Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Scalable packet classification with controlled cross-producting

Pi-Chung Wang *,1

Institute of Networking and Multimedia, National Chung Hsing University, Taichung 402, Taiwan, ROC
Department of Computer Science and Engineering, National Chung Hsing University, Taichung 402, Taiwan, ROC

ARTICLE INFO

Article history:
Available online 13 December 2008

Keywords:
Traffic classification
Packet classification
Firewalls
QoS
Forwarding

ABSTRACT

Packet classification is central among traffic classification techniques that categorize packets with a traffic descriptor or with user-defined criteria. This categorization may make information accessible for quality of service or security handling on the network. To make packet classification both fast and scalable, we propose a new algorithm that combines *cross-producting* with *linear search*. The new algorithm, *Controlled Cross-producting*, could improve the scalability of cross-producting significantly with respect to storage, while maintaining the search latency. In addition, we introduce several refinements and procedures for incremental update. We evaluate the performance of our scheme with filter databases of varying sizes and characteristics. Specifically, we experimented with 12 different types of filter databases, whose sizes vary from 16 K to 128 K. The experimental results demonstrate the feasibility and scalability of our scheme. A comparison with the prominent existing schemes further indicates that the proposed scheme takes less time and space.

1. Introduction

Traffic classification is an enabling technique for various applications, including security monitoring, accounting, quality of service and intrusion detection and prevention. The techniques of traffic classification can be divided into two categories: statistical and explicit. Statistical techniques classify the traffic according to its statistical properties, such as traffic load, packet size or packet inter-arrival time [1]. While a statistical traffic classifier could classify traffic with little or no prior knowledge, it is usually limited by scalability issues and the sustained probability of false positives.

Explicit traffic classification techniques usually classify traffic based on predefined filters or policies. These filters can either be configured manually or generated automatically by analyzing the statistical traffic classifier. Although the effectiveness of an explicit traffic classifier relies on the quality of the predefined filters, it usually is more scalable and accurate than statistical traffic classifier. Moreover, predefined filters can let a definite traffic classifier adjust its data structures for specific applications.

Packet classification is a specialized technique for explicit traffic classification of field specifications of packet headers. Packet classification can meet network applications' requirements by applying consistent actions defined in the filters to the applications' incoming packets. The actions include queuing disciplines, access control, accounting and intrusion detection [2,3]. In addition, various devices, such as routers, firewalls and network intrusion detection systems, have used packet classification to practical effect. Thus, a wide range of network applications depend on the performance of state-of-the-art packet classification. Nevertheless, packet classification with a large number of filters is complex and usually has poor worst case performance [4–7].

The filters for packet classification consist of a set of fields and an associated action. Each field, in turn, corresponds to one field of the packet header. The value

* Tel.: +886 4 22840497x710.
E-mail address: pcwang@cs.nchu.edu.tw
1 This work is supported in part by the National Science Council under Grant No. NSC 97-2221-E-005-049.

in each field could be a variable-length prefix, range, explicit value or wild card. The most common fields include the source and destination IP address prefixes, source and destination port ranges of the transport protocol and the protocol type in the packet header. Formally, we define a filter F with d fields as $F = (f_1, f_2, \ldots, f_d)$. While classifying packets, we say that a packet header P matches a particular filter F if for all i, the ith field of the header satisfies f_i. Each action has a cost that defines its priority among the actions of the matching filters: the classifier only applies the lowest-cost action from the matching filters [3].

The problem of packet classification resembles the point location problem in multidimensional space [5]. The point location problem finds the enclosing region of a point within a set of non-overlapping regions. The best upper bounds for classifying packets on N filters with d fields are either $O(\log N)$ time and $O(N^d)$ space or $O(\log^{d-1} N)$ time and $O(N)$ space.

To improve the search performance, much effort has been devoted to packet classification in recent years, yielding a number of algorithms. Out of these, the algorithms based on cross-producting (e.g., [4,5]) are usually regarded as the fastest. These algorithms generate all possible combinations of the filter field specifications and pre-compute the best matching filter for each combination. Each packet classification starts from d one-dimensional searches for the best matching prefix of each field. Then, the algorithm finds the index of the best matching filter from the combination of these best matching prefixes. The search is efficient because a one-dimensional search is much faster than a multidimensional one. However, due to the extensive pre-computation, these algorithms cannot support incremental updates. Moreover, the storage consumption of the cross-producting-based algorithms is usually unacceptable with increasing sizes of filter databases.

In this paper, we propose a new cross-producting-based algorithm with linear search to handle any filter databases undergoing a size expansion as a result of new applications. The proposed algorithm, *Controlled Cross-producting*, is inspired by the following observation. Prior to finding the matching prefixes by cross-producting, we do d one-dimensional searches for the best matching prefix of each field. As the number of distinct prefixes in each field increases, the search performance of the one-dimensional searches degrades, and the cross-product table consumes an exponentially increasing amount of storage. However, an increasing number of distinct prefixes also indicates that more filters can be distinguished simply by the specifications of one field. These distinguishable filters can be searched with one-dimensional data structures and excluded from the cross-product table. As a result, the overall storage performance improves by orders of magnitude.

The proposed scheme can adjust storage usage by controlling the number of filters that are distinguishable through one-dimensional searches. With more filters searched by one-dimensional data structures, the storage needed for the cross-product table could be further decreased while trading off search speed. Although cross-producting-based algorithms are rarely regarded as updatable, our scheme supports incremental updates fully. In our experiments, we evaluate the performance of our scheme with filter databases of varying sizes and characteristics. We use twelve different types of filter databases, whose sizes vary from 16 K to 128 K. Our results show that our scheme can provide superior search performance while keeping the space requirement comparable with the prominent existing schemes.

The rest of the paper is organized as follows: Section 2 addresses related work. Section 3 presents our novel algorithm and data structure. Section 4 describes our approach to incremental updates. Section 5 explains the experimental setup and evaluates our scheme in detail. Finally, Section 6 concludes.

2. Related works

Algorithms aimed at a fast and efficient packet classifier have appeared in the recent literature [5,8,4,6,9,10,7,11–20]. These existing algorithms can be divided into six categories. The following subsections provide a brief description of these categories.

2.1. Hardware-based solutions

There are two categories of hardware-based solutions: ternary content addressable memory (TCAM) and bit vector. The TCAM cell stores an extra "Don't Care" state to achieve arbitrary bit mask matches, such as IP address lookup and packet classification. TCAMs have been proven effective for packet classification with a high degree of parallelism [2]. The drawbacks of TCAMs include their smaller density, power dissipation and extra entries due to the range-to-prefix transformation [12,13]. Much effort has gone into improving the power and storage efficiency of TCAMs [14,16,15,17–20].

Another scheme, *Lucent Bit Vectors* (*LBV*), does d one-dimensional searches to derive d lists of filters with at least one matching field. Since each list is in the form of a bit vector, *LBV* is suitable for hardware implementation. The main drawback is its memory consumption: $O(dN^2)$. Further work in [13] introducing aggregate bit-vectors has demonstrated dramatic improvement in the speed performance of *LBV*.

2.2. Decision-tree-based solutions

The decision-tree-based algorithms include works presented in [11,7]. Both schemes divide the filters into multiple groups with a decision tree. Each group corresponds to a leaf node of the decision tree, and the algorithms traverse the group with a linear search. The number of filters in each group is limited by a predefined value. The cut rule at each node may either be a value [11] or a bit [7] of any field. An optimized set of cut rules would minimize the required storage and search time. *HyperCuts* [21] further extends the one-dimensional cut rules to multidimensional ones.

2.3. Cross-producting-based solutions

Cross-producting is a general mechanism that looks up best matching prefixes on individual fields and combines the results of individual prefix lookups with a pre-computed table [4]. However, this scheme suffers from an $O(N^d)$ memory blowup for d-field filters. Gupta and McKeown [5] present an algorithm, *Recursive Flow Classification*, that can be seen as a generalization of cross-producting: it does cross-producting recursively. In each iteration, the algorithm generates a cross-producting table with only a subset of the inspected fields, saving storage by eliminating unmatched entries. The algorithm lowers storage usage significantly, but its space complexity remains $O(N^d)$. Likewise, in the case of two-field filters, this scheme is identical to the cross-producting scheme and requires $O(N^2)$ space. In a recent work, *Distributed Cross-producting of Field Labels* (*DCFL*) [22] further improves the representation of cross-producting entries in [5] with efficient set membership data structures and reduces storage usage greatly, although it relies heavily on hardware parallelism.

