IEICE TRANS. INF. & SYST., VOL.E95–D, NO.2 FEBRUARY 2012

PAPER Special Section on Architectures, Protocols, and Applications for the Future Internet

An Efficient Conflict Detection Algorithm for Packet Filters*

Chun-Liang LEE[†a)], ***Member***, **Guan-Yu LIN**[††], ***and*** **Yaw-Chung CHEN**[††], ***Nonmembers***

SUMMARY Packet classification is essential for supporting advanced network services such as firewalls, quality-of-service (QoS), virtual private networks (VPN), and policy-based routing. The rules that routers use to classify packets are called packet filters. If two or more filters overlap, a conflict occurs and leads to ambiguity in packet classification. This study proposes an algorithm that can efficiently detect and resolve filter conflicts using tuple based search. The time complexity of the proposed algorithm is $O(nW + s)$, and the space complexity is $O(nW)$, where n is the number of filters, W is the number of bits in a header field, and s is the number of conflicts. This study uses the synthetic filter databases generated by ClassBench to evaluate the proposed algorithm. Simulation results show that the proposed algorithm can achieve better performance than existing conflict detection algorithms both in time and space, particularly for databases with large numbers of conflicts.

key words: *packet classification, conflict detection, tuple space search*

1. Introduction

Packet classification is a key component of a variety of advanced network services, including security, policy-based routing, and quality-of-service (QoS). Packet filters are the rules that routers use to classify incoming packets into different flows. A filter $F = (f[1], f[2], \ldots, f[k])$ is called k-dimensional if the filter consists of k fields, where each field can be a variable length prefix, a range, an exact value or wildcard. The most common fields are the network source address, network destination address, source port, destination port, and protocol type. A packet P matches a particular filter F if for all i, the i-th field of the header satisfies $f[i]$. For example, a two-dimensional (2-D) filter $F = (163.25.*.*, *)$ denotes a flow originating from the subnet 163.25, and destined for any host. An IP packet $P1 = (163.25.114.1, 140.113.1.1)$ matches the filter F, while $P2 = (163.1.1.1, 140.113.1.1)$ does not.

Each filter has an associated action that specifies how to treat packets matching this filter. A packet may also match multiple filters. In this case, ambiguity may arise if the actions of matching filters conflict. For example, Table 1 gives a 2-D filter database with 4 filters. Suppose that the length of each field is 8 bits. A packet $P1 = (00110000, 00001111)$ matches filter A, and will be allowed to pass through if this is a firewall. However, another packet $P2 = (00110000, 00101111)$ matches both filters A and B leading to ambiguity. The conflicting actions of filters A and B cause a security problem if packets that should be blocked are allowed to pass through. Previous research lists three possible solutions to solve the problem of filter conflict [1]:

1. Select the first matching filter in the filter database.
2. Assign each filter a priority. The matching filter with the highest priority is selected.
3. Assign each field a priority. Select the matching filter with the most specific matching field with the highest priority.

However, none of these solutions can fully solve the ambiguity problem caused by filter conflict. Since a 2-D filter can be viewed as a rectangle in 2-D space, the filters in Table 1 can be represented as four rectangles (Fig. 1). A packet is a point in the space. Thus, overlap regions denote conflicts. For example, packets in the overlap between rectangles A and B match filters A and B. In this case, there are four overlap regions (i.e., conflicts) in the example filter database. The upper rectangle in a overlap denotes the desired filter to match in the overlap region. Therefore, if a packet matches both filters A and B, A is selected. Three solutions mentioned above cannot solve all conflicts in this

Table 1 An example filter database.

Filter	Field 1	Field 2	Action
A	001*	*	Accept
B	*	001*	Reject
C	110*	*	Accept
D	*	110*	Reject

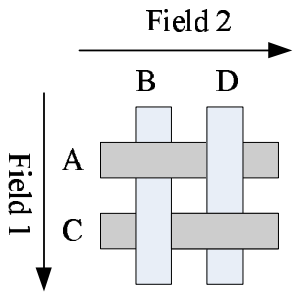


Fig. 1 Rectangular presentation of filters.

Manuscript received May 16, 2011.
Manuscript revised September 20, 2011.

[†]The author is with the Department of Computer Science and Information Engineering, Chang Gung University, Taoyuan, Taiwan.
[††]The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan.

*This work was supported in part by the National Science Council under Grant No.NSC 96-2221-E-182-013 and NSC 99-2221-E-182-053.

a) E-mail: cllee@mail.cgu.edu.tw

DOI: 10.1587/transinf.E95.D.472

filter database. For example, if we use the first solution, it is impossible to find an order of the filters such that the desired filters can be selected in all overlap regions. The second solution is similar to the first solution if we treat the priority assigned to a filter as the order of the filter in the first solution. Let $pri(A)$ denote the priority of filter A. Suppose that if filter A has a higher priority than filter B, then $pri(A) > pri(B)$. We have the following inequalities: $pri(A) > pri(B)$, $pri(B) > pri(C)$, $pri(C) > pri(D)$, and $pri(D) > pri(A)$. These inequalities lead to a contradiction. That is $pri(A) > pri(A)$. Similarly, the third solution cannot solve all conflicts.

