MIT LCS Technical Memo: MIT-LCS-TM-637

Scalable Packet Classification Using Bit Vector Aggregating and Folding

Ji Li, Haiyang Liu, Karen Sollins
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jli, hyliu, sollins}@ mit.edu

April, 2003

***Abstract*--- Packet classification is a central function for a number of network applications, such as routing and firewalls. Most existing algorithms for packet classification scale poorly in either time or space when the database size grows. The scalable algorithm Aggregated Bit Vector (ABV) is an improvement on the Lucent bit vector scheme (BV), but has some limitations. Our algorithm, Aggregated and Folded Bit Vector (AFBV), seeks to reduce false matches while keeping the benefits of bit vector aggregation and avoiding rule rearrangement. It combines bit vector aggregation and folding to achieve this goal. Experiments showed that our algorithm outperforms both the BV and ABV schemes in synthetically generated databases.**

***Keywords*--- packet classification; scalability; bit vector; aggregation; rule rearrangement; folding**

I. INTRODUCTION

Packet classification puts packets into classes based on their headers. There are several network services that require packet classification, such as policy-based routing, service differentiation and firewall management. For these services, it is necessary to determine which rule and action to apply to an arriving packet in order to decide which action to take with respect to the packet. For example, for an dubious packet arriving at a firewall, the corresponding action may be to discard the packet or trace back to the source.

Although the processing associated with these rules may vary according to services, all packets arriving at the service point need to be classified into equivalent classes based on their headers. The classification is performed by a packet classifier. The packet classifier has a rule database, in theory, one for each type of packets with different headers. In this paper, we focus on how to classify packets and not the actions taken for each class.

There usually are some requirements in processing a packet. For instance, in a router, forwarding at wire speed requires forwarding minimum sized packets within the time it takes to arrive on a link. There are two reasons that the processing speed is crucial. First, the router may not be able to provide the promised service to applications such as real-time video without satisfying the minimum processing speed requirement. Second the router may have to drop packets due to buffer overflow. Consequently, the processing speed is the most important metric for packet classification. As memory access is one of the most time-consuming operations in the processing, the number of memory accesses becomes the most important criterion in evaluating the efficiency of a packet classification algorithm.

In the initial stages of Internet, most rule databases were used for firewall management, and the database size was relatively small, with no more than a few thousand rules [5]. Subsequently, the introduction of differential service required routers to classify packets into several distinct classes based on the IP headers. For a core router, this is likely to result in a large rule database containing about ten thousand rules. With the development of the Internet, the size of rule databases can only grow with time. Therefore, we conclude that scalability is also an important feature for packet classification algorithms.

In this paper we propose the Aggregated and Folded Bit Vector algorithm, a scalable algorithm for packet classification. Our algorithm combines aggregating and folding to reduce both false matches and the number of memory accesses. This contrasts the Aggregated Bit Vector algorithm [1], which uses bit vector aggregation and rule rearrangement to classify packets. Experiments show that our algorithm outperforms ABV in synthetically generated databases.

The rest of the paper is organized as follows. In section 2 we introduce previous work on packet classification and their scalability. Section 3 describes the Lucent bit vector scheme and the ABV in detail since our AFBV algorithm borrows several ideas from them. Section 4 proposes our AFBV in an attempt to address the shortcomings of the rule rearrangement, which avoids the cost of rule rearrangement while keeping the benefit of aggregating. Experimental results are presented in Section 5, demonstrating measured improvements. Section 6 concludes our work.

II. PREVIOUS WORK

Most previous work in packet classification either takes

linear time to sequentially search through all the rules in the database, or uses a linear amount of parallelism. The simplest algorithm evaluates each rule sequentially until a rule that matches all the headers of an arriving packet is found. Obviously, it is not efficient or scalable: its time complexity is linear to the number of rules in the database.

The Lucent bit vector scheme (BV) [2] is one such approach. The idea in this work is to search for rules that match each field of the packet header first, and represent the result as bit vector of rules that match the packet in the corresponding field. Then the rules that match the header can be obtained by intersecting the bit vectors for all relevant fields. The scheme can facilitate the classifying process by accessing a large number of bits in one memory access in hardware, but it is still a linear scheme and not scalable.