2.4. Extended grid-of-tries with path compression

The algorithm of *Extended Grid-of-Tries with Path Compression* (*EGT-PC*) evolves from *Grid-of-Tries* (*GT*) in [4] to support filters with more than two fields [12]. The algorithm combines an enhanced data structure of *GT* with the filter lists of linear search. Its search performance may not be comparable to other schemes, since it queries at most $O(W^2)$ combinations of source and destination address prefixes [3].

2.5. Independent sets

The idea behind *Independent Sets* is to categorize multi-dimensional filters according to the specifications of one field [23]. The filters in each independent set are mutually disjoint; therefore, the algorithm can derive the matching filter in each independent set with binary search. Exactly one independent set stores each filter, which reduces memory consumption. The search performance is thus tied to the number of independent sets.

2.6. Hash-based solutions

The hash-based idea [8] has given rise to multidimensional filters in a previous study [24]. A hash table stores filters with identical prefix length combinations, and creating a hash key just means concatenating the prefixes. For example, the two-dimensional filters $F = (10*, 110*)$ and $G = (11*, 001*)$ both belong to the tuple $T_{2,3}$. When searching for $T_{2,3}$, the algorithm makes a hash key by concatenating two bits of the source field with three bits of the destination field. It can find the matching filters by probing each tuple alternately. Rectangle search and pruned tuple space search improve the performance of tuple space search. In [25,26], the speed and storage performance of the rectangle search are improved by reducing the number of tuples. In [27], *Entry Pruned Tuple Search* (*EPTS*) enhances the pruned tuple space search by storing pruning information in each filter in the form of a bitmap of tuples containing non-conflicting filters. However, inserting a new tuple might cause all tuple bitmaps in the filters to be updated. In addition, *EPTS* requires an amount of storage proportional to the number of tuples. Hence, the *EPTS* scheme may not be scalable with respect to the size of tuple space.

Table 1
A comparison of time and space complexity.

Algorithm	Time	Space
Ternary CAM	$O(1)$	$O(N)$
Lucent bit vector [6]	$O(d\log N + N/B)$	$O(dN^2)$
Aggregate bit vector [13]	$O(d\log N + N/B)$	$O(dN^2)$
HiCuts [11]	$O(d)$	$O(N^d)$
HyperCuts [21]	$O(d)$	$O(N^d)$
Cross-producting [4]	$O(d\log N + 1)$	$O(N^d)$
RFC [5]	$O(d)$	$O(N^d)$
DCFL [22]	$O(d)$	$O(dNW)$
Grid of tries [4]	$O(W^{d-1})$	$O(dNW)$
EGT-PC [12]	$O(W^2)$	$O(dNW)$
Independent sets [23]	$O(d\log N + I)$	$O(IN)$
Rectangle search [8]	$O(2W - 1)$	$O(NW)$
Pruned tuple space search [8]	$O(d\log N + W^d)$	$O(N)$
Controlled cross-producting	$O(d\log N + I)$	$O(((W - I)N/W)^d)$

B: memory width, d: number of fields, I: number of independent sets, N: number of filters, T: number of hash tables, W: field length, $d\log N$: one-dimensional search time.

In summary, we compare the time and space complexity of the different algorithms in Table 1. Theoretically, the schemes based on decision trees [11,21] and cross-producting [4,5] achieve the best search performance, while *Pruned Tuple Space Search* [8] and *Independent Sets* [23] require minimal storage. However, these schemes also have some drawbacks. For example, the schemes in [8,23] do not seem to bound search latency [8,2], and the schemes in [4,5] may not update or scale for large filter databases [2,3]. We are aware of the trade-off between time and space complexity in the existing schemes and attempt to make packet classification more scalable. We propose several new ideas to increase the storage efficiency of the data structures for storing filters. As shown in Table 1, the performance of the proposed scheme varies between that of *Independent Sets* and cross-producting schemes. The next section derives the derivation of time and space complexity of our scheme.

3. Our scheme

Before introducing our scheme, we will discuss the cross-producting-based schemes in terms of their important properties that lead to the inspiration of the proposed scheme.

Srinivasan et al. introduced the seminal technique of cross-producting [4]. The *Cross-producting* scheme is motivated by the observation that the number of unique field specifications is usually significantly less than that the number of filters in the filter databases. Therefore, the *Cross-producting* scheme pre-computes the best matching filter for each combination of the d field specifications in the cross-product table. To classify packets, the algorithm does d one-dimensional searches to find the best matching prefix of each field. Next, it retrieves the combination corresponding to the best matching prefixes to derive the index of the best matching filter from the cross-product table.

Table 2 uses a filter database with 12 filters and five fields as an example. Table 3 lists the distinct specifications of each field. There are 720 different combinations in total, which means there are 60 combinations for each filter on average. Table 4 shows the generated cross-product table. In the worst case, every filter has different a field specification; hence, N filters would result in N^d combinations.

There are several approaches to improving the storage performance of the original cross-producting scheme. The *Recursive Flow Classification* (*RFC*) scheme [5] eliminates unused combinations by gradually cross-producting a subset of the inspected fields. Although the *RFC* scheme could reduce storage requirements significantly as compared with the original cross-producting scheme, it is still not feasible to support filter databases with more than 15 K filters [5].

Another approach to reducing the number of combinations is to use a hash-based cross-product table. In [4,5], the cross-product table is assumed to be implemented with a direct-access table, in which each combination corresponds to one entry of the table. However, there are usually several combinations that do not match any filter. In the previous example, the combination, $C = \langle *, 1*, [10:10], *, TCP\rangle$, which partially overlaps with F_2, F_3, F_7, F_{11} and completely covers F_4, is disjoint with the other filters. However, the incoming packet header that matches combination C may not match these overlapping filters since none of them completely cover the space of combination C. As a result, combination C is useless, since it does not match any filters. In the extreme case of an N-filter database in which the field specifications of the filters are either completely disjoint or completely overlapping, the size of the cross-product table can be lowered to N by removing the redundant combinations.

The third approach is inspired by the correlation between the number of distinct field specifications and the required storage of the cross-product table. While storage performance depends on the number of distinct field specifications, the cross-producting-based schemes would suffer from severe degradation on storage performance when the size of filter database expands. Nevertheless, there are some algorithms that perform well on the filter databases with numerous field specifications. Take the decision-tree-based algorithms as an example. The procedure of tree construction usually picks the field with the most distinct specifications for space decomposition. Since space decomposition reduces the number of filters sharing the same specification in the selected field, it could categorize filters better. Hence, the decision-tree-based algorithms perform better on the filter databases with miscellaneous field specifications.

Another algorithm that could benefit from the miscellaneous field specifications is the *Independent Sets* scheme [23]. As mentioned above, the filters are categorized into different independent sets, whose filters are disjoint on a selected field. With more distinct field specifications, each independent set could store more filters, which results in fewer independent sets and yields better search performance.

Based on the observation, we are motivated to combine the cross-producting-based schemes with the decision-tree-based algorithms or the *Independent Sets* scheme to reach a better balance between storage and speed performance. However, decision-tree-based algorithms do not require one-dimensional searches. Hence, we optimize the original cross-producting scheme by incorporating the concept of independent sets into it. The union of the two schemes only requires moderate modification, since the search procedures of both schemes perform one-dimensional searches.

Our crude idea to merge the two schemes is to divide the filters into two groups. The first group includes filters searched by one-dimensional search and the other group includes those searched by cross-producting. The selection procedure involves the use of field specifications. Take the filters in Table 2 as an example. The filters F_1, F_4, F_7, F_8, F_9 and F_{11} contain one field specification, which is not specified in the other filters. Therefore, we can categorize these filters into the one-dimensional search group. After the categorization, the rest of the filters would contribute to at most $162 (= 3 \times 3 \times 3 \times 3 \times 2)$ combinations. As compared to the original cross-producting scheme, which gen-

Table 2
An example with 12 filters on five fields.

Filter	f_1	f_2	f_3	f_4	f_5	Action
F_0	000*	111*	[10:10]	*	*UDP*	act_0
F_1	000*	111*	[01:01]	[10 : 10]	*UDP*	act_0
F_2	000*	10*	*	[10:10]	*TCP*	act_1
F_3	000*	10*	[00:10]	[01:01]	*TCP*	act_2
F_4	000*	10*	[10:10]	[11:11]	*TCP*	act_1
F_5	0*	111*	[10:10]	[01:01]	*UDP*	act_0
F_6	0*	111*	[10:10]	[10:10]	*UDP*	act_0
F_7	0*	1*	*	*	*TCP*	act_2
F_8	*	01*	[00:10]	*	*TCP*	act_2
F_9	*	0*	*	[01:01]	*UDP*	act_0
F_{10}	*	*	*	*	*UDP*	act_3
F_{11}	*	*	*	[01:11]	*TCP*	act_4

Table 3
The distinct specifications of each field.

f_1	f_2	f_3	f_4	f_5
000*	111*	[10:10]	*	*UDP*
0*	10*	[01:01]	[10:10]	*TCP*
*	1*	[00:10]	[01:01]	
	01*	*	[11:11]	
	0*		[01:11]	
	*			

Table 4
The cross-product table of the filters in Table 2.

