Diagonal Tuple Space Search in Two Dimensions

Mikko Alutoin and Pertti Raatikainen

VTT Information Technology
P.O. Box 1202, FIN-02044
Finland
{mikko.alutoin, pertti.raatikainen}@vtt.fi

Abstract. Due to the evolution of the Internet and its services, the process of forwarding packets in routers is becoming more complex. In order to execute the sophisticated routing logic of modern firewalls, multidimensional packet classification is required. Unfortunately, the multidimensional packet classification algorithms are known to be either time or storage hungry in the general case. It has been anticipated that more feasible algorithms could be obtained for *conflict-free* classifiers. This paper proposes a novel two-dimensional packet classification algorithm applicable to the conflict-free classifiers. It derives from the well-known tuple space paradigm and it has the search cost of $O(\log w)$ and storage complexity of $O(n2^w \log w)$, where w is the width of the protocol fields given in bits and n is the number of rules in the classifier. This is remarkable because without the conflict-free constraint the search cost in the two-dimensional tuple space is $\Theta(w)$.

1 Introduction

Traditional packet forwarding in the Internet is based on one-dimensional route lookups: destination IP address is used as the key when the Forwarding Information Base (FIB) is searched for matching routes. The routes are stored in the FIB by using a network prefix as the key. A route matches a packet if its network prefix is a prefix of the packet's destination IP address. In the event that several routes match the packet, the one with the longest prefix prevails.

This well-know process does not inherently meet requirements of some of the new routing techniques. For example, in *firewalling, QoS based routing*, *programmable* and *active networking* [1] as well as in *application level routing* the forwarding decision is based on multiple protocol fields [2]. The forwarding is no longer based on just the destination IP address, but other attributes are considered as well. In firewalling, for example, the packet may be matched against a 5-tuple, composed of the source and destination IP address, source and destination port and the protocol field of the IP header. In application level routing, a URL can be used as an attribute when making the forwarding decision. In summary, all these new routing techniques require multidimensional packet classification [3, 4].

N. Mitrou et al. (Eds.): NETWORKING 2004, LNCS 3042, pp. 308–319, 2004.

When it comes to the methods of packet classification, linear search through the FIB is an option. In the linear search, all FIB entries are compared with the packet one by one in order to eliminate the non-matching entries. Among the matching entries, the one with the highest priority (e.g. the one with the longest network prefix in the case of one-dimensional packet classification) is the best matching entry. Unfortunately, the linear search is too time-consuming for backbone routers, which have to make tens of millions of forwarding decisions per second in order to keep up with the line speed. Thus packet classification algorithms are called for.

Efficient algorithms that facilitate wire-speed route look-ups have been developed for the problem of one-dimensional packet classification [5]. When the number of dimensions grows, so does the search complexity. A general k-dimensional ($k > 3$) packet classification algorithm has $O(log\ n)$ search complexity with $O(n^k)$ memory space or $O(log^{k-1} n)$ search complexity with $O(n)$ memory space, where n is the number of FIB entries [6]. This is impractical for a high-speed router. In [7], it has been suggested that more efficient packet classification algorithms could be developed for the conflict-free FIBs. To support this claim, a two-dimensional packet classification algorithm, which exploits the conflict-free constraint, has been provided. The algorithm is based on Tuple Space Search [8].

The work reported in [7] inspired us to study the subject more deeply. After a careful study, we came to the conclusion that the proposed algorithm does not work. However, by elaborating the ideas in [7] and by adding some new ones, it was possible to come up with an algorithm that makes use of the conflict-free constraint. The algorithm is shown to have the search cost of $O(log\ w)$ and the storage requirement of $O(n2^w\ log\ w)$. This is remarkable, because without the conflict-free constraint the number of search steps has been shown to be exactly 2w-1, i.e., $\Theta(w)$ [8].

Rest of the paper is organized as follows. Section 2 explains the concept of conflict-free constraint and section 3 describes the concept of tuple space. Section 4 introduces our contribution, the diagonal tuple space search in two dimensions, section 5 includes performance evaluation and section 6 concludes the paper.