The RFC algorithm [3] proposed by Gupta and McKeown is a multi-stage classification algorithm. It is based on the observation that the classifiers contain considerable structure and redundancy that can be exploited. But it consumes much storage when there are several fields in the rule database.

Srinivasan et al [11] described grid-of-trie and cross-producting for solving the least cost matching filter problem. Grid-of-tire is specialized for two fields in the packet header, and achieves logarithmic time and linear storage. However, their algorithm is limited to two fields in the packet header.

Based on the observation that set bits are very sparse in most bit vectors, Baboescu and Varghese [1] introduced the novel notions of bit vector aggregation and rule rearrangement in their ABV algorithm to make the Lucent bit vector scheme scalable. This scheme uses aggregation to compress bit vectors and rule rearrangement to reduce false matches caused by bit vector aggregation. However, the rule rearrangement leads to several problems, which, we believe, can be avoided without reducing the benefits of bit vector aggregation. We will discuss it in detail in the following sections.

In the next sections we investigate the Lucent bit vector scheme and the ABV scheme. We follow this with the design of a novel scheme to reduce false matches and thus memory accesses – the Aggregated and Folded Bit Vector algorithm.

III. LUCENT BV AND ABV

We introduces our new scheme by first describing the Lucent bit vector algorithm and the ABV algorithm as our algorithm inherits their basic ideas of bit vector and aggregation.

A. Lucent bit vector algorithm

A rule database usually consists of a sequence of rules, each of which consists of *k* values corresponding to *k* fields in the packet header respectively. The rule with lower order has higher priority. Table I shows a simple rule database with only 9 rules with 2 fields per rule.

The Lucent bit vector algorithm divides the matching problem into several independent sub-matching problems, one for each field. That is, for each field of an incoming packet, the Lucent bit vector algorithm searches the corresponding field of rules in the rule database and finds all the rules that match that field. Subsequently, all the results are combined together to find the first rule that matches all the fields of the packet.

The Lucent bit algorithm can be implemented with a trie structure. A trie is constructed for each field in the rule database to show the rules that match a prefix. Each node in the trie denotes a prefix and is associated with a bit vector. The prefix is specified by the path from the root to the current node in the trie. The length of each bit vector is equal to the number of rules in the database, and a set bit in the vector means that the rule in this position matches the current prefix. When a packet arrives, the bit vector for each of its fields is obtained by searching the trie for this field under the packet's field in the header. After obtaining all bit vectors, and intersecting them, the first one in the resulting bit vector is the position of the first rule in the database that matches the packet header. Fig.1 shows the two tries constructed according to the two fields in

TABLE I. A RULE DATABASE EXAMPLE

Rule	Filed 1	Field 2
R0	00*	00*
R1	00*	01*
R2	10*	11*
R3	11*	10*
R4	0*	10*
R5	11*	11*
R6	0*	0*
R7	10*	01*
R8	1*	01*

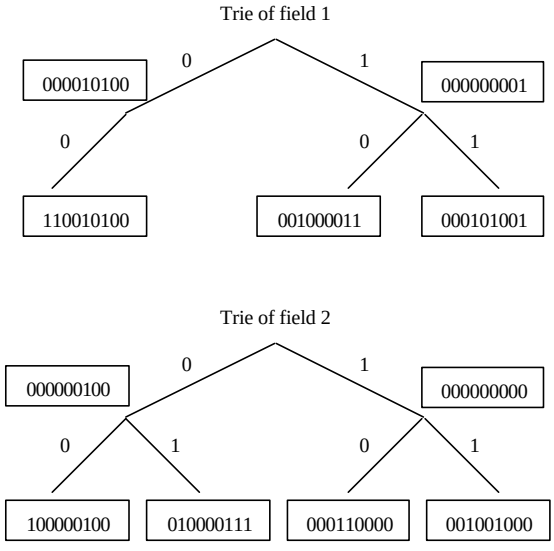

Figure 1. Lucent BV tries associated with the rule database in Table I

Table I.

The Lucent bit vector is a linear scheme, but it provides a good idea for further improvement.

B. ABV algorithm

In the ABV algorithm the authors made two major observations: firstly that the set bits in bit vectors are sparse, and secondly that a packet matches only a few rules. The two key ideas were introduced in ABV to take advantage of these two observations: rule aggregation and rule rearrangement.