Index	Crossproduct entry	Matching filter
0	$000*, 111*, [10:10], *, UDP$	F_0
1	$000*, 111*, [10:10], *, TCP$	F_7
2	$000*, 111*, [10:10], [10:10], UDP$	F_0
3	$000*, 111*, [10:10], [10:10], TCP$	F_7
4	$000*, 111*, [10:10], [01:01], UDP$	F_0
5	$000*, 111*, [10:10], [01:01], TCP$	F_7
•	•	•
•	•	•
718	$*, *, *, [01:11], UDP$	F_{10}
719	$*, *, *, [01:11], TCP$	F_{11}

erates 720 combinations, the new scheme could make storage orders of magnitude more efficient. We can repeat this procedure to achieve further reduction. For example, filter F_3 is put into the one-dimensional search group in the second iteration, and filter F_9 is selected in the third iteration. Then, the number of combinations is further reduced to 72. Therefore, the proposed scheme can adjust the required storage to fit into the available memory space.

Nevertheless, there is a trade-off in the proposed scheme. Additional search cost is necessary since the original cross-producting scheme needs one-dimensional searches, the cost lies in making extra comparisons for the filters in a one-dimensional search group. Although this extra cost is a reasonable trade-off for storage savings, it can be minimized further by taking the cost of extra comparisons into account. We can model the problem of minimizing the number of cross-product entries and the number of filters in the group of one-dimensional search as a maximum clique problem. Assume that we represent the filters with a graph $G = (V, E)$ in which each vertex in V represents a filter. Two vertices are connected by an edge if the two filters are completely disjoint. To achieve optimal filter categorization, we must find the largest clique in the graph G. However, the maximum clique problem is known to be NP-complete [28]. In the following, we present a greedy algorithm to construct our data structures.

3.1. Controlled Cross-producting

The first step in constructing the data structures of our scheme is to transform ranges of the filters to prefixes. In the original cross-producting scheme, the ranges must be divided into primitive ranges that are mutually disjoint [15]. Consequently, the primitive ranges are not updatable due to the new range insertion that might result in new primitive ranges. To support incremental updates, we transform the ranges into prefixes by splitting each range into multiple subranges, where each subrange uniquely corresponds to one prefix [4]. The main drawback of this approach is an increase in the number of filters and distinct field specifications.

Let us consider the example in Table 2. Each filter contains two range fields, f_3 and f_4, which must be converted to the prefix form. For example, range [10:10] is converted to prefix $\langle 10* \rangle$, and range [00:10] is converted to two prefixes, $\langle 0* \rangle$ and $\langle 10* \rangle$. Once the range is converted into more than one prefix, the filter is duplicated as well. As shown in Table 5, the original filter database is transformed into 15 filters after doing the basic range-to-prefix transformation. In the worst case, $2(W-1)$ prefixes are required to cover a range, where W is the length of a range field. Therefore, a single filter with two range fields could be transformed into at most $4(W-1)^2$ filters with pure prefix fields. The distinct field specifications might increase as well, although that is not the case in this instance.

After doing the range-to-prefix transformation, the algorithm constructs a prefix trie of each field based on the field specifications of the new filters. The trie node corresponding to the field specification of the filter called the "prefix node". Each prefix node maintains a filter list for the subsequent procedures. The filter is inserted into the filter list of the prefix node to which its field specification corresponds. A filter that is put into the one-dimensional search group is removed from the filter list. When the filter list length of a prefix node reaches zero, the field specification corresponding to the prefix node can be removed to diminish the number of cross-producting combinations. Accordingly, the proposed algorithm tries to maximize the number of prefix nodes with an empty filter list while minimizing the number of extra comparisons for the filters in the one-dimensional search group.

Table 5
The converted filters with the range-to-prefix transformation.

Filter	f_1	f_2	f_3	f_4	f_5	Action
F_a	000*	111*	10*	*	*UDP*	act_0
F_b	000*	111*	01*	10*	*UDP*	act_0
F_c	000*	10*	*	10*	*TCP*	act_1
F_d	000*	10*	0*	01*	*TCP*	act_2
F_e	000*	10*	10*	01*	*TCP*	act_2
F_f	000*	10*	10*	11*	*TCP*	act_1
F_g	0*	111*	10*	01*	*UDP*	act_0
F_h	0*	111*	10*	10*	*UDP*	act_0
F_i	0*	1*	*	*	*TCP*	act_2
F_j	*	01*	0*	*	*TCP*	act_2
F_k	*	01*	10*	*	*TCP*	act_2
F_l	*	0*	*	01*	*UDP*	act_0
F_m	*	*	*	*	*UDP*	act_3
F_n	*	*	*	01*	*TCP*	act_4
F_o	*	*	*	1*	*TCP*	act_4

In the following, we describe the procedure for filter selection on each iteration. First, the algorithm selects the filters to be searched in the one-dimensional data structure with the prefix trie with the maximal number of leaf nodes. The purpose of this step is to maximize the number of filters put into the one-dimensional search group in this iteration. As the number of distinct field specification increases, the number of leaf nodes usually multiplies as well. As a result, the algorithm could lower the number of cross-product entries further by selecting the filters associated with these leaf nodes. Note that there might be more than one associated filter in a leaf node. In this case, we select the filter such that one of the corresponding prefix nodes has the shortest filter list.

After removing the selected filters, all prefix tries are recomputed based on the field specifications of the rest of the filters. The algorithm finds the number of cross-product entries of the remaining filters to decide whether the required storage is below a predefined threshold. If not, the above steps, including selecting the prefix trie with maximal leaf nodes and selecting another set of filters, are repeated before the procedure of filter categorization is complete.

We note that the filters selected in each iteration must be disjoint in at least one field. Therefore, these filters can belong to an independent set, and the one-dimensional search group could be treated as a set of independent sets. Since each independent set would result in at most one extra filter comparison, we can rephrase the goal of the proposed algorithm as maximizing the number of prefix nodes with empty filter lists while minimizing the number of independent sets in the one-dimensional search group. Therefore, the proposed algorithm is different from the algorithm in [23] that focuses on minimizing the number of independent sets.

We illustrate the procedure of filter selection with the previous example. Fig. 1 shows the prefix tries of five fields. For an established binary trie, each prefix node has a unique identifier to simplify the following description. We also list the associated filters of each prefix node. The number in parentheses denotes the number of associated filters.

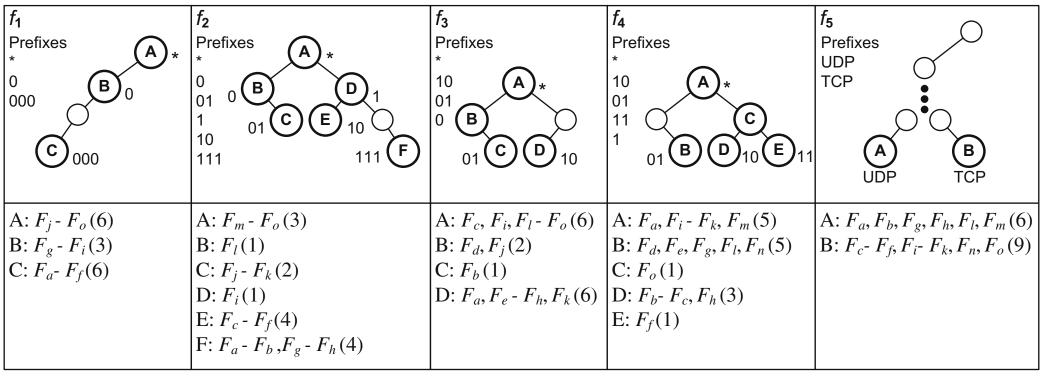
The first step selects a prefix trie with the maximal number of leaf nodes. This step can select either the prefix trie of f_2 or of f_4, and we randomly choose the former. Next, for the prefix trie of f_2, the filters in the filter list of each leaf node are the candidate selected filters. Among these candidates, one filter is put into the corresponding independent set for each leaf node. In the prefix trie of f_2, there are two candidates, F_j and F_k, in node **C**. For filter F_j, its third field specification corresponds to node **B** of prefix trie of f_3 which is associated to two filters. Since the corresponding nodes of F_k associate with at least five filters, F_j is selected and removed from the all corresponding prefix nodes, including node **A** of f_1, node **C** of f_2, and so on.