2 Conflict-Free Constraint

Multiple FIB entries can match a packet at each look-up and thus some arbitration must be done to determine the best matching one. In the one-dimensional look-ups, the length of the prefix is used for this purpose. FIB entries with a longer network prefix get priority. In the multidimensional look-ups, the principles remain the same, i.e., the longer match gets priority. Nevertheless, it is not always that simple to determine which match is the longest one. From now on, a FIB entry is referred to as a *rule* - a commonly used term in packet classification [3]. Consider what happens if a FIB contains the following rules:

```
Rule1    From network a.b.c.* to network n.*.*.*
         DENY packets
```

```
Rule2     From network a.*.*.* to network n.b.c.*
          PERMIT packets
```

Let's suppose that a packet arrives from network a.b.c.* and its destination is in network n.b.c.*. Both rules match the packet, but which one is the longer match. It is impossible to say, because both are longer in one dimension but shorter in the other. So, the two rules are in conflict. Generally, two rules are in conflict when they *overlap* and neither one *encloses* the other [9]. Overlap means that all the prefixes of the two rules are non-disjoint. This is true in the example case, because a.b.c.* is a subset of a.*.*.* and n.b.c.* is a subset of n.*.*.*. Enclosure means that one of the rules is at least as specific as the other one in all dimensions. Clearly neither Rule1 or Rule2 encloses the other one.

Before a rule can be inserted into a FIB, all conflicts between the rule and the already inserted rules need to be detected [10] and resolved. There are two methods for resolving conflicts, i.e., *implicit conflict resolution* and *explicit conflict resolution* [9]. In the former case, the conflicting rules are assigned priorities that are used to arbitrate between the matching rules. In the explicit conflict resolution, a *resolving rule* is required for each conflict. A resolving rule specifies explicitly the action that prevails in the conflict region. Resolving rules are no different than the ordinary rules in the FIB, except that they are removed when either one of the conflicting rules is removed. A pseudo-code for computing prefixes of a resolving rule is given below. It is adapted from [9].

```
Function ResolvingRule(Ra,Rb)
          for i = 1 to k do
                    Rc[i]= Longer(Ra[i], Rb[i])
          end for
          return (Rc)
end Function
```

In the above example, the resolving rule would be:

```
Rule3     From network a.b.c.* to network n.b.c.*
          ACTION (= PERMIT packets or DENY packets)
```

Whenever a packet matches both Rule1 and Rule2, it will also match the resolving rule (Rule3). In such cases, the resolving rule is the best match, because it is always at least as specific as either one of the conflicting rules in every dimension. The action part of the resolving rule is decided by the entity that handles the conflict resolution.

The conflict-free constraint on the FIB means that there is a resolving rule for each conflicting rule pair in the FIB. This is a mandatory requirement for the algorithm which is put forward in this paper.

3 Tuple Space Search

The tuple space search [8] is a scheme proposed for multidimensional packet classification applications. Next, the basic features of the scheme and its major tools are explained.

3.1 Tuple Space Paradigm

In the tuple space search, rules are grouped based on their prefix length and a group is referred to as a *tuple* [8]. The groups are stored in hash tables and each group forms a separate hash table. In a k-dimensional FIB, the tuples are vectors of length k. For example, in a two-dimensional tuple space, rules R_1 = (100*, 11*) and R_2 = (001*, 01*) both map to tuple T_1 = [3, 2], while R_3 = (*, 110*) maps to tuple T_2 = [0, 3].

The key idea is that rules are hashed by using the concatenation of the prefix strings. When a packet is being classified, each tuple is probed for a matching rule. The concatenation of bits from the packet's header fields forms the hash key. The tuple vector indicates the number of bits taken in each dimension. Note that a probe results in finding either one or none matching tuple entries.

The search complexity of this packet classification method is proportional to m, the number of tuples. This is an improvement to the basic linear search through the FIB, for which the search cost is proportional to N, the number of rules in the FIB. However, the worst case bound is still O(N).

3.2 Markers and Pre-computation

Markers and pre-computation were introduced in [5] to carry out binary search for one-dimensional IP route look-ups. The FIB in [5] can be thought of as one-dimensional tuple space: the FIB entries are routes to networks and they are grouped to tuples by the length of the destination IP address prefixes. A hash key is generated for each tuple by taking as many most significant bits of the destination IP address as the hash table is wide. The basic linear search through the tuples has a search cost of O(w), w being the width of the destination IP address in bits. However, a much better bound O($logw$) can be obtained by using binary search for the hash table probes. The binary search can be applied by employing *markers* and *pre-computation*.