A possible way to get around this difficulty is to use resolve filters [1]. A resolve filter is a filter that matches the packets in the overlap region of two conflicting filters. However, finding the minimum number of resolve filters is an NP-hard problem [1], [2]. Therefore, it is a challenging task to design an algorithm that can efficiently detect and resolve filter conflicts. This study proposes an efficient algorithm to detect filter conflicts using tuple space search [3]. The key idea behind the proposed algorithm is based on the intrinsic properties of the data structures generated by the rectangle research [3], a well-known packet classification algorithm. By exploiting the unrevealed relationship between the tuple space and filter conflicts, the proposed algorithm is not only faster than the existing conflict detection algorithms, but also requires less memory space.

The rest of this paper is organized as follows. Section 2 reviews existing conflict detection algorithms and describes the tuple space search, which is closely related to the proposed algorithm. Section 3 presents the proposed algorithm, and provides a simple example to show how it works. Section 4 presents the experimental setup and results. Finally, Sect. 5 concludes the paper.

2. Related Work

2.1 Existing Conflict Detection Algorithms

A straightforward approach to detect all filter conflicts is to check every pair of filters in the filter database. This approach is simple and does not require extra storage. However, it takes $O(n^2)$ time to detect all conflicts, where n is the number of filters. Obviously, this approach is not feasible for large filter databases. Hari et al. [1] proposed the use of a grid of tries [4] to detect all conflicts in 2-D prefix filters. The time complexity of their algorithm is $O(nW + s)$, where W is the length of the longest prefix and s is the number of pairs of conflicting filters. Baboescu and Varghese [5] proposed several conflict detection algorithms based on the bit vector scheme [6] and the aggregated bit vector scheme [7]. Among their algorithms, the best one runs in $O(n^2)$ time and requires $O(n^2)$ space. Lu and Sahni [2] proposed a plane-sweep algorithm that improves the performance of both time and space. The key idea behind the plane-sweep algorithm is to treat each filter as a rectangle with four line segments in the space, and then find all overlap regions (i.e., conflicts) by finding orthogonal line segment intersections. The plane-sweep algorithm runs in $O(n\log n + s)$ time and requires $O(n)$ space.

2.2 Tuple Space Search

The basic idea behind the tuple space search is that the number of distinct prefix lengths of filters is much less than the number of filters [3]. Given a filter database, a *tuple* is defined for each combination of field length, and the resulting set is called *tuple space*. For example, assume that we have a 2-D filter database, in which each filter has two fields, specifying the source and the destination address, respectively. We say that a filter F belongs to tuple $T_{i,j}$ if the prefix length of its first field is i, and that of its second field is j. Since each tuple has a specific length for each field, these bit strings can be concatenated to form a hash key that can be used to perform the tuple lookup. For an incoming packet, the matching filters can be found by probing all tuples. For example, filter $F = (11*, 110*)$ belongs to tuple $T_{2,3}$, while $G = (110*, 0011*)$ belongs to tuple $T_{3,4}$. Given a packet $P = (11100000, 11011111)$, to probe tuple $T_{2,3}$, two and three bits will be extracted from two fields respectively to construct the hash key (i.e., 11110). Similarly, the hash key 1111101 will be constructed for tuple $T_{3,4}$. Even a linear search in the tuple space represents a considerable improvement over a linear search of the filters.

To improve the search time, previous research proposes a tuple-based algorithm called rectangle search [3]. The use of markers and pre-computation, which were first introduced in [8], greatly reduces the number of tuples to be probed. A *marker* is a special type of filter generated by a real filter to eliminate a subset of the tuple space after probing a tuple. Let filter $F = (f[1], f[2])$ with the length combination (i, j) belong to tuple $T_{i,j}$. For each tuple left to $T_{i,j}$, say $T_{i,j'}$, $0 \le j' < j$, a marker will be generated by eliminating the bits behind the j'th bits. After inserting all filters and generating required markers, pre-computation is performed for entries at each tuple, say $T_{i,j}$, to find the best matching filter (i.e., the least-cost filter) among the tuples above $T_{i,j}$, say $T_{i',j}$, $0 \le i' < i$. Figure 2 shows how markers are generated using two filters F and G described above. If we use the packet (11100000, 11011111) to probe tuple $T_{4,0}$, we will fail to get a match, which means it is impossible to find a matching filter in the right side of tuple $T_{4,0}$ in the same row. Thus, we can skip probing these tuples.

Since the rectangle search provides a feasible way for packet classification, researchers have proposed a number of related algorithms to further improve its performance both in time and space. Warkheda et al. [9] improved the rectangle research by exploiting the conflict-free constraint in the 2-D tuple space and introduced a binary search algorithm. In [10], Wang et al. proposed an algorithm to reduce the number of tuples to be probed. They adopted a dynamic programming scheme to calculate the optimal set of tuples and reorganize the rules. However, the cost of precomputation increases exponentially as the classifier expands, mak-

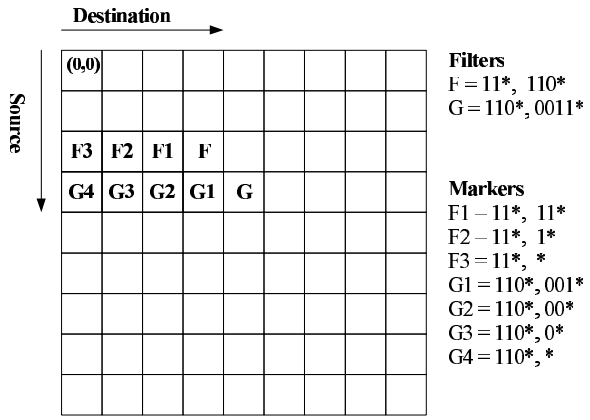


Fig. 2 Illustration of markers.

ing the scheme unsuitable for large filter databases. In [11], Wang et al. proposed a new hybrid scheme "filter rephrasing" to improve both the search and storage complexity of the rectangle search by reorganizing the entries in the hash tables.