Similarly, the filter F_f is selected in node **E**, and F_b is selected in node **F**. After dissociating these selected filters from their corresponding nodes, both the numbers of distinct field specifications on f_3 and f_4 are decreased by one, and the number of the cross-product entries is reduced from 720 to 432. The resulting prefix tries are shown in Fig. 2, where the newly generated independent set is listed in the shaded grid.

Until the number of distinct cross-producting combinations decreases to less than a predefined threshold (for example, 100), the above steps are repeated. In the second and third iterations, the prefix trie of f_2 is still selected. Three filters, F_d, F_h and F_k, are taken in the second iteration and another three filters, F_c, F_g and F_l, are picked in the third one. After three iterations of filter selection, three independent sets are generated, and the number of cross-producting combinations is reduced to 144. Fig. 3 shows the resulting prefix tries.

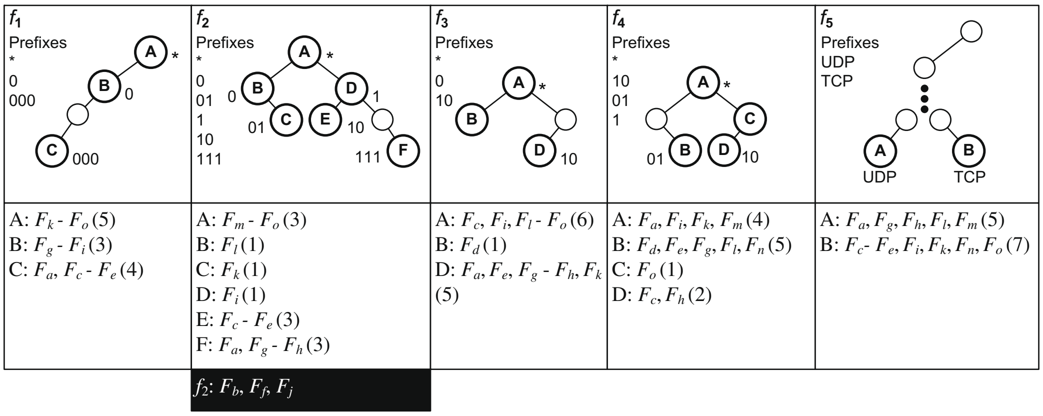
The fourth iteration adopts another prefix trie of f_4 and chooses two filters, F_e and F_o. After the fourth iteration, the number of cross-producting combinations drops to 72, which causes the procedure to stop. Fig. 4 shows the resulting prefix tries, where the ratio of storage reduction is 90%. There are four independent sets with nine filters. To classify packets, the algorithm inserts the filters of the independent sets into the one-dimensional data structure and inserts the rest into the cross-product table. In the following, we present the search procedure and the data structures of the proposed scheme.

The search procedure consists of two parts: d one-dimensional searches and one access to the cross-product



f_1	f_2	f_3	f_4	f_5
Prefixes * 0 000	Prefixes * 0 01 1 10 111	Prefixes * 10 01 0	Prefixes * 10 01 11 1	Prefixes UDP TCP
A: F_j - F_o (6) B: F_g - F_i (3) C: F_a- F_f (6)	A: F_m - F_o (3) B: F_l (1) C: F_j - F_k (2) D: F_i (1) E: F_c - F_f (4) F: F_a - F_b ,F_g - F_h (4)	A: F_c, F_i, F_l - F_o (6) B: F_d, F_j (2) C: F_b (1) D: F_a, F_e - F_h, F_k (6)	A: F_a, F_i - F_k, F_m (5) B: F_d, F_e, F_g, F_l, F_n (5) C: F_o (1) D: F_b- F_c, F_h (3) E: F_f (1)	A: F_a, F_b, F_g, F_h, F_l, F_m (6) B: F_c- F_f, F_i- F_k, F_n, F_o (9)

Fig. 1. The prefix tries of the filters in Table 5.



f_1	f_2	f_3	f_4	f_5
Prefixes * 0 000	Prefixes * 0 01 1 10 111	Prefixes * 0 10	Prefixes * 10 01 1	Prefixes UDP TCP
A: F_k - F_o (5) B: F_g - F_i (3) C: F_a, F_c - F_e (4)	A: F_m - F_o (3) B: F_l (1) C: F_k (1) D: F_i (1) E: F_c - F_e (3) F: F_a, F_g - F_h (3)	A: F_c, F_i, F_l - F_o (6) B: F_d (1) D: F_a, F_e, F_g - F_h, F_k (5)	A: F_a, F_i, F_k, F_m (4) B: F_d, F_e, F_g, F_l, F_n (5) C: F_o (1) D: F_c, F_h (2)	A: F_a, F_g, F_h, F_l, F_m (5) B: F_c- F_e, F_i, F_k, F_n, F_o (7)
	f_2: F_b, F_f, F_j			

Fig. 2. The prefix tries with one independent set.

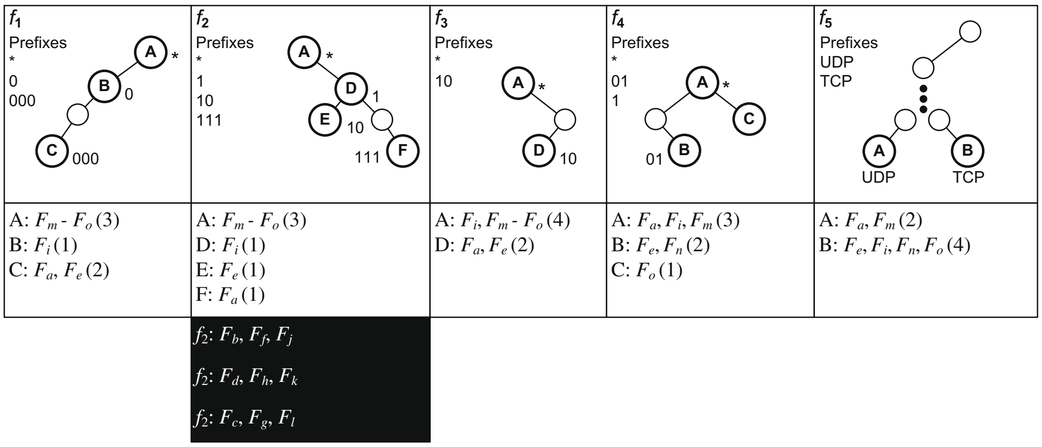


Fig. 3. The prefix tries with three independent sets.

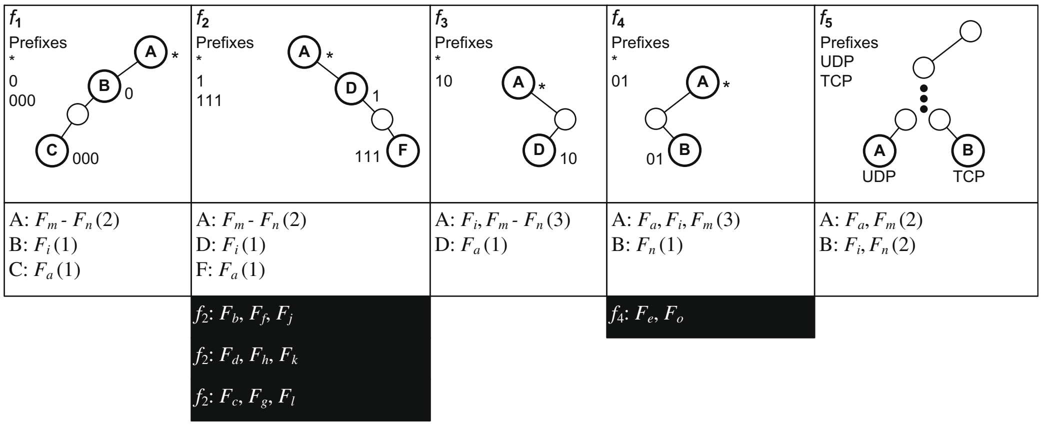


Fig. 4. The prefix tries with four independent sets.

table. The original cross-producting scheme differs from the proposed one-dimensional search in that it requires extra comparisons to the filters in the independent sets. Therefore, the data structure for one-dimensional search contains the necessary field specifications for indexing the cross-product table and those for accessing the independent sets.

In our example, there is no independent set on f_1, f_3 and f_5. Hence, the data structures of these three fields can be constructed from the corresponding field specifications of the filters that are excluded from the independent sets. For the field f_2, the data structure includes the field specifications of most filters, except for F_e and F_o. Fig. 5 shows the prefix tries for one-dimensional searches. In the prefix trie of f_2, there are four nodes, **B, C, E** and **F**, which contain the filters to be compared, and three nodes, **A, D, F**, which are searched for the best matching prefixes. Since node **F** has two purposes, it is shaded in light grey along with node **B** in field f_4.

