \frac{2^m-1}{2^k})}\right] \times 2^{(k-l)\times 2^m}$$

$$g(l) = \left[\frac{1.(1 - \frac{1}{2^l})(1 - \frac{2}{2^l})\ldots(1 - \frac{2^m-1}{2^l})}{1.(1 - \frac{1}{2^k})(1 - \frac{2}{2^k})\ldots(1 - \frac{2^m-1}{2^k})}\right] \qquad (1)$$

$$g(l) \approx \frac{\left(1 - \frac{(2^m-1).2^m}{2.2^l}\right)}{\left(1 - \frac{(2^m-1).2^m}{2.2^k}\right)}$$

In the above step, we use the approximation that:$(1 - x_1).(1 - x_2)\ldots(1 - x_n) \approx 1 - \sum_i^n x_i$ when $\forall i,\ 1 \leq i \leq n,\ x_i << 1$ under the assumption that $2^m << 2^l$ and $2^m << 2^k$.

Again, using an approximation that $\frac{(1-x)}{(1-y)} = (1 - x + y)$, when $x, y << 1$, and an assumption that $2^{2m} << 2^l$, $2^{2m} << 2^k$ we get:

$$g(l) \approx \left(1 - \frac{(2^m - 1).2^m}{2.2^l} + \frac{(2^m - 1).2^m}{2.2^k}\right)$$

$$g(l) \approx \left[1 - \frac{(2^m - 1).2^m}{2} \times \left(\frac{1}{2^l} - \frac{1}{2^k}\right)\right]$$

Assuming that $2^{2m} >> 2^m$, and $2^k >> 2^l$, we get:

$$g(l) \approx 1 - \frac{2^{2m}}{2^{(l+1)}} \approx 1 - 2^{(2m-l-1)}$$

Substituting this approximate value of $g(l)$ back in $f(l)$, we obtain:

$$f(l) \approx 1 - 2^{[(2m-l-1)\times\binom{k}{l}]}$$

Note that the exponent of the second term $[(2m - l - 1) \times \binom{k}{l}]$ is negative for $l > (2m - 1)$ and $\binom{k}{l}$ is a very large value when l is close to $k/2$. From the above expression, we can observe that $f(l)$ is close to 1 as $l > (2m - 1)$.

References

[1] J. Boyd. Using CAM in Today's High-Speed Networks. http://www.isdmag.com/design/cam/cam.html.

[2] M. Buddhikot, S. Suri, and M. Waldvogel. Space Decomposition Techniques for Fast Layer-4 Switching. In *Protocols for High Speed Networks*, Aug. 1999.

[3] M. Charikar, V. Guruswami, R. Kumar, S. Rajagopalan, and A. Sahai. Combinatorial Feature Selection Problems. In *Symposium on Foundations of Computer Science*, 2000.

[4] M. Dash and H. Liu. Feature Selection Methods for Classification: A Survey. Technical report, Department of Information Systems and Computer Science, National Institute of Singapore, 1999.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *ACM SIGCOMM*, Oct. 1997.

[6] A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. In *IEEE INFOCOM*, March 2000.

[7] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *ACM SIGCOMM*, Sept. 1999.

[8] P. Gupta and N. McKeown. Packet Classification Using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, Aug. 1999.

[9] P. Gupta and N. Mckeown. Algorithms for Packet Classification. *IEEE Network Magazine*, March/April 2001.

[10] D. Koller and M. Sahami. Toward Optimal Feature Selection. In *Machine Learning*, 1996.

[11] T.V. Lakshman and D. Stiliadis. High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *ACM SIGCOMM*, Sept. 1998.

[12] Netlogic Microsystems. http://www.netlogicmicro.com.

[13] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network Magazine*, March/April 2001.

[14] Sibercore Technologies. http://www.sibercore.com.

[15] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *ACM SIGCOMM*, Sept. 1999.

[16] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer four Switching. In *ACM SIGCOMM*, Sept. 1998.

[17] P. Tsuchiya. A Search Algorithm for Table Entries with Non-contiguous Wildcarding. unpublished report.

[18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, Oct. 1997.

[19] T.Y.C. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *IEEE INFOCOM*, March 2000.