The purpose of rule aggregation is to construct a reduced size bit vector that captures partial information from the whole bit vector. This allows us to guess whether a rule applies without comparing the bits in the original bit vectors. Furthermore, the construction of the aggregated bit vector should be efficient. In ABV, an aggregation size *A* is selected to optimize the performance of the algorithm. This means that a new aggregated bit vector is defined containing one bit for each group of *A* bits in the original bit vector. The aggregate bit will be set if any of the *A* bits in the original bit vector are set to 1; otherwise the bit will be cleared. Aggregation can be performed recursively.

While aggregation does often reduce the number of memory accesses, it also leads to false matches, as we cannot determine which bit or bits in the original bit vector have led to a 1 in the aggregated bit vector. The worst case occurs when a false match occurs for every aggregate bit.

To reduce the probability of false matches, the rules are rearranged before aggregation so that multiple rules matching a specific prefix are placed close to each other. This is done by first sorting the rules according to one field. Then the rules are grouped together according to the length of prefixes within that field in a non-decreasing order. Next, rules are sorted by prefix value within each group, thereby grouping together all filters with the same prefix value. Subsequently, the rules are sorted by the other fields in each group with the same prefix. This process is repeated recursively.

Table II shows the database in Table I after rule rearrangement. Fig.2 shows the new tries obtained from Table II. The aggregate bit vectors are shown before the original but sorted bit vectors.

IV. AFBV:

AGGREGATED AND FOLDED BIT VECTOR ALGORITHM

The bit vector aggregation ideally can create a logarithmic algorithm, which is more scalable than the Lucent Bit Vector algorithm. Realizing vanilla bit vector aggregation performs poorly when there are a large number of false matches, Varghese and Baboescu [1] tried to manipulate bit vectors with rule rearrangement, which suffers various drawbacks in the following.

- *Expensive "all-match" rule matching*. Rule rearrangement makes it necessary to find all matching rules and choose the one with the lowest cost. This property does not affect the number of memory access in the worse case, but significantly increases the average cost, which, in turn greatly reduces the classifier throughput. If we can avoid rule rearrangement, we will only need to do first match, as compared to all-match.
- *Large variance in performance*. ABV provides a simple sorting method. But this sorting scheme separates prefixes with their sub-prefixes. For example, 0* and 000* will be separated if there is a 1*. It is hard to find a systematic way to choose a generally effective rule rearrangement method. Furthermore, it is important yet difficult to determine the order of fields to sort by in a large rule database with several fields. And with more fields, it is hard to tell how such sorting will affect the performance. Therefore, the performance can vary greatly, depending on the rule

TABLE II. THE DATABASE AFTER RULE REARRANGEMENT

Rule	Filed 1	Field 2
R6	0*	0*
R4	0*	10*
R8	1*	01*
R0	00*	00*
R1	00*	01*
R7	10*	01*
R2	10*	11*
R3	11*	10*
R5	11*	11*

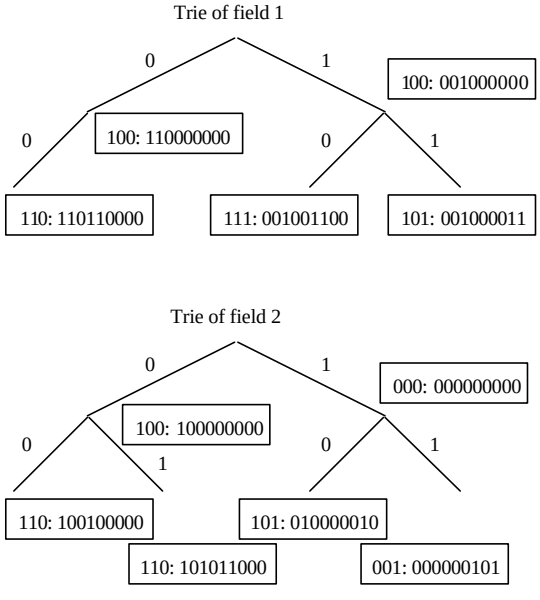

Figure 2. ABV tries associated with the rearranged rule database in Table II (Aggregate size = 3)

databases and the way that rules are rearranged.