One-dimensional search on the prefix trie must handle three cases. In the first case, the best matching prefix node is unshaded, meaning that there are no filters of indepen-

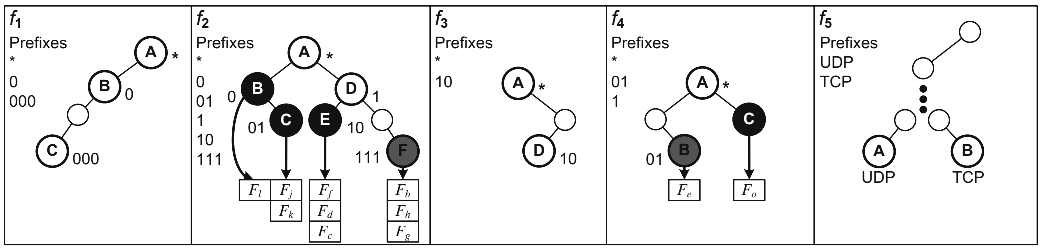


Fig. 5. The prefix tries for one-dimensional searches.

dent sets to be compared. In this case, the prefix corresponding to the best matching node is used only to calculate the index of the cross-product table. In the second case, if the best matching prefix node is shaded in dim gray, the algorithm compares the associated filters sequentially and cross-products with the prefix corresponding to the last-retrieved, unshaded prefix node. In the last case, the best matching node in light gray corresponds to the best matching prefix for cross-producting and a filter bucket for linear search. In these two cases in which shaded prefix nodes match, the algorithm must compare the associated filters in the parent of the best matching node for possible matching.

In Fig. 5, a specification of f_2, "101", matches to node **E**, meaning that the best matching prefix for cross-producting is $\langle 1* \rangle$, and three filters, F_f, F_d and F_c, are compared. Another f_2 specification "111" matches to node **F**, whose best matching prefix is $\langle 111* \rangle$, and another three filters, F_b, F_h and F_g, are compared. When node **C** is matched, the associated filters of node **C** and node **B** are compared sequentially.

We note that the prefix trie data structures can be replaced by other data structures to search for the best matching prefix, such as hash tables [24] or multi-way search trees [29]. In this work, we adopt the multi-way search tree data structure for one-dimensional search. Our main reason for choosing this data structure is its scalability with respect to the number of prefixes and the prefix lengths. Although the data structure only supports local reconstruction, we can minimize this disadvantage because the filters usually share their field values [22,30]. The time complexity of the proposed scheme is thus equal to $O(d \log N + I)$, where I is the number of independent sets.

The space complexity depends on the storage that the cross-product table uses. Assume that there are N distinct field specifications in each field, among which at least N/W specifications correspond to the leaf nodes. Hence, there are N/W filters in the smallest independent set. After generating I independent sets, the number of cross-producting combinations thus equals $(N - I \times N/W)^d = O(((W - I)N/W)^d)$.

3.2. Refinements

In the following, we present two techniques to improve the search and storage performance. The first technique is an approach of approximate range-to-prefix transformation. The other technique is cache line alignment for optimizing linear search access speed. In the following, we describe these two techniques in detail.

3.2.1. Approximate range-to-prefix transformation

As mentioned above, supporting incremental updates requires transforming the range fields of a filter. However, such an approach would create more field specifications and increase the number of cross-product entries. In addition, the increasing number of filters would also create more independent sets.

Several TCAM-based algorithms have been designed for improving the storage efficiency of range representation significantly. These algorithms can be categorized into those that encode ranges with mask extension[2] [15,17–20] and those that generate extra fields [16]. Unfortunately, these algorithms do not apply to our scheme. The algorithms of the former category use the hardware support of three states in TCAMs, and the extra field for the algorithm in [16] would increase the number of cross-product entries.

We develop an efficient approach to range-to-prefix transformation for our data structures. Our approach is to represent a range by its smallest enclosure prefix. For the filters in Table 2, the ranges [00:10] and [01:11] are both transformed to prefix $\langle * \rangle$. Since one prefix represents each range, our approach does not duplicate any filters. Table 6 lists the transformed filters. The algorithm then uses these filters to generate the prefix tries for deriving the independent sets. We note that the new approach is only used to select filters. Therefore, the approach of approximate range-to-prefix transformation only affects how the filters are categorized. The construction of the cross-product table and of the filters in the filter bucket for linear search is still based on the original filters; that is, the filters with range fields, for correctness.

Next, the algorithm generates three independent sets on field f_2. In the first iteration, it picks three filters, F_1, F_4 and F_8, and in the second iteration, it picks another three filters, F_2, F_6 and F_9. In the last iteration, it picks the filters F_0 and F_3. Then it generates the cross-product table with the remaining four filters by doing the original range-to-prefix transformation. Fig. 6 shows the prefix tries for one-dimensional searches. In this example, there are only three independent sets with 72 cross-producting combinations. Therefore, the storage needed for the one-dimensional data structures could be reduced while improving search performance.

This approach has a potential drawback of increasing the number of associated filters in each prefix node. This drawback would increase linear search time. However, our experiments show that most independent sets are generated on the fields of IP source and destination address prefixes. Therefore, such effects can be minimized since

Table 6
The converted filters by using the approximate range-to-prefix transformation.

Filter	f_1	f_2	f_3	f_4	f_5	Action
F_0	000*	111*	10*	*	*UDP*	act_0
F_1	000*	111*	01*	10*	*UDP*	act_0
F_2	000*	10*	*	10*	*TCP*	act_1
F_3	000*	10*	*	01*	*TCP*	act_2
F_4	000*	10*	10*	11*	*TCP*	act_1
F_5	0*	111*	10*	01*	*UDP*	act_0
F_6	0*	111*	10*	10*	*UDP*	act_0
F_7	0*	1*	*	*	*TCP*	act_2
F_8	*	01*	*	*	*TCP*	act_2
F_9	*	0*	*	01*	*UDP*	act_0
F_{10}	*	*	*	*	*UDP*	act_3
F_{11}	*	*	*	*	*TCP*	act_4

[2] The technique of mask extension extends the prefix mask to be any arbitrary combination of bits [31].

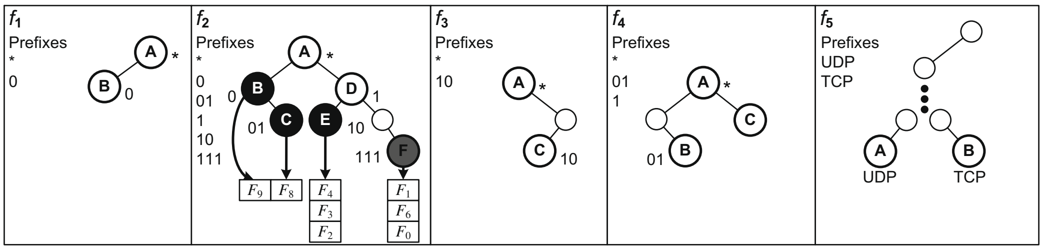


Fig. 6. The prefix tries for one-dimensional searches with approximate range-to-prefix transformation.

these fields are not affected by the proposed approach to range-to-prefix transformation.

3.2.2. Cache line alignment

The other technique to improve search performance is cache line alignment, which takes advantage of the fact that processors fetch entire cache lines from dynamic random access memory (DRAM) even when only a single word is required [9]. The access time is dominated by the latency of accessing the first word in the cache line; therefore, the remaining words come at a very low cost [9]. This technique has been used to speed up several algorithms [32,29,33,9]. We apply this technique to increase the rate of accessing the linear-searched filters by fitting these filters into one or more cache lines. Currently, the size of each cache line in a modern processor could be as large as 64 or 128 bytes. Therefore, each cache line could hold three filters (20 bytes per filter). With this technique, we could merge several filter buckets to achieve better search efficiency by re-associating filters from a node to its parent node.

For the example in Fig. 6, the filter buckets of node **B** and node **C** can be merged to adhere to the size limit of a cache line. To achieve this, F_8 is re-associated to node **B** and node **C** is no longer needed. As a result, all filters of a bucket can be read in a single cache line access.

4. Incremental updates

A filter update can either be an insertion or a deletion. For our scheme, both types of updates would affect one of the data structures (independent sets or a cross-product table). These data structures also relate to the data structure of the best matching prefix search. Since the update procedure on the best matching prefix data structure has been studied extensively [34], we focus on the update procedure of the proposed data structures. Moreover, the cross-product table can be implemented as a direct-access array or a hash table, as mentioned above. The implementation using the direct-access array can increase the access rate, but it is usually not updatable due to its consecutively allocated entries. Therefore, our update procedure only applies to a hash-based cross-product table. In the following, we present the procedures for each type of update and data structure.