Markers are used to direct the binary search to look for matching routes with even longer network prefixes. The idea is that adding a route that has a network prefix of length l will result not only in insertion of the route in the hash table of width l but also in insertion of a marker in each hash table of width shorter than l. For example, addition of route that has prefix 1101 will produce markers 110, 11 and 1, which are inserted in hash tables of width 3, 2 and 1, respectively. Thus an entry in a tuple can be associated with one route and one marker. Note that routes whose network prefixes start with the same l-bit sequence share markers in the hash tables that have width $\leq l$.

When a hash table (width l) is probed during a binary search, the other hash tables can be divided into two groups: longer half (width > l) and shorter half (width ≤ l). If no matching marker is found, the longer half can be eliminated and the search focuses on the shorter half. This can be done because every matching route in the longer half would have left a matching marker in the probed hash table. If instead a matching marker is found, the binary search is directed to the longer half. Now, the shorter half cannot be dismissed straight away, because there is no guarantee that the longer half will eventually contain any matching routes. This situation is dealt with pre-computation.

The idea of pre-computation is that one can compute the best matching route in the shorter half for each marker beforehand and store it in the marker. In this way, one can dismiss the shorter half since the matching marker has already yielded the best matching route in that set. The algorithm must keep track of the current best matching route all along the search and update it each time a new matching marker is found. If the matching entry is associated with a route, but not with a marker, the search stops and that route is the best matching one.

To summarize, the markers and pre-computation can be used to trade off memory space and route/rule insertion time for faster look-up time.

3.3 Markers and Pre-computation in Multidimensional Tuple Space

To understand how markers and pre-computation work in the multidimensional tuple space, consider a tuple $T_i = [l_1, l_2, ..., l_k]$. The tuple space can be partitioned into three disjoint sets with respect to T_i, i.e., *Short*(T_i), *Long*(T_i) and *Incomparable*(T_i) [8]. Set *Short*(T_i) contains the tuples that are no longer than T_i in any dimension, i.e., tuple $T_j = [h_1, h_2, ..., h_k]$ belongs to set *Short*(T_i) if and only if $h_i \leq l_i$ ($1 \leq i \leq k$) and $T_j \neq T_i$. Similarly, tuple $T_j = [h_1, h_2, ..., h_k]$ belongs to set *Long*(T_i) if and only if $h_i \geq l_i$ ($1 \leq i \leq k$) and $T_j \neq T_i$. The rest of the tuple space belongs to set *Incomparable*(T_i). Note particularly that if two overlapping rules R_i and R_j map to tuples T_i and T_j, respectively, and if tuple T_j belongs to set *Incomparable*(T_i) then the two rules are in conflict.

Each rule that maps to tuple T_i can leave a marker in tuples in set *Short*(T_i). Markers in T_i in turn contain their best matching rule, obtained by pre-computation, in set *Short*(T_i). It follows that if tuple T_i is being probed and it does not contain a matching marker, set *Long*(T_i) can be dismissed and the search can be restricted to sets *Short*(T_i) and *Incomparable*(T_i). Let us call the union of these sets as *Fail*(T_i). If instead there is a matching marker in T_i then one can dismiss set *Short*(T_i) by pre-computation and restrict the search to sets *Long*(T_i) and *Incomparable*(T_i). Let's call the union of these sets as *Success*(T_i).

Can binary search work for k-dimensional tuple spaces? It turns out that it cannot, because set *Incomparable*(T_i) is included both in *Success*(T_i) and *Fail*(T_i). Due to this overlap, the binary search cannot work. In fact, it has been proved in [8] that the best case search cost for any algorithm, which performs a search in k-dimensional tuple space (k>2), is $\Omega(w^{k-1})$. A related result has been provided in [11], where it has been stated that by deploying markers and pre-computation the worst case search cost is

$O(w^{k-1} \log w)$. For the special case of two-dimensional tuple space, the search cost has been shown to be exactly $2w$-1, i.e., $\Theta(w)$ [8].

Despite these rather disturbing results, it has been suggested in [7] that by imposing the conflict-free constraint on the FIB faster search algorithms can be obtained. To prove their claim the authors put forward an algorithm for two-dimensional packet classification for the conflict-free FIBs [7]. After a careful study, we came to the conclusion that the proposed algorithm does not work. However, by refining some of the given ideas and adding new ones, we were able to come up with an algorithm that seems to work. This algorithm is named as the diagonal tuple space search in two dimensions and indeed, it has the search cost lower than $O(w)$.