- *Rule mapping back cost*. The mapping from rules in the original order to the rearranged one must be pre-computed. After all the matches are found, matched rules must be mapped back to the original order so as to find the one with the lowest cost (the rule with the lowest order). This requires additional memory accesses, and is expensive if there are a number of matches.
- *Preprocessing / update cost.* Although rule rearrangement is supposed to be done during the preprocessing, it is still quite a burden for a large rule database. Moreover, it makes rule database update much more expensive since the rule order has been changed by rearrangement.

The above problems motivate us to design a new scheme that avoids rule rearrangement but still keeps the benefit of bit vector aggregation.

In this section we propose a novel approach to packet classification that makes use of bit vector aggregation and folding. Our scheme inherits the bit vector and trie structure from the Lucent bit vector scheme and the bit vector aggregation from the ABV scheme. Our scheme avoids the drawbacks suffered by rule rearrangement by using multiple aggregated bits to reduce false matches.

We use a hash function to compress bits from the original vectors effectively into a small number of bits while still retaining a sufficient amount of information pertaining to the set bit position from the original vectors because only those set bits are important. This allows us to filter out a large portion of false match cases, which cannot be detected by a single-bit aggregation without accessing the original bit vector. Second, hashing allows us to trace back to the original set-bit positions easily from the hash bits. This property reduces the number of memory accesses when there is a real match. Last but not least, the hash function will never miss a real match. We investigate bit vector folding as the hash function in this paper because it satisfies all the three criteria described above, and can be executed at negligible cost.

A. Bit vector folding

In the ABV algorithm, A (the aggregation size) bits in the original bit vector are aggregated to one bit, indicating whether there is any set bit in the aggregate group of the original vector. In our AFBV algorithm, in addition to the bit, another f (the folding size) bits are used to store the original bit vector in a folded form. The f bits are generated by bit-wise OR-ing f groups of bits in the A bits. In other words, bits in the original aggregation group whose position has the same mod-f value, say j, are OR-ed together and occupy the position j in the folded vector. (If the last fold has less than f bits, zero padding is used to extend the bit vector to exactly f bits).

This scheme uses f bits (f is much smaller than A) to store set-bit information of an A-bit vector. Only set bits at positions with the same mod-f value can cause false matches. In ABV, set bits at any position can cause false matches. If there is a match with the folded vector at position k ($0 = k = f - 1$), we know that the real match positions in the original bit vector must take form of ($i \cdot f + k$). Finally if there is a real match in the aggregation group, the folded vector of this group will definitely generate a match, and thus no real match would be missed using our folding scheme. This property guarantees the correctness of our AFBV algorithm.

A design option is the granularity of the folding group, i.e., the folding range. By now the folding is performed with one aggregation group, but the folding range can be a whole bit vector (*global folding*), a single aggregation group (*local folding*), or multiple aggregation groups (*regional folding*). The choice on the granularity depends on the computational complexity and the requirement on storage usage. If there is a strict limitation on the storage, a global folding may be a better choice. Otherwise, single folding or double folding is better since they use smaller folding sizes and thus require less memory access in each match of the aggregate bit vectors. For example, if the length of a bit vector is 2048 and a global folding size is 64, the storage used by global folding is only 1/32 (about 3%) of the original bit vector; if the aggregation size is 128 and the local folding size is 16, the storage used by the folding is 1/8 (12.5%) of the original bit vector. We can also cut down the storage usage in the local folding scheme by using a smaller folding size, but this also cuts down the effect of the folding since it now carries too little information. However, in global folding, the 64 bits must be accessed in each matching process while in local folding only 16 bits need to be fetched from the storage. Furthermore, the large folding size in global folding needs more computation to locate the matching positions. Regional folding provides a balance between the two extremes. The performance comparison of the three methods is shown in the next section.

The aggregation can be viewed as a special case of folding size 1 and then the folding range is equal to the aggregation size, as compared with the more general case of folding size f in AFBV. Meanwhile, in global folding if the folding size is equal to the aggregation size, then folding and aggregation is orthogonal to each other. So their combination provides much information on the original bit vector.

B. Multiple Folding

Besides granularity, the number of foldings is another design issue. For one folding group, we can use single folding (one folded bit vector for each folding range), double folding (two folded bit vectors for each folding range), or even more folded bit vectors. The more the number of folding sizes, the more information we can carry. Again, our discussion is under the same amount of storage. Subsequently, we can use one large folding size for a single folding and smaller folding sizes for multiple folding.