3. Tuple-based Conflict Detection Algorithm

This section presents the proposed tuple-based conflict detection algorithm (TCDA). After describing the key idea behind the proposed algorithm, this section shows how to construct the required data structures. Then process of detecting all conflicts is also discussed. Finally, this section provides a simple example to help the reader understand how the TCDA operates.

3.1 Key Idea

For a 2-D filter database, if the length of both field is W bits, each filter is mapped to one of $(W+1)^2$ tuples. The simplest way to detect all conflicts is to look up all tuples for each filter. Therefore, the number of tuples required to lookup for a specific filter is simply $(W+1)^2-1$, which is obviously not feasible. To improve performance, we need more insights into the tuple space. Two filters F and G are in conflict if and only if all of the following three conditions hold [2]:

1. There is at least one packet that is matched by both F and G.
2. There is at least one packet that is matched by F but not by G.
3. There is at least one packet that is matched by G but not by F.

For a filter F in tuple $T_{i,j}$, the remaining tuples can be partitioned into three disjoint sets [3]:

1. Shorter tuples: A tuple $T_{i',j'}$, where $T_{i',j'} \neq T_{i,j}$, belongs to this set if and only if $i' \leq i$ and $j' \leq j$.
2. Longer tuples: A tuple $T_{i',j'}$, where $T_{i',j'} \neq T_{i,j}$, belongs to this set if and only if $i' \geq i$ and $j' \geq j$.
3. Incomparable tuples: A tuple $T_{i',j'}$, where $T_{i',j'} \neq T_{i,j}$, belongs to this set if and only if it belongs to neither the shorter tuples nor the longer tuples .

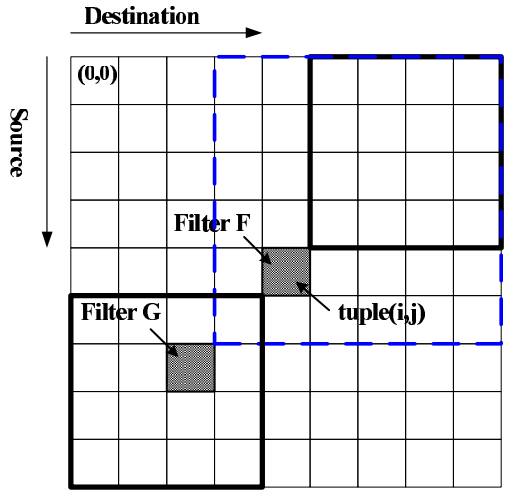


Fig. 3 The incomparable tuples of tuple $T_{i,j}$.

Among these three sets, only the *incomparable tuples* may contain the filters that are in conflict with filter F. The lengths of filter fields in the shorter tuples are smaller than filter F. Thus, filters in the shorter tuples are less specific than filter F, and cannot satisfy the three conditions mentioned above. Similarly, filters in the longer tuples are more specific than filter F. The three conditions do not hold as well. Figure 3 marks incomparable tuples with solid-line rectangles. The number of tuples to look up is reduced by nearly a half. Moreover, since we only need to generate one resolve filter for two conflicting filters, the bottom-left solid-line rectangle can be omitted. Let us use an example to explain this. Suppose that we have a filter G that belongs to a tuple in the bottom-left solid-line rectangle and is in conflict with filter F. Note that we have to process every filter. When filter G is processed, a resolve filter for filters F and G will be generated since filter F belongs to a tuple in the upper-right incomparable tuples of filter G, as marked by a dashed-line rectangle in Fig. 3. Therefore, the bottom-left solid-line rectangle can be omitted without missing any resolve filters.

The markers mentioned in Sect. 2 can further reduce the number of tuples to look up. Figure 5 (a) shows two filters A and B, and the generated markers, which are indicated by light gray color. Through the help of markers, the tuples to look up are the left-most column of the upper-right marked region (Fig. 4). In other words, for each filter F in tuple $T_{i,j}$, it is only necessary to look up i tuples (i.e., $T_{0,j+1}, T_{1,j+1}, \ldots, T_{i-1,j+1}$). The number of tuples to look up has been reduced from $O(W^2)$ to $O(W)$. For example, if we are dealing with a filter F in tuple $T_{i,j}$, and a match is returned when probing some tuple, say $T_{1,j+1}$, the marker rule tells us that there must be at least one filter conflicting with F in the right side of $T_{1,j+1}$ in the same row. The tuple-based algorithms mentioned in Sect. 2 provide no information that can be used to find all the conflicting filters efficiently. The

following subsection shows how to solve this problem by adding new fields in markers and filters.