First, consider filter deletion. Since each filter can be stored in either data structure of our scheme, the update cost depends on where the deleted filter is located. If the deleted filter is stored in the one-dimensional search group, the filter can be removed directly. Otherwise, the cross-product entries which only relate to the deleted filter are removed. The deleted filter would overlap with the other filters in the cross-product table in the worst case; therefore, the update cost is M^d, where M is the number of filters stored in the cross-product table. The existing literature has reported that filter overlapping is quite rare [5,12,13,22,30]; thus, the update cost is moderate in practice. In the case of updating a large number of cross-product entries for a deleted filter, we can insert a new filter, whose field specifications are identical to those of the deleted one, into the one-dimensional search group to signal filter deletion. The packet matching the deleted filter would also match the new filter, and the action of the deleted filter is ignored. The new filter will be eliminated after the corresponding cross-product entries are removed or updated.

Next, we introduce the update procedure for filter insertion. Unlike the update procedure for filter deletion, we can decide into which one of the data structures to insert the new filter by considering space and time costs. Subsequently, we will present the insertion procedures for both data structures.

In the case of inserting a new filter into the one-dimensional search group, the corresponding prefix node of each field is derived first. The new filter can be inserted into the linear search filter bucket, which corresponds to one of these prefix nodes. Intuitively, we select the shortest one to minimize search latency.

In the second case, we present how to insert a new filter into the cross-product table cheaply. The major issue of updating the cross-product table is how to handle the new combinations caused by a newly inserted filter. When a new filter is inserted, the cross-product table would have to include $(N+1)^d - N^d$ new combinations of new intersections in the worst case.

To address this problem, we replace the new field specification of the inserted filter with its longest existing enclosure prefix. After the replacement, there is at most one new combination for the cross-product table. Yet, the existing combination might have corresponded to a matching filter. In this case, the new filter is appended to the existing filter and searched linearly. The other existing entries intersected by the new filter are also modified if necessary. Although all entries are still modified in the worst case, filter intersection is usually far less than in the worst case [5,12,13,22,30]. Hence, the cost of inserting a new filter can be reduced greatly with only moderate storage overhead and few memory updates.

Let us consider the insertion of a new filter, $\langle 01*, 11*, [10:10], [01:01], TCP\rangle$, into our filter data structure from Table 2. Assume that the filter is inserted into the cross-product table. According to Fig. 6, $\langle 01*\rangle$ is a new prefix of f_1, and $\langle 11*\rangle$ is a new prefix of f_2. Therefore, the number of new combinations could be as many as 72. With the proposed approach, the new filter would be represented by $\langle 0*, 1*, 10*, 01*, TCP\rangle$. Since all field specifications exist, no new combination is inserted, and the cross-product table need not be reconstructed.

Although the proposed insertion procedure for the cross-product table could minimize the update cost, search performance degrades if the inserted filter intersects with the other filters. In this case, we could insert the new filter into the one-dimensional search group for better storage usage.

In summary, the proposed procedures could minimize filter update costs. The efficiency comes from the flexibility of the unique design of hybrid data structures. As a result, both search performance and storage efficiency could be maintained at the same level without losing the accuracy of our data structures.

5. Performance evaluation

In this section, we describe how our *Controlled Cross-producting* scheme performs on both real and synthetic databases in terms of its speed and storage. We use three real databases in [35] and 52 synthetic databases, generated by a publicly available tool, *ClassBench*, [36] in the experiments. We used the synthetic databases to test the scalability of our scheme since the largest real database contains only 1550 filters. Each synthetic database has different characteristics that are extracted from one of the 12 real databases [36]. Therefore, we can investigate the performance of packet classification in different circumstances or devices, such as ISP peering routers, backbone core routers, enterprise edge routers and firewalls [36]. Besides synthetic databases, *ClassBench* can also generate the tested packet trace for the designated filter databases. Our experiments generate a packet trace that includes headers with 50 times the number of filters along with each tested filter database.

The performance evaluation has three parts. In the first part, we demonstrate the effectiveness of our scheme by presenting the trade-off between the required space and speed. The second part focuses on relating the numerical results of the proposed scheme to other existing schemes in a performance study. In the last part, we evaluate performance based on large synthetic databases to test the scalability of our scheme with respect to the size of filter database. In all parts, the performance metrics include the required storage in kilobytes, and the numbers of memory accesses in the average case (AMA) and the worst case (WMA). The required storage includes space for d multi-way search trees, filter buckets for linear search and one cross-producting table. The values of AMA are derived by dividing the total number of memory accesses for classifying every packet header in the trace to the number of classified packet headers.

5.1. Trade-off between speed and storage

We start the performance evaluation by presenting the trade-off between the required amount of storage and the number of memory accesses for classifying packets. We did this evaluation on three real filter databases that are publicly available in [35]. For each database, we adjust the number of iterations for selecting filters into a one-dimensional search group. Since the cross-product table based on a direct-access array usually consumes much more space, we only consider the case of hash-based cross-product table. The refinement of approximate range-to-prefix transformation is enabled, but we disable cache line alignment to better illustrate the trade-off between required space and speed.

As shown in Fig. 7, the required storage gradually decreases as the number of iterations increases. Since the space used by one-dimensional search groups is propor-

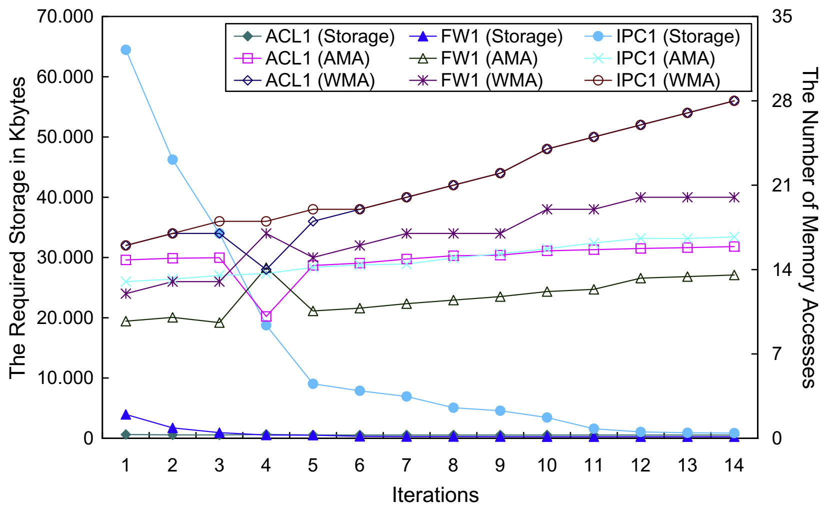


Fig. 7. Trade-off between the required storage and the number of memory accesses.

tional to the number of iterations, storage reduction comes from the decreased number of distinct field specifications. However, such effectiveness becomes less obvious after several iterations for two reasons. The first is that the decreasing number of leaf nodes in prefix tries would result in fewer selected filters in each iteration. The other is that our algorithm selects the filters with rarely used field specifications first. Therefore, the remaining filters usually share more field specifications, which makes the remaining combinations of field specifications difficult to eliminate. Although the number of memory accesses increases as well as the number of iterations, the performance degradation is moderate because only one extra memory access is required for each additional iteration.

We also note that our scheme could achieve better improvement on the required storage for larger filter databases since these databases usually have more distinct field specifications. In Fig. 7, the required storage of IPC1 is much larger than the other two databases initially. However, with our scheme, the required storage could be reduced to the same level as that of the other two filter databases, albeit with higher numbers of memory accesses. Therefore, our scheme is more flexible and feasible with respect to filter database size.

5.2. *Comparative analysis of our scheme and previous schemes*

We did the second evaluation by comparing our scheme with other notable schemes. The schemes from previous work include *ABV* [13], *HyperCuts* [21], *Independent sets* [23] and *RFC* [5]. For the *ABV* scheme, the aggregate size is 32 bits, and the memory width is 256 bits. *HyperCuts* adopts the setting in which the space factor is 1 and bucket size is 32. The source code for the first two existing algorithms is publicly available in [35]. In the following experiments, both refinements, approximate range-to-prefix transformation and cache line alignment, are enabled to show their effect.

To begin with, the evaluation uses the real databases. Table 7 shows the storage performance of the proposed scheme and other existing schemes, and Table 8 lists the speed performance. As compared to *RFC*, the search performance of the proposed scheme is slower as a trade-off to yield better storage consumption and scalability. Although both schemes are based on cross-producting, the proposed scheme could reduce the required storage by eliminating the distinct field specifications. As a result, the scalability of our scheme could be significantly improved.