The advantage of multiple folding is that it becomes increasingly difficult for false matches to survive with multiple folded vectors that store the same set-bit position information using different folding sizes. This property can help to locate real match positions without accessing all bits in the original vector. With multiple folded vectors, the possible match positions can be narrowed further. The following sections show how bit vector multiple folding scheme helps filter out false matches and locate real match positions.

1) Detecting False Matches

Fig.3 shows two bit vectors (corresponding to the same aggregation bits of two fields) with the folding length 48 bits.

The example in Fig.3 shows that there are several set bits in the original bit vector of for Field 1 and Field 2. Therefore, if the aggregation size is 48 bits, the intersection of the two aggregation bit vector would indicate there might be a match in the original bit vector. In AFBV, before checking the original 48 bits, we turn to the folded vectors first. Note that the folded vector 1 with $f = 7$ has no match and folded vector 2 with $f = 9$ has three matches. Since a real match would have matches in folded vectors no matter how the folding is done, we can conclude that there is no match in the original bit vectors. In this case, we only need to fetch the 7-bit folded vector to conclude it is a false match instead of accessing the original all 48 bits as required in BV and ABV. In multiple folding, we can conclude a false match if any of the folding bit vectors, in this case, 7 or 9, does not have a match.

2) Locating real match positions

Single folding can locate the match positions, but multiple folding can greatly increase the locating accuracy. The more the number of folding, the more accurate we can determine the possible matching positions. But under the same bit vector storage, a larger number of folding imply a smaller folding size, hence, we need to reach a balance between the folding size and the number of folding.

In Fig.4, both folded vectors have matches at positions 1 and 4 in folded vector 1, positions 4 and 6 in folded vector 2. The positions in the original bit vector 1 that satisfy the result are {1, 8, 15, 22, 29, 36, 43} for position 1 in the folded bit vector 1 (mod 7 = 1), and {4, 11, 18, 25, 32, 39, 46} for position 4 (mod 7 = 4), so the first set S1 = {1, 4, 8, 11, 15, 18, 22, 25, 29, 32, 36, 39, 43, 46}. Similarly, the positions in the original bit vector 2 are {4, 13, 22, 31, 40} for position 4, and {6, 15, 24, 23, 42} for position 6, so the second set S2 = {4, 6, 13, 15, 22, 23, 24, 31, 40, 42}. The intersection of the two sets is S = {15, 22}. So only positions 15 and 22 at the original vector might have real matches. Therefore, in AFBV only the bytes that contain bits 15 and 22 need to be fetched to verify if there are real matches. In this example, position 15 is the real match we are looking for.

To locate real match positions, global folding provides more information than local folding and regional folding, assuming that the storage for each folding scheme is constant. For example, suppose that the bit vector is 512 bits, the aggregation size is 128 bits, and the memory for the folding is 32 bits. Then the aggregated bit vector is 4 bits. In global folding, the 32 folding bits are used for the whole bit vector while in local folding each aggregation group has 8 bits to carry the information. If two aggregated bit vectors are 1100 and 1010, then the intersection result is 1000, so only the first 128 bits in the original bit vector may have real matches. In global folding, 32 bits are used to carry information pertaining to the 128 bits while in local folding there are only 8 bits. Therefore, global folding gives more information to locate the matching positions than local folding, as the intersection of the aggregation groups reduces the possible matching positions for global folding. However, as we mentioned before, more bits are fetched each time in global folding, which reduces the performance in case of many false matches in the aggregate bit vectors.

C. AFBV algorithm

We use local folding in this section to show how the algorithm works. In the preprocessing phase, the trie structures are constructed, as shown in Fig.5. The number of rules in the database is 9, the aggregate size is 9, the folding range is also 9, and the folding sizes are 2 and 3. The first line in each node is the aggregate bit vector, folded vector 1 and folded vector 2, respectively. The second line is the original bit vector.

Note that not all the bit vectors are necessary. Only those bit vectors in which their corresponding nodes that do not have two children need to be stored. For example, in Fig.1 the bit vector for 1* in the trie of field 1 is not necessary because the packet header will not stop at this node but go down the trie. Similarly, the bit vectors of 0* and 1* in the trie of field 2 are not necessary either. In this way we can save some storage.