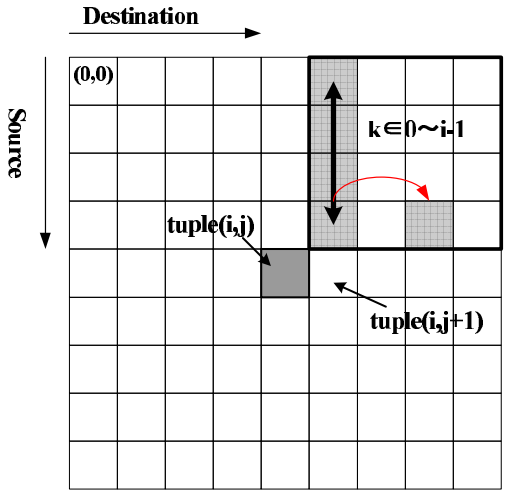


Fig. 4 Reducing the number of tuples needed to lookup.

3.2 Constructing the Required Data Structures and Performing Conflict Detection

We have shown the key idea behind the proposed algorithm. However, the information included in markers is not enough to detect and resolve all conflicts efficiently. If filter F finds a matching marker in tuple $T_{i-1,j+1}$, then filter F conflicts with one or more filters in tuples to the right of tuple $T_{i-1,j+1}$. To resolve these conflicts, we need to find the filters that generate the marker. To address this issue, we introduce the *marker pointer*, which is a field of a marker. The marker pointer of a marker points to the filter that generated it. The settings of marker pointer can be categorized into three types:

1. The first type is simplest, as Fig. 5 (a) shows. During the process of generating markers, neither filter encounters any duplicate markers or filters.
2. Figure 5 (b) shows the second type, where filter A encounters a duplicate filter (i.e., filter B). In this case,

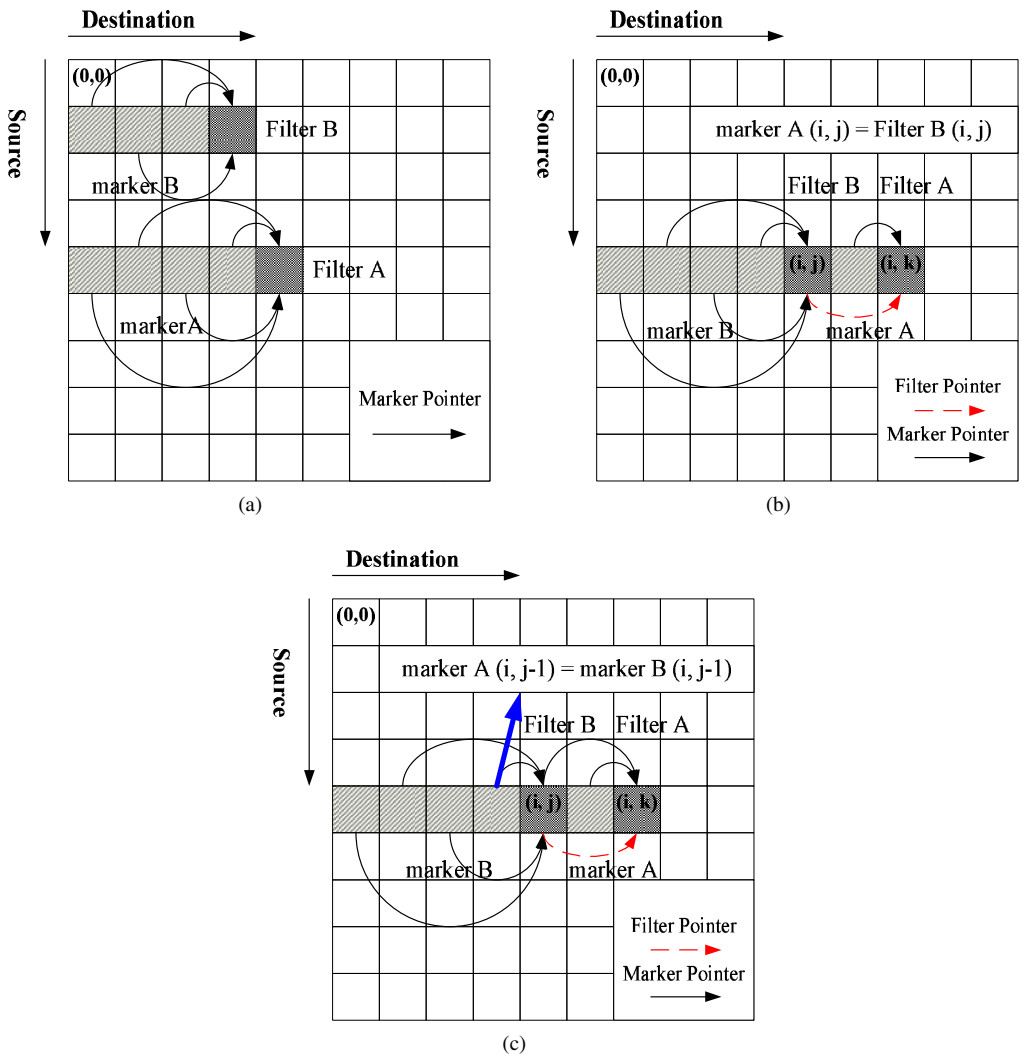


Fig. 5 Relationships between two filters.

the *filter pointer* of filter B, which indicates that there is a more specific filter in the right tuple, will point to filter A. The filter pointer is also a newly introduced field similar to the marker pointer while it is only for filters. Since the marker generated by filter A is identical to filter B in tuple $T_{i,j}$, filter A will stop generating markers in tuple $T_{i,j'}$, $0 \leq j' < j$.