As compared to other non-cross-producting-based schemes, the proposed scheme seems uncompetitive in storage performance, but our scheme outperforms these schemes in both average and worst case search performance. Both the storage disadvantage and speed advantage come from the cross-product table. While the cross-product table is superior in search speed, it also incurs extra storage.

Next, we use 12 synthetic databases for further evaluation. The synthetic databases are generated by *ClassBench* with default settings. Each database is initialized with 16,000 filters. However, some filters might be redundant, and, as a result, the actual number of filters in each database is usually less than 16,000.

While the storage performance of the proposed scheme is not comparable to the existing schemes using the relatively small, real, filter databases, this effect disappears when the size of the filter database increases. As shown in Table 9, the memory consumption of the proposed scheme is much less than that of *ABV* and most cases of

Table 9
Storage performance of the existing algorithms using synthetic databases.

Syn DBs	Original filters	Aggregate bit vector	HyperCuts	Independent sets	Controlled cross-producting
ACL1	15,926	36,523.81	369.18	557.37	827.55
ACL2	15,447	55,141.81	1,359.68	250.53	1,690.12
ACL3	14,729	7,803.02	2,721.50	187.16	1,650.91
ACL4	15,405	13,306.42	1,985.26	229.30	1,562.94
ACL5	10,379	4,545.09	294.45	119.47	654.02
FW1	14,898	32,494.13	23,883.84	295.37	2,075.47
FW2	15,501	39,885.48	10,859.94	242.20	1,766.12
FW3	14,297	27,532.10	27,172.06	231.41	1,983.00
FW4	13,856	31,231.93	10,717.61	236.98	2,151.53
FW5	14,009	26,260.04	19,503.95	269.85	1,935.06
IPC1	14,954	16,116.21	3,557.90	285.63	1,519.83
IPC2	16,000	44,405.32	12,450.29	250.00	1,707.25

Table 7
Storage performance of the existing algorithms using real databases.

Real DBs	Original filters	Aggregate bit vector	HyperCuts	Independent sets	RFC	Controlled cross-producting
ACL1	752	296.34	29.45	8.56	497.62	417.72
FW1	269	263.73	33.41	4.05	1,094.55	262.11
IPC1	1,550	331.46	142.18	23.70	8,984.18	879.00

Table 8
Speed performance of the existing algorithms using real databases.

Real DBs	Aggregate bit vector		HyperCuts		Independent sets		RFC		Controlled cross-producting	
	AMA	WMA	AMA	WMA	AMA	WMA	AMA	WMA	AMA	WMA
ACL1	35.13	44	14.79	22	25.94	31	11	11	13.02	15
FW1	20.49	30	21.37	46	27.26	34	11	11	9.90	14
IPC1	26.14	50	22.44	49	51.86	81	11	11	15.29	19

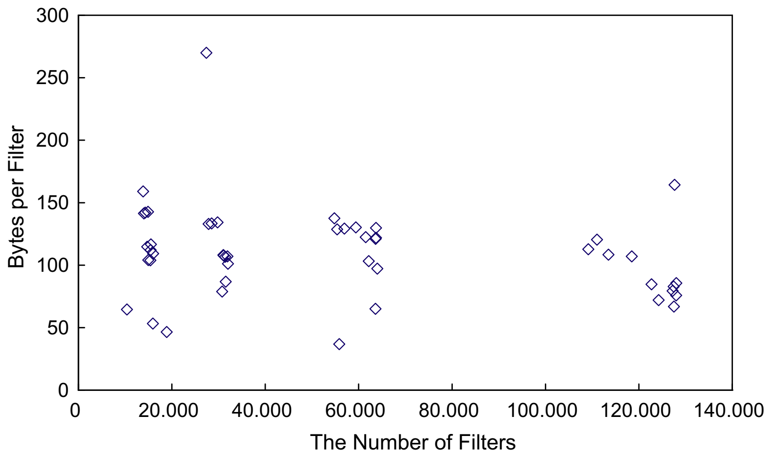


Fig. 8. Storage performance of the proposed scheme using bytes per filter.

Table 10
Speed performance of the existing algorithms using synthetic databases.

Syn DBs	Aggregate bit vector		HyperCuts		Independent sets		Controlled cross-producting	
	AMA	WMA	AMA	WMA	AMA	WMA	AMA	WMA
ACL1	33.30	44	21.07	56	21.72	26	10.69	13
ACL2	36.55	50	21.72	132	36.98	45	8.47	11
ACL3	52.55	84	22.78	119	67.39	102	18.70	27
ACL4	50.00	84	21.78	92	64.48	101	18.46	28
ACL5	33.72	57	28.28	60	58.04	107	13.55	15
FW1	47.33	56	22.52	221	45.29	52	9.05	10
FW2	33.28	34	21.12	43	28.34	30	2.78	3
FW3	51.71	61	23.47	201	44.73	51	8.07	9
FW4	53.47	87	24.22	639	53.45	65	9.85	11
FW5	51.65	61	23.95	182	46.95	56	9.08	10
IPC1	44.05	65	22.44	87	51.39	80	14.79	23
IPC2	32.34	33	17.14	41	25.16	27	2.73	3

HyperCuts. In addition, the proposed scheme also surpasses both schemes in search performance, as shown in Table 10. Although the proposed scheme needs more storage than the *Independent Sets* scheme, it seems that the extra cost could be offset by better search performance. The experimental results of *RFC* are not listed since the table construction would consume more than 4 GB of memory and cannot be completed on our 32-bit machine. Although a 64-bit machine might be able to complete the construction procedure, we believe that the required storage for *RFC* is unlikely to be acceptable even with the state-of-the-art hardware.

In sum, the proposed scheme could significantly improve the storage efficiency with moderate speed degradation as a trade-off. By combining the advantages of the *Cross-producting* scheme and *Independent Sets*, the proposed scheme achieves a better balance between speed and storage performance.

5.3. Scalability evaluation

To further investigate performance variation with respect to the size of databases, we use *ClassBench* to generate larger databases from 32 K to 128 K[3] with different characteristics. The performance metrics include bytes per filter (BpF) and the number of memory accesses in the average and worst case. BpF is equal to the total storage space divided by the number of filters. Since this experiment evaluates the proposed scheme with databases of different sizes, BpF could provide a better judgment on the storage performance. These metrics are divided into four groups, and the experimental results for each group are shown in Figs. 8 and 9.

As shown in Fig. 8, the values of BpF vary from 37 to 270 for filter databases with from 16 K to 128 K filters. In fact, the BpF values are smaller than 165 bytes in most cases, and, as a result, the required storage of the proposed scheme does not incur any exponentially increased storage as the original cross-producting scheme or *RFC* does. Moreover, the average BpF value of the proposed scheme is only 96 bytes even when the sizes of filter databases increase to more than 100 K filters. If the number of distinct field specifications boosts, the filters in independent sets are multi-

[3] The maximum number of filters generated by *ClassBench* is bounded by 130 K.

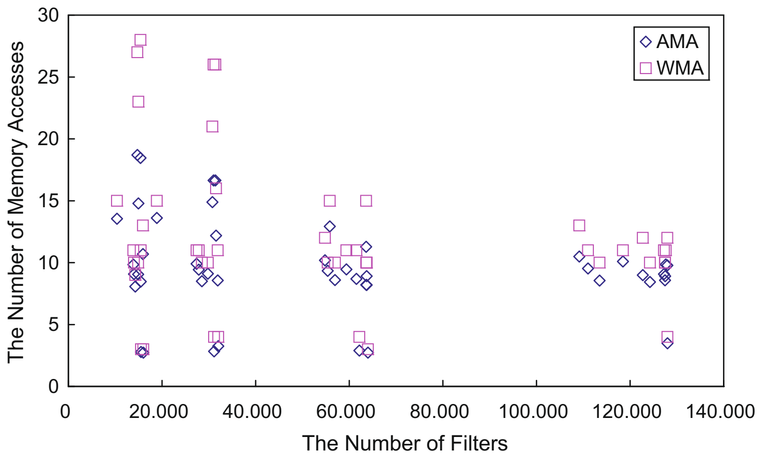


Fig. 9. Storage performance of the proposed scheme using the number of memory accesses.

plied as well, so that the size of the cross-product table could remain at an acceptable level. Therefore, the proposed scheme could preserve the required storage without being affected by the increased number of filters.

Fig. 9 shows the search performance of varying-size filter databases. Apparently, the number of memory accesses is not influenced by the number of filters. In addition, the difference of memory accesses between the worst case and average case gets less significant when the database size increases. This is because, with more filters, the percentage of the best matching prefixes corresponding to the filters of independent sets is usually increased as well. Furthermore, the proposed scheme could exploit the increasing number of filters by properly selecting the filters of independent sets. Therefore, the search performance in the worst case improves.