AFBV works in the following steps:

1. Check two aggregate bit vectors to find the aggregation bits where there might be matches in the original bit vector they represent. This is the same as ABV.

2. If matches are found in the aggregate bit vectors, check the corresponding folded vectors to see if there is a match. If there are matches, go to step 3; otherwise there is no match in

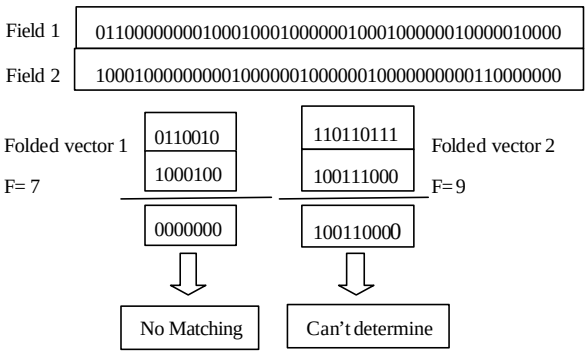

Figure 3. Multiple folding: Detecting false matches

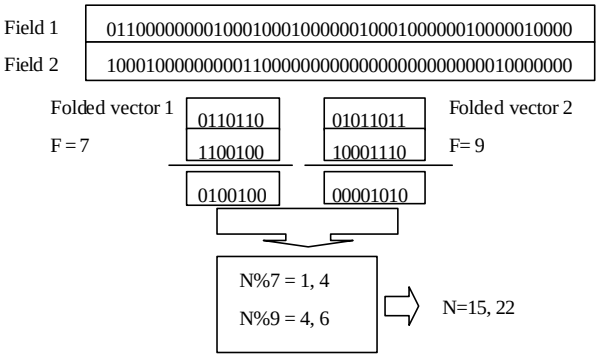

Figure 4. Multiple folding: Locating matching positions

the original bit vectors.

3. Compute all the possible match positions using the matching positions in folded vectors, as shown in Fig.4.

4. Fetch bits from the original bit vectors starting from the lowest possible match position obtained in step 3, and check for is a real match. Only the first match position needs to be found.

AFBV offers the following benefits:

- Only the first match needs to be found, unlike ABV, where all-match is required. This helps to achieve much smaller average memory accesses in AFBV as compared to ABV, especially when there are a number of matching rules for the incoming packet.
- AFBV is scalable in terms of the number of matching rules. When the number of matching rules increases, the rule mapping back required by rule rearrangement performs poorly. Since AFBV only needs to find the first match, the increased number of matching rules does not degrade its performance.
- No complex pre-processing of rule rearrangements is needed. This also speeds up bit vector update when rule database changes. Rule rearrangement requires sorting all the rules, whose complexity is $O(N^2)$.
- In addition to detecting false matches, folded vectors can narrow the possible match positions to a small range, and thus effectively reduce the number of memory accesses for fetching the original bits.
- ABV cannot afford a large aggregation size, because the greater the size, the higher the chances of false matches and the greater the cost of detecting false match. A large aggregation size can be used in AFBV since it has the folded vectors to filter out false matches to some extent.

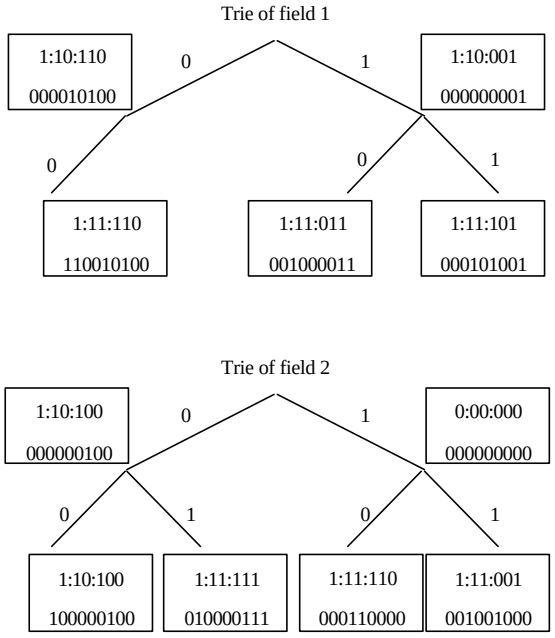

Figure 5. AFBV tries associated with the rule database in Table I (Aggregate size = 9, Folding range = 9, Folding size = 2, 3)

V. Performance Evaluation

A. Experimental Platform

For purposes of simulation, we assume the machine word size is 32 bits.