3. The third type is more complicated (Fig. 5 (c)). The marker generated by filter A in tuple $T_{i,j-1}$ is duplicated with the marker generated by filter B. In this case, the marker generated by filter A in tuple $T_{i,j}$ still points to filter A. In addition, the filter pointer of filter B points to filter A because if a filter finds a matching marker in tuple $T_{i,j-1}$, both filter A and B that generate the same marker may conflict with it. Thus, the filter pointers can detect the conflicts more efficiently.

After describing how to set a marker pointer, we can now present the procedure of constructing the required data structures. First, every filter is inserted into the corresponding tuple according to its prefix lengths of fields. Then all tuples are processed from left to right and from top to bottom. Note that we can omit the tuples in the left-most column since they do not have tuples left to them. In other words, these tuples do not generate any markers. For each filter in a tuple, markers are generated according to the way we mentioned in Sect. 2.2, and the marker pointers are set as described above. The detailed procedure is shown in Algorithm 1. Table 2 summarizes the notation used in Algorithms 1 and 2.

To detect and resolve all conflicts, tuples are processed from left to right and from top to bottom. Since the tuples in the first row do not have the upper-right incomparable tuples, the conflict detection starts from the second row. Similarly, the tuples in the right-most column do not have the upper-right incomparable tuples, and thus can be omitted. In other words, the sequence of tuples to probe is $T_{1,0}, T_{1,1}, \ldots, T_{1,W-1}, T_{2,0}, T_{2,1}, \ldots, T_{2,W-1}, \ldots, T_{W,W-1}$. For each filter F in tuple $T_{i,j}$, tuples $T_{0,j+1}, T_{1,j+1}, \ldots, T_{i-1,j+1}$ should be probed. Note that for each of these tuples to probe, filter F has to generate two filters since the prefix length of the second field of filter F is j, while that of the tuple to probe is $j+1$. For example, suppose that we have a filter $F = (11*, 110*)$ in tuple $T_{2,3}$. The sequence of tuples to be processed is thus $T_{0,4}$ and $T_{1,4}$. To probe $T_{0,4}$, two filters, say $M = (*, 1100*)$ and $N = (*, 1101*)$, are generated to detect conflicts. For either filter M or N, if there is a match in the tuple, which means a conflict is detected, then the resolve filters will be generated by following the marker pointer to find the filter or the filter pointer for the chained filters. The detailed procedure for conflict detection is shown in Algorithm 2.

3.3 A Simple Example of the TCDA

The end of this section uses a simple example to illustrate how the proposed algorithm constructs the required data

Algorithm 1: Required data structures construction

```
1  foreach filter F in the filter database do
2      i ← Length(F[1]);
3      j ← Length(F[2]);
4      tuple T_{i,j} += F /* insert F to T_{i,j} */;
5  end
6  for i ← 0 to W do
7      for j ← 1 to W do
8          foreach filter A in tuple T_{i,j} do
9              for k ← j − 1 to 0 do
10                 A_m ← Generate a marker in tuple T_{i,k} using
                   filter A;
11                 if A_m finds a matching filter B in tuple T_{i,k}
                   then                                  /* Type 2 */
12                     B.pointer ← A;
13                     break;
14                 else if A_m finds a matching marker B_m in
                   tuple T_{i,k} then                    /* Type 3 */
15                     B_m.pointer.pointer ← A;
16                     break;
17                 else                                  /* Type 1 */
18                     A_m.pointer ← A;
19                     tuple T_{i,k} += A_m;
20                 end
21             end
22         end
23     end
24 end
```

Table 2 Summary of notation.

F	filter (same as other capital letters)
F_m	the marker generated by filter F
$F[i]$	the i-th field of filter F
$Length(F[i])$	the prefix length of $F[i]$
W	the length of the longest prefix
$F.pointer$	the filter pointer of filter F
$F_m.pointer$	the marker pointer of marker F_m

Algorithm 2: Conflict detection

```
1  for i ← 1 to W do
2      for j ← 0 to W − 1 do
3          foreach filter A in tuple T_{i,j} do
4              for k ← 0 to i − 1 do
5                  Generate filters M and N in tuple T_{k,j+1}
                   using A;
6                  if M or N finds a matching marker B_m in
                   tuple tuple T_{k,j+1} then
7                      C ← B_m.pointer;
8                  else if M or N finds a matching filter B in
                   tuple tuple T_{k,j+1} then
9                      C ← B;
10                 else
11                     C ← null;
12                 end
13                 if C ≠ null then
14                     Generate a resolve filter for A and C;
15                     Follow the filter pointer of filter C to
                       find all filters that are in conflict with
                       filter A and generate resolve filters;
16                 end
17             end
18         end
19     end
20 end
```

structures and uses them to detect all conflicts. Table 3 provides an example filter database with 10 filters. To ease the discussion, we assume that the length of each field is 4 bits. Thus the tuple space is a 5×5 square. First, each filter is inserted to the corresponding tuple, as Fig. 6 (a) shows. Then, markers are generated for each filter in exactly the same way as the rectangle search algorithm mentioned in Sect. 2.2. Recall that all filters are processed tuple-by-tuple from left to right and from top to bottom. Figure 6 (b) indicates each marker by a leading 'M', followed by the number of the corresponding filter that generates it. For example, the four markers indicated by $M6$ are generated by filter $F6$. The newly introduced marker pointer of each $M6$ points to filter $F6$. For clarity, this figure does not show all marker pointers. When filter $F4$ in tuple $T_{1,3}$ is processed, a marker indicated by $M4$ will be generated and inserted to tuple $T_{1,2}$. However, the marker to be generated in tuple $T_{1,1}$ is duplicated with filter $F3$. Therefore, $F4$ stops generating markers. In addition, the filter pointer of $F3$ points to $F4$, as indicated by an arrow in the figure. Similarly, the filter pointer of $F7$ points to $F9$. After generating all required markers

Table 3 A filter database with 10 filters.