4 Diagonal Tuple Space Search in Two Dimensions

The diagonal tuple space search algorithm uses markers in a new and innovative way, i.e., markers are inserted diagonally. The pseudo-code below describes the procedure of inserting the markers. This procedure is executed each time a new rule is added into the FIB.

```
Function SetMarkers(Rule R)
/* Tuple T is initially the tuple
 * to which the rule maps to */
Tuple T = [|R[1]|, |R[2]|];

/* One marker is inserted in each iteration */
while (T ! = [0, 0])
if (T[1] > T[2])
        T[1] = T[1] - 1;
else
        if (T[2] > T[1])
                T[2] = T[2] - 1;
        else /* T[1] equals T[2] */
                T[1] = T[1] - 1;
                T[2] = T[2] - 1;
InsertMarkerAtTuple(T, R);
end while
end Function
```

An example of a two-dimensional FIB is shown in Fig. 1. The arrows describe the way the rules place their markers. For example, if a rule maps to the shadowed tuple T_d = [2, 2] markers are inserted into tuples [1, 1] and [0, 0]. A rule mapping to tuple [0, 5] inserts markers to tuples [0, 4], [0, 3], [0, 2], [0, 1] and [0, 0]. Tuple [0, 0] is a virtual tuple which contains the default rule $R_{default}$ = (*,*). Fig. 1 also shows how the rest of the tuple space is divided into sets $Short(T_d)$, $Long(T_d)$ and $Incomparable(T_d)$

with respect to diagonal tuple T_d = [2, 2]. Definitions of the sets were given in the previous section.

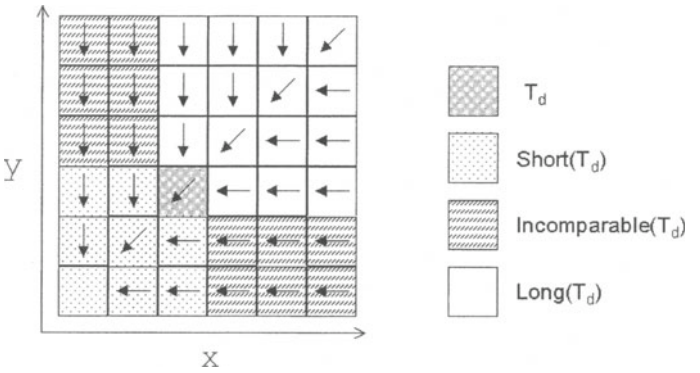

Fig. 1. Two-dimensional tuple space with respect to diagonal tuple [x, y]=[2, 2]

The algorithm starts with a binary search over the diagonal tuples in order to find the longest diagonal tuple T_d, which contains a matching entry. The following theorem states where the best matching rule resides with respect to tuple T_d.

Theorem 1

If the longest diagonal tuple, which contains a matching entry, is T_d = [d, d] then the best matching rule resides in set $Short(T_{d+1}) \cup Incomparable(T_{d+1})$, where T_{d+1} = [d+1, d+1].

Proof.

Any matching rule in set $Long(T_{d+1})$ places a matching marker in the diagonal tuple T_{d+1}. If T_d is the longest diagonal tuple, containing a matching entry then set $T_{d+1} \cup Long(T_{d+1})$ contains no matching rules. Since $T_{d+1} \cup Long(T_{d+1}) \cup Short(T_{d+1}) \cup Incomparable(T_{d+1})$ = 1, it follows that the best matching rule resides in union $Short(T_{d+1}) \cup Incomparable(T_{d+1})$.

Recall that set $Short(T_d)$ is covered by pre-computation and T_d has already been probed. Thus the search can be restricted even further. Excluding set $T_d \cup Short(T_d)$ yields search area $(Short(T_{d+1}) \cup Incomparable(T_{d+1})) \cap (Long(T_d) \cup Incomparable(T_d))$. This remaining search area consists of two rectangles (see Fig. 2). Later we will show that if the matching entry in T_d, let this entry be E_d, is not associated with a marker, but a rule only, then that rule is the best matching rule. For now, suppose that a marker is associated with E_d in step 1 and the algorithm continues to step 2.