The experiment is based on several synthetic rule databases. The rule databases have only two fields. Their sizes are: 1024 rules, 2048 rules, 4096 rules, 8192 rules and 10240 rules. The prefix distribution in each field length is shown in Table III. In this table, *32* represents the length of IP address, and the prefix length distribution is based on the observation of the real rule databases in BGP described in [1]. *16* represents the length of port number, *8* represents the length of the protocol type, and their prefix length are evenly distributed. The databases are randomly generated.

B. Storage requirement comparison

Suppose that the number of rules in a database is R, the number of bit vectors in the trie is N, the aggregate size of ABV is A_{abv}, the aggregate size of AFBV is A_{afbv}, the sum of the folding sizes $F_1, F_2, \ldots, F_k$ in ABFV is F, and the folding range is B ($B = m \cdot A_{afbv}$, $1 = m = R / A_{afbv}$).

The storage requirement of ABV bit vectors is:

$S_{abv} = N \cdot R + N \cdot R / A_{abv}$,

The storage requirement of AFBV bit vectors is:

$S_{afbv} = N \cdot R + N \cdot R / A_{afbv} + N \cdot F \cdot R / B$

$= N \cdot R + N \cdot R / A_{afbv} + N \cdot F \cdot R / (m \cdot A_{afbv})$

So the storage ratio of the two algorithms is:

$T = S_{afbv} / S_{abv} = (1 + (1 + F / m) / A_{afbv}) / (1 + 1 / A_{abv})$, which depends on the parameters only, not the number of rules in the database.

When $m = R/A_{afbv}$, $T = (1 + 1 / A_{afbv} + F / R) / (1 + A_{abv})$, which also depends on the number of rules in the database besides those parameters.

In Table IV, A_{abv} is fixed to the word size of 32 bits. m is the number of aggregation groups that a folding range covers. When the folding range is one aggregate group, i.e., $m = 1$, about 20% additional storage is required when $A_{afbv} = 4 \cdot F$, and less than 10% is required when $A_{afbv} = 8 \cdot F$. When $m = 2$, about 10% additional storage is required when $A_{afbv} = 4 \cdot F$, and less than 5% is required when $A_{afbv} = 8 \cdot F$. Therefore, the AFBV

TABLE III. Prefix Distribution In Synthetic Databases

Field Length	**32**					**16**	**8**
Prefix Length	0-15	16	17-23	24	25-32	Even	Even
Percentage	1%	8%	35%	55%	1%		

algorithm does not require much additional storage compared to the original bit vectors. In AFBV we can use a larger aggregation size than ABV, so we need even less storage than ABV in some cases, especially when a large folding range is used.

C. Performance comparison

We compare the performance of the Lucent BV, ABV and AFBV on the databases with different sizes. The performance is indicated by the number of bytes accessed by each algorithm. We limit the additional storage required by the folded bit vectors to 3%-10% of the whole storage. The experiment result is shown in Fig. 6, Fig.7, Fig.8 and Fig.9. Fig.6 and Fig.7 show the graphs for using a single folding while Fig.8 and Fig.9 show the graphs for double folding. The keys of the figures are explained in Table V. BV is the Lucent bit vector algorithm. ABV-32 and ABV-64 are the aggregated bit vector algorithm with aggregation size of 32 bits and 64 bits respectively. We test AFBV with three folding methods: local folding (AFBV-1-64,8), regional folding (AFBV-2-64,16) and global folding (AFBV-R-64,64).

Fig.6 shows the average performance, which is obtained by

TABLE IV. MEMORY REQUIREMENT COMPARISON

(A_{abv}, A_{afbv}, F)	S_{afbv} / S_{abv}	
	m = 1	m = 2
(32, 64, 8)	1.106	1.045
(32, 64, 16)	1.227	1.106
(32, 64, 32)	1.470	1.227
(32, 128, 8)	1.038	1.008
(32, 128, 16)	1.098	1.038
(32, 128, 32)	1.220	1.098
(32, 128, 64)	1.462	1.220
(32, 256, 