Filter	Src Field	Dst Field
F1	*	1*
F2	1 *	*
F3	0 *	1*
F4	0 *	110*
F5	01 *	*
F6	00 *	0011
F7	000 *	1*
F8	010 *	0*
F9	000*	110*
F10	1101	011*

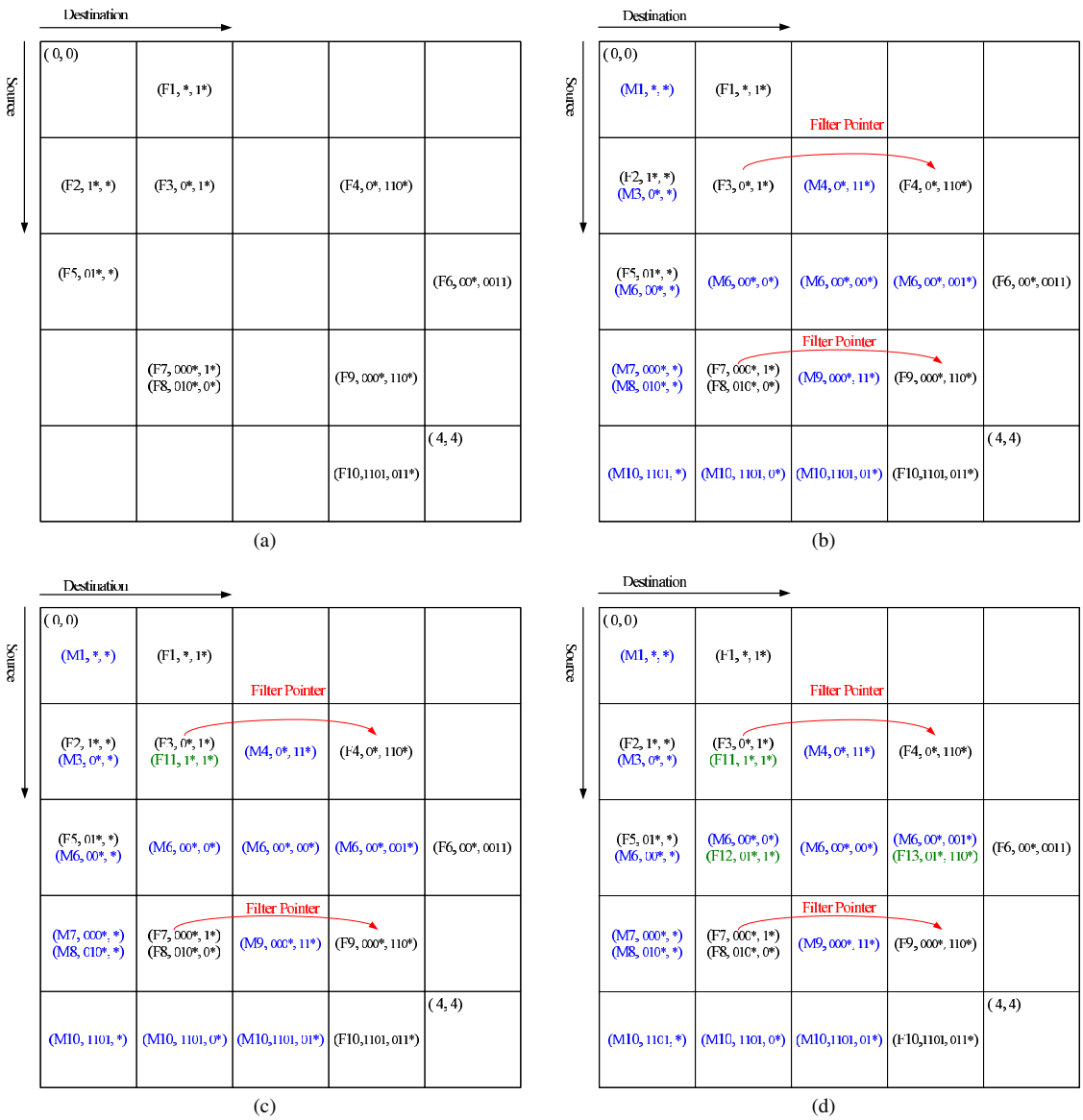


Fig. 6 Illustration of TCDA.

and setting marker pointers and filter pointers (Fig. 6 (b)), it is possible to start conflict detection.