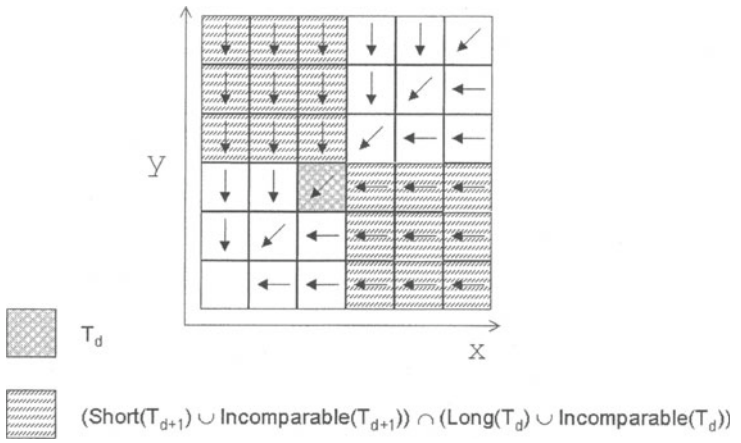

Fig. 2. Remaining search area after step 1 of the algorithm

In order to keep the search cost low, the algorithm uses a new technique that we call *mirroring*. Due to mirroring, only two additional binary searches are needed to conclude the packet classification. These two binary searches are performed on tuples $[d, y \geq d]$ and tuples $[x \geq d, d]$. This results in the search cost of O(*log w*), because three binary searches are enough to classify a packet. The basic need for mirroring is that it must be ensured that every matching rule in any tuple $[x < d, y > d]$ is represented among the tuples $[d, y > d]$ and that every matching rule in any tuple $[x > d, y < d]$ is represented among the tuples $[x > d, d]$. This representation is achieved by using *mirror rules*.

Mirror rules are called for, when a conflict arises between a rule and a diagonal marker (i.e. a marker that resides in a diagonal tuple). The mirror rules are updated each time the FIB is changed. The condition part of a mirror rule is computed in the same way as the condition part of a resolving rule (see the `ResolvingRule` procedure in section 2). The action part of a mirror rule is the same as that of the rule, which is used to produce it. For example, if a rule R_1 = (*, 110*) has a conflicting marker M = (00*, 11*) at tuple [2, 2], a mirror rule R_m = (00*, 110*) is produced in tuple [2, 3] to represent R_1 in column 2. If multiple rules produce a mirror in the same entry, the mirror that is produced by the longest rule prevails. For example, if there is another rule R_2 = (0*, 110*), which is also in conflict with marker M, then the action part of mirror rule R_m is that of R_2.

Let us concentrate on explaining how mirroring works in the upper rectangle. From now on, the term *original rule* is used to refer to the rules which are not mirror rules. In other words, an original rule is either an ordinary rule or a resolving rule.

Step 2 of the algorithm is to perform binary search on tuples $[d, y \geq d]$. Suppose step 2 returns a matching entry that is in tuple $T_u = [d, y_u]$, $T_u \neq T_d$. Now, tuples $[x \leq d, y > y_u]$ can be dismissed from the search, because all the matching rules in that part of the tuple space are either in column d or have a corresponding mirror rule in column

d. This is contradictory to the fact that $[d, y_u]$ was found to be the longest tuple in column d. Hence, the tuples $[x \leq d, y > y_u]$ do not contain a matching rule.

At this point, it is clear that if the upper rectangle contains a matching rule, it will be in set $T_u \cup Short(T_u)$. If T_u contains an original rule, this is clearly the best matching rule in the upper rectangle. If it does not contain the original rule but contains a mirror rule, the mirror rule is the best match in the upper rectangle. If T_u contains no rules but only a matching marker, pre-computation is used to determine whether the upper rectangle contains matching rules at all.

The next theorem shows that if a matching rule is found in the upper rectangle, it is the best matching rule in the whole tuple space.

Theorem 2

If tuple T_m contains a matching rule and set $Long(T_m)$ contains no matching rules then no tuple in set $Incomparable(T_m)$ contains a matching rule.

Proof.

If two matching rules R_m and R_n reside in pair-wise incomparable tuples T_m and T_n then, by the conflict-free constraint, there is a third matching rule R_r which belongs to set $Long(T_m) \cap Long(T_n)$. Now, if set $Long(T_m)$ contains no matching rules then set $Long(T_m) \cap Long(T_n)$ contains no matching rules either. This is contradictory to the assumption that both tuples contain a matching rule. Thus if tuple T_m contains a matching rule and set $Long(T_m)$ contains no matching rules then no tuple in set $Incomparable(T_m)$ contains a matching rule.