In our scalability evaluation, the experimental results have shown that our scheme could achieve consistent storage requirement and search performance for 48 different filter databases. Although the storage performance of the proposed scheme may not compete with some existing schemes with small filter databases, the side effect disappears when the number of filters increases. Overall, the proposed scheme could provide superior scalability with respect to varying sizes and characteristics of filter databases.

6. Conclusions

In this work, we have developed a compound scheme for scalable packet classification. The scheme uses linear search to improve the storage requirement of the original cross-producting scheme. Such combination comes from the observation that each existing scheme would perform better if the filter databases exhibited certain characteristics. We found that the original cross-producting scheme and the independent sets scheme complement each other according to their performance on different filter databases. In contrast to the performance divergence, they share the same search procedure of one-dimensional searches. Therefore, we are inspired to propose the new scheme, *Controlled Cross-producting*. Furthermore, we address the problem of supporting incremental updates in our data structures and present new approaches that do not insert a large number of new entries.

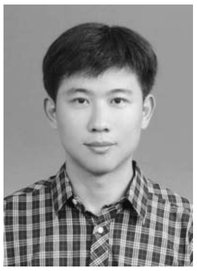
In our experiments of software implementations, we have demonstrated the scalability of our scheme in terms of speed and space with respect to filter databases of varying sizes. Our results show that each packet classification takes less than 20 memory accesses on average and 30 memory accesses in the worst case for large filter databases with more than 10 K filters. As compared to the existing schemes, the proposed scheme better balances speed and storage performance. Therefore, the proposed scheme is suitable for network applications with numerous filters or multifunction network devices that integrate routers, firewalls and network intrusion detection systems (NIDS). We believe that our proposed scheme will remain up-to-date with new network applications due to its insensitivity to filter databases with different characteristics.

For future work, we intend to implement our scheme based on a commodity field program array gate platform. Packet classification still plays an important role in supporting new IPv6 applications, such as flow labeling, encapsulating and authentication. We are collecting real-world IPv6 filter databases for further demonstration, though such databases are still quite rare to date. Finally, we believe that such a novel combination of complementary algorithms might be a new direction to achieve better performance in practice.

References

[1] A.W. Moore, D. Zuev, Internet traffic classification using Bayesian analysis techniques, in: Proceedings of ACM SIGMETRICS '05, 2005, pp. 50–60.
[2] P. Gupta, N. McKeown, Algorithms for packet classification, IEEE Network Magazine 15 (2) (2001) 24–32.
[3] D.E. Taylor, Survey and taxonomy of packet classification techniques, ACM Computing Survey 37 (3) (2005) 238–275.
[4] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, in: Proceedings of ACM SIGCOMM '98, 1998, pp. 191–202.
[5] P. Gupta, N. McKeown, Packet classification on multiple fields, in: Proceedings of ACM SIGCOMM '99, 1999, pp. 147–160.

[6] T. Lakshman, D. Stidialis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, in: Proceedings of ACM SIGCOMM '98, 1998, pp. 203–214.
[7] T. Woo, A modular approach to packet classification: algorithms and results, in: Proceedings of IEEE INFOCOM '00, 2000, pp. 1213–1222.
[8] V. Srinivasan, G. Varghese, S. Suri, Packet classification using tuple space search, in: Proceedings of ACM SIGCOMM '99, 1999, pp. 135–146.
[9] A. Feldmann, S. Muthukrishnan, Tradeoffs for packet classification, in: Proceedings of IEEE INFOCOM '00, 2000, pp. 1193–1202.
[10] M.M. Buddhikot, S. Suri, M. Waldvogel, Space decomposition techniques for fast layer-4 switching, in: Proceedings of IFIP Sixth International Workshop on High Speed Networks, 1999, pp. 25–42.
[11] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, in: Proceedings of Hot Interconnects VII, 1999.
[12] F. Baboescu, S. Singh, G. Varghese, Packet classification for core routers: Is there an alternative to cams? in: Proceedings of IEEE INFOCOM '03, 2003, pp. 53–63.
[13] F. Baboescu, G. Varghese, Scalable packet classification, in: Proceedings of ACM SIGCOMM '01, 2001, pp. 199–210.
[14] E. Spitznagel, D.E. Taylor, J.S. Turner, Packet classification using extended tcams, in: Proceedings of IEEE International Conference on Network Protocols (ICNP '03), 2003, pp. 120–131.
[15] J. van Lunteren, T. Engbersen, Fast and scalable packet classification, IEEE Journal on Selected Areas in Communications 21 (4) (2003) 560–571.
[16] H. Che, Z. Wang, K. Zheng, B. Liu, Dres: dynamic range encoding scheme for tcam coprocessors, IEEE Transactions on Computers 57 (7) (2008) 902–915.
[17] A. Bremler-Barr, D. Hendler, Space-efficient tcam-based classification using gray coding, in: INFOCOM 2007, 26th IEEE International Conference on Computer Communications, IEEE, 2007, pp. 1388–1396.
[18] A. Bremler-Barr, D. Hay, D. Hendler, B. Farber, Layered interval codes for tcam-based classification, in: SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM, New York, NY, USA, 2008, pp. 445–446.
[19] K. Lakshminarayanan, A. Rangarajan, S. Venkatachary, Algorithms for advanced packet classification with ternary cams, SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, NY, USA, 2005, pp. 193–204.
[20] H. Liu, Efficient mapping of range classifier into ternary-cam, in: Hot Interconnects X, 2002, pp. 95–100.
[21] G.V. Sumeet Singh, Florin Baboescu, J. Wang, Packet classification using multidimensional cutting, in: Proceedings of ACM SIGCOMM '03, 2003, pp. 213–224.
[22] D.E. Taylor, J.S. Turner, Scalable packet classification using distributed crossproducting of field labels, in: Proceedings of IEEE INFOCOM '05, 2005, pp. 269–280.
[23] X. Sun, S.K. Sahni, Y.Q. Zhao, Packet classification consuming small amount of memory, IEEE/ACM Transactions on Networking 13 (5) (2005) 1135–1145.
[24] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable high speed ip routing lookups, in: Proceedings of ACM SIGCOMM '97, 1997, pp. 25–36.
[25] P.-C. Wang, C.-T. Chan, S.-C. Hu, C.-L. Lee, W.-C. Tseng, High-speed packet classification for differentiated services in ngns, IEEE Transactions on Multimedia 6 (6) (2004) 925–935.
[26] P.-C. Wang, C.-L. Lee, C.-T. Chan, H.-Y. Chang, Performance improvement of two-dimensional packet classification by filter rephrasing, IEEE/ACM Transactions on Networking 15 (4) (2007) 906–917.
[27] V. Srinivasan, A packet classification and filter management system, in: Proceedings of IEEE INFOCOM '01, 2001, pp. 1464–1473.
[28] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, 1972, pp. 85–103.
[29] B. Lampson, V. Srinivasan, G. Varghese, Ip lookups using multiway and multicolumn search, IEEE/ACM Transactions on Networking 7 (4) (1999) 323–334.
[30] M.E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, A.T. Campbell, Directions in packet classification for network processors, in: Proceedings of Second Workshop on Network Processors (NP2), 2003.
[31] H. Liu, Routing table compaction in ternary cam, IEEE Micro 22 (1) (2002) 58–64.
[32] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transctions on Computer Systems 17 (1) (1999) 1–40.
[33] H. Lu, S. Sahni, O($\log w$) multidimensional packet classification, IEEE/ACM Transactions on Networking 15 (2) (2007) 462–472.
[34] E.W.B. Miguel, A. Ruiz-Sanchez, W. Dabbous, Survey and taxonomy of ip address lookup algorithms, IEEE Network Magazine 15 (2) (2001) 8–23.
[35] H. Song, Design and evaluation of packet classification systems, Ph.D. Thesis, Department of Computer Science and Engineering, Washington University, 2006.
[36] D.E. Taylor, J.S. Turner, Classbench: a packet classification benchmark, in: Proceedings of IEEE INFOCOM '05, 2005, pp. 2068–2079.



Pi-Chung Wang (M'02/ACM'06) received the M.S. and Ph.D. degrees in Computer Science and Information Engineering from the National Chiao Tung University in 1997 and 2001, respectively. From 2002 to 2006, he was with Telecommunication Laboratories of Chunghwa Telecom, working on network planning in broadband access networks and PSTN migration. During these four years, he also worked on IP lookup and classification algorithms. Since February 2006, he has been an assistant professor of Computer Science at National Chung Hsing University. His research interests include IP lookup and classification algorithms, scheduling algorithms, congestion control, network processors, algorithms and applications related computational geometry. He is currently working on high speed string matching for network intrusion detection.