The first tuple to be processed is tuple $T_{1,0}$, which contains only one filter (i.e., $F2$). To detect possible conflicts in tuple $T_{1,0}$, two filters (*, 0*) and (*, 1*) are generated using $F2$ to probe tuple $T_{0,1}$. Clearly, there is a match in tuple $T_{0,1}$. Since we have matched $F1$, which means $F1$ and $F2$ are in conflict, a resolve filter $F11$ is generated in tuple $T_{1,1}$, as Fig. 6 (c) shows. Similarly, when tuple $T_{2,0}$ is processed, two filters (0*, 0*) and (0*, 1*) are generated using $F5$ to probe tuple $T_{1,1}$. Again, there is a match, and a resolve filter $F12$ is generated in tuple $T_{2,1}$. Note that since the filter pointer of the conflicting filter $F3$ points to another filter, a resolve filter $F13$ will be generated in $T_{2,3}$ by following the filter pointer. This makes it possible to detect all conflicts and generate the required resolve filters, as Fig. 6 (d) shows. Clearly, the proposed TCDA takes $O(nW + s)$ time to report all conflicts, where n is the number of filters, W is the number of bits in a header field, and s is the number of conflicts. Note that the time complexity of $O(nW + s)$ is obtained by assuming that n is larger than W, which is a practical assumption. Otherwise, the time complexity is $O(W^2 + s)$.

4. Experimental Results

This section compares the proposed TCDA with the FastDetect algorithm [1] and the plane-sweep conflict detection algorithm [2]. All algorithms were implemented in C++, and benchmarked on a 3.2 GHz AMD Phenom II X4 PC with 4 GB of memory. The tested filter databases were synthesized by ClasssBench [12]. This tool includes 12 seed files derived from real filter sets to reflect the characteristics of filters for different applications, including three categories: the access control list (ACL), firewall (FW), and IP chain (IPC). For each seed file, we generate filter sets with 1 K, 5 K, 10 K, 20 K, and 30 K filters. Since the filters generated by ClassBench are 5-D, the number of filters for each file generated is slightly less than the specified number after removing duplicate filters by only considering source and destination address fields. The reported memory requirement does not include the memory required to store the original filter set. For the proposed TCDA, the memory space used to implement the hash function is included.

Tables 4 and 5 show the time requirements for different conflict detection algorithms to report all conflicts. Due to

Table 4 Time (in ms) to detect all conflicts for the seed files with the least conflicts in each category.

	ACL5			FW4			IPC1		
Filter Size	FastDetect	Plane-Sweep	TCDA	FastDetect	Plane-Sweep	TCDA	FastDetect	Plane-Sweep	TCDA
1 K	3.558	1.077	0.488	18.077	3.548	0.479	11.185	2.078	0.588
5 K	22.037	6.182	3.811	1,993.474	161.503	9.085	77.029	26.482	8.996
10 K	41.408	13.232	7.023	13,367.232	704.740	42.688	214.842	68.590	13.615
20 K	97.346	26.020	22.908	52,900.648	4,205.391	162.603	1,243.933	377.048	38.745
30 K	159.624	45.385	36.495	106,414.782	9,833.510	412.080	4,232.259	1,162.827	75.924

Table 5 Time (in ms) to detect all conflicts for the seed files with the most conflicts in each category.

	ACL3			FW5			IPC2		
Filter Size	FastDetect	Plane-Sweep	TCDA	FastDetect	Plane-Sweep	TCDA	FastDetect	Plane-Sweep	TCDA
1 K	12.771	1.843	0.888	87.943	4.953	0.274	21.015	3.627	0.517
5 K	81.904	19.321	5.304	8,936.516	199.417	20.895	2,072.464	158.314	4.357
10 K	194.428	44.391	13.562	55,865.839	932.029	94.472	13,415.186	613.596	16.198
20 K	1,022.153	199.690	32.018	222,450.552	5,069.460	462.182	50,888.207	3,965.395	86.769
30 K	4,009.130	826.100	72.894	498,589.323	12,291.351	1,342.515	112,247.022	9,514.543	215.908

Table 6 Performance evaluation for synthetic databases.

Databases	Statistics		FastDetect		Plane Sweep		TCDA	
	Number of Filters	Number of Conflicts	Time (ms)	Space (KB)	Time (ms)	Space (KB)	Time (ms)	Space (KB)
ACL1	29,809	8,960	420.927	27,497	57.064	3,100	16.612	3,597
ACL2	29,357	4,006,275	2,786.980	14,509	1,002.700	3,053	87.319	2,297
ACL3	29,367	4,884,375	4,009.130	12,478	826.100	3,054	72.894	2,422
ACL4	29,113	1,993,007	1,522.382	10,247	486.831	3,028	48.659	2,033
ACL5	29,522	152	159.624	6,622	45.385	3,070	36.495	1,340
FW1	29,378	89,599,625	390,297.176	12,426	11,975.669	3,055	1,431.130	1,869
FW2	29,279	54,492,459	68,409.287	10,428	2,802.817	3,045	352.340	1,042
FW3	29,196	98,774,337	411,709.041	10,561	12,656.453	3,036	1,490.969	1,676
FW4	29,128	49,802,186	106,414.782	12,765	9,833.510	3,029	412.080	1,818
FW5	29,470	115,748,863	498,589.323	11,151	12,291.351	3,065	1,342.515	1,576
IPC1	28,780	5,068,689	4,232.259	15,026	1,162.827	2,993	75.924	2,575
IPC2	30,000	27,247,914	112,247.022	17,795	9,514.543	3,120	215.908	2,637

the space limitation, Tables 4 and 5 only list the results of the seed files with the least and the most conflicts in each category, respectively. The proposed TCDA provides a significant speed improvement over the plant-sweep algorithm, not mention to the FastDetect algorithm, which is the slowest of these three algorithms. The performance improvement increases with the number of filters and the number of conflicts. For example, ACL3 has more conflicts than ACL5 for each filter size, and the performance improvement achieved by the TCDA for ACL3 is larger than that for ACL5.