The algorithm as a whole is as follows.

Step 1: Perform binary search on the diagonal tuples in order to find the longest matching entry E_d among them. Let E_d reside in $T_d = [d, d]$. If E_d is associated with a marker proceed to step 2, otherwise return the rule which caused the match in T_d as the best matching rule.

Step 2: Perform binary search on tuples $[d, y \geq d]$ to find the longest matching entry E_u among them. Let E_u reside in tuple T_u.

Step 3: If $T_u \neq T_d$, go to step 4, else go to step 6.

Step 4: If E_u is not associated with any rule (but a marker only), go to step 5.1, else go to step 5.2.

Step 5.1: Find the best matching rule in set $Short(T_u)$ by pre-computation. If the rule is in rectangle $[x \leq d, y > d]$, return this rule, else go to step 6.

Step 5.2: If E_u is associated with an original rule, return that rule, else return the mirror rule.

Step 6: Perform binary search on tuples $[x \geq d, d]$ to find the longest matching entry E_l among them. Let E_l reside in tuple T_l.

Step 7: If $T_l \neq T_d$, go to step 8, else go to step 10.

Step 8: If E_l is not associated with any rule (but a marker only), go to step 9.1, else go to step 9.2.

Step 9.1: Return the best matching rule residing in set *Short*(T_l).

Step 9.2: If E_l is associated with an original rule, return that rule, else return the mirror rule.

Step 10: If T_d contains an original rule, return that rule, else return the best matching rule residing in set *Short*(T_d) (and which is found by pre-computation).

Since the two rectangles are pair-wise incomparable, the search continues to the lower rectangle only when no matching rule is found in the upper rectangle. Due to the symmetrical nature of the problem, there is no need to explain steps 6 to 9 in detail. Step 10 is reached only if neither of the rectangles contains a matching rule.

To conclude the explanation, recall that the proof of step 1 was partly postponed to a later stage. Since Theorem 2 is now available, it is relatively easy to finalize the proof. The claim was that if E_d in step 1 is not associated with a marker, but with a rule only, then that rule is the best matching rule. Now, T_d obviously contains a matching rule while set *Long*(T_d) contains no matching rules. Consequently, set *Incomparable*(T_d) is also dismissed by Theorem 2.

As a final remark, a short explanation is provided why the algorithm in [7] does not work. The reasoning in [7] is based on the assumption that can be formulated as follows: *"If a matching marker resides in tuple T_m and a matching rule R_n resides in tuple T_n and tuples T_m and T_n are pair-wise incomparable then, by the conflict-free constraint, there is a matching resolving rule R_r in set Long(T_m)"*. This theorem does not hold, because a matching marker in tuple T_m does not guarantee that there is a matching rule in set *Long*(T_m). Namely, it is possible for a rule to insert a matching marker even if the rule itself does not match. Our algorithm tackles this problem via mirroring.

5 Performance Evaluation

In this section, the search and storage complexities of the algorithm are evaluated. When it comes to other lookup algorithms [5, 7, 8, 11], which deploy hash tables, the search complexity/cost has been evaluated in terms of the asymptotic tight bound on the number of hash probes required to classify a packet. The storage complexity is generally evaluated by deriving asymptotic tight bound on the number of hash table entries needed to store the FIB and its associated data structures. These measures are used in the following analysis as well.

The search cost of the diagonal tuple space search in two dimensions is O(*log w*), because three binary searches at most are needed to classify a packet. This looks very good, recalling that without the conflict-free constraint the theoretical best bound is O(*w*). What can we say about the storage complexity? Recall that an original rule requires one mirror rule for each conflicting diagonal tuple. Within an incomparable diagonal tuple $T_d = [d, d]$ a rule that maps to tuple $[x < d, y > d]$ may have up to $2^{(d-x)}$

conflicting markers. At first glance, this yields the storage complexity of $O(n2^w w)$, where n is the number of rules in the FIB. However, the binary search on a column/row does not require that markers are created in all the tuples. It is enough to create them only in the tuples, which may be visited during the binary search [5]. Thus any rule leaves $log\ w$ markers at most. This gives the storage complexity of $O(n2^w\ log\ w)$. Table 1 contains a comparison between the search and storage complexities of packet classification algorithms, which are usable for two-dimensional FIBs.

Table 1. Comparison between two-dimensional packet classification algorithms

Algorithm	Search	Storage
Grid of tries	$O(w)$	$O(nw)$
Cross-producting	$O(log\ w)$	$O(n^2)$
Tuple space search	$O(w^2)$	$O(n)$
Rectangle search	$O(w)$	$O(nw)$
Diagonal tuple space search	$O(log\ w)$	$O(n2^w\ log\ w)$

The grid of tries and cross-producting have been described in [12]. Our algorithm exploits the conflict-free constraint in reducing the search cost dramatically, while the storage still remains linear with respect to the number of rules in the FIB. The downside is that the storage complexity grows drastically as the protocol fields get wider.

6 Conclusions

New routing techniques, such as firewalling and application level routing, require multidimensional packet classification in routers. Unfortunately, the general k-dimensional packet classification problem has been found to be either time or storage hungry. This fact has steered the research on packet classification algorithms towards hardware based as well as heuristic schemes. Nevertheless, it has recently been anticipated [7] that more efficient look-up algorithms could be achieved by imposing the conflict-free constraint on the Forwarding Information Base (FIB).

This paper proposes a novel search algorithm, named as the diagonal tuple space search in two dimensions, applicable to the conflict-free FIBs. The algorithm derives from the tuple space paradigm [8] and its search complexity is $O(log\ w)$. This is remarkable, because without the conflict-free constraint the number of search steps in a two-dimensional tuple space is known to be $2w$-1, i.e., $\Theta(w)$ [8].

The algorithm scales well with respect to the size of the FIB, because its storage complexity is $O(n2^w\ log\ w)$. Nonetheless, the worst case storage requirement grows drastically with respect to the width of the protocol fields. Yet, it has to be stated that the derived worst-case storage complexity is overly pessimistic and we believe that

for real-life FIBs the scalability of the algorithm would be clearly better. It is for further study to develop estimates for the practical storage requirement.

Characteristics of the developed search algorithm support the claim that in some cases the conflict-free constraint can be leveraged in finding more efficient packet classification algorithms. At present, the algorithm is applied in two dimensions and further work concentrates on analyzing the implications of the conflict-free constraint for the tuple space in three or more dimensions.

References

[1] Alutoin, M., Raatikainen, P.: Control Interface for Router Extension. Proceedings of 21st IASTED Conference on Applied Informatics (2003) 697-702

[2] Gupta, P., McKeown, N.: Packet Classification on Multiple Fields. Proceedings of ACM SIGCOMM'99, vol. 29, no. 4 (1999) 147-160

[3] Gupta, P., McKeown, N.: Algorithms for Packet Classification. IEEE Network, vol. 15, issue 2 (2001) 24-32

[4] Chao, H.J.: Next Generation Routers. Proceedings of the IEEE, vol. 90, no. 9 (2002) 1518 – 1558

[5] Waldvogel, M., Varghese, G., Turner, J., Plattner, B.: Scalable High Speed IP Routing Lookups. Proceedings of ACM SIGCOMM'97 (1997) 25-36

[6] Overmars, M.H., van der Stappen, A.F.: Range Searching and Point Location Among Fat Objects. Journal of Algorithms, 21(3) (1996) 629-656

[7] Warkhede, P., Suri, S., Varghese, G.: Fast Packet Classification for Two-Dimensional Conflict-Free Filters. Proceedings of 20th IEEE Infocom, vol. 3 (2001) 1434-1443

[8] Srinivasan, V., Suri, S., Varghese, G.: Packet Classification using Tuple Space Search. Proceedings of ACM SIGCOMM'99 (1999) 135-146

[9] Hari, A., Suri, S., Palkar, G.: Detecting and Resolving Packet Filter Conflicts. Proceedings of 19th IEEE Infocom, vol. 3 (2000) 1203-1212

[10] Baboescu, F., Varghese, G: Fast and Scalable Conflict Detection for Packet Classifiers, IEEE Computer Networks, vol. 42 (2003) 717-735

[11] Waldvogel, M.: Multi-dimensional Prefix Matching Using Line Search. Proceedings of IEEE Local Computer Networks (2000) 200-207

[12] Srinivasan, V., Varghese, G., Suri, S., M. Waldvogel: Fast and Scalable Layer Four Switching. Proceedings of ACM SIGCOMM'98 (1998) 191-202