Table 6 lists the memory and time requirements for all seed files, and the filter size of each database is 30 K. Compared to the other two algorithms, the proposed TCDA achieves a significant speed improvement. Since the time and space requirements of the FastDetect algorithm are apparently higher than the other two algorithms, the following discussion focuses on the plane-sweep algorithm and the TCDA. The smallest improvement is for ACL5. The time required to detect and resolve all conflicts is reduced by 19.6%. The largest improvement is for IPC2. The required time is reduced by 97.7%. Note that the number of conflicts of ACL5 is 152, which is also the smallest among all seed files. As for IPC2, the number of conflicts is over 27 M, which is not the largest, but is obviously a large number. In short, the proposed algorithm can reduce the times by over 70% for 11 out of 12 seed files. As for the memory requirements, the proposed TCDA still outperforms the other two algorithms except for ACL1. This is due to the rule used to generate markers in [3]. As the number of marker increases, the memory required by the proposed algorithm also increases. Since the proposed algorithm can reduce the time by 70.1% for ACL1, the extra cost in memory is acceptable.

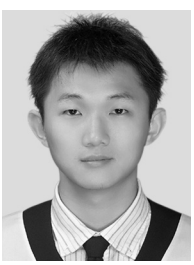
5. Conclusion

Packet classification plays an important role in providing advanced services in routers. Filter conflicts may lead to ambiguity in packet classification. This study proposes a tuple-based conflict detection algorithm to efficiently detect and resolve all conflicts. As compared with the plane-sweep algorithm, experimental results show that the proposed algorithm can reduce the detection time by 19.6% to 97.7%. More importantly, it reduces storage requirements for most filter databases. The performance improvement in time is particularly significant for filter databases with many conflicts.

References

[1] A. Hari, S. Suri, and G. Parulkar, "Detecting and resolving packet filter conflicts," Proc. IEEE INFOCOM, pp.1203–1212, March 2000.
[2] H. Lu and S. Sahni, "Conflict detection and resolution in two dimensional prefix router tables," IEEE/ACM Trans. Netw., vol.13, no.6, pp.1353–1363, 2005.
[3] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," Proc. ACM SIGCOMM, pp.135–146, Sept. 1999.
[4] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," Proc. ACM SIGCOMM, pp.191–202, 1998.
[5] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifier," Comput. Netw., vol.42, no.6, pp.717–735, Aug. 2003.
[6] T.V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," Proc. ACM SIGCOMM, pp.203–214, Aug. 1998.
[7] F. Baboescu and G. Varghese, "Scalable packet classification," Proc. ACM SIGCOMM, pp.199–210, Aug. 2001.
[8] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed ip routing lookups," Proc. ACM SIGCOMM, pp.25–36, Sept. 1997.
[9] P. Warkhede, S. Suri, and G. Varghese, "Fast packet classification for two-dimensional conflict-free filters," Proc. IEEE INFOCOM, pp.1434–1443, March 2001.
[10] P.C. Wang, C.T. Chan, S.C. Hu, C.L. Lee, and W.C. Tseng, "High-speed packet classification for differentiated services in next-generation networks," IEEE Trans. Multimedia, vol.6, no.6, pp.925–935, 2004.
[11] P.C. Wang, C.L. Lee, C.T. Chan, and H.Y. Chang, "Performance improvement of two-dimensional packet classification by filter rephrasing," IEEE/ACM Trans. Netw., vol.15, no.4, pp.906–917, 2007.
[12] D.E. Taylor and J.S. Turner, "Classbench: a packet classification benchmark," IEEE/ACM Trans. Netw., vol.15, no.3, pp.499–511, 2007.

Chun-Liang Lee received M.S. and Ph.D. degrees in computer science and information engineering from National Chiao Tung University, Hsinchu, Taiwan in 1997 and 2001, respectively. From 2002 to 2006, he worked with the Telecommunication Laboratories, Chunghwa Telecom Co., Ltd. Since February 2006, he has been an assistant professor of Computer Science and Information Engineering at Chang Gung University, Taoyuan, Taiwan. His research interests include the design and analysis of network protocols, quality of service in the Internet, and packet classification algorithms.



Guan-Yu Lin received an M.S. degree in computer science and information engineering from Chang Gung University in 2009. He is currently pursuing a Ph.D. degree in computer science and information engineering at National Chiao Tung University. His research interests include packet classification algorithms and peer-to-peer overlay networks.

Yaw-Chung Chen received his B.S. degree from National Chiao Tung University, Hsinchu, Taiwan, his M.S. degree from Texas A&M University, Kingsville, Texas, USA, and his Ph.D. degree from Northwestern University, Evanston, Illinois, USA. In 1986 he joined AT&T Bell Laboratories, where he worked on various exploratory projects. He joined National Chiao Tung University, Hsinchu, Taiwan as an associate professor in 1990. He is currently a professor in the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. His research interests include wireless networks, mobility management, P2P systems and green computing. He is a senior member of IEEE and a member of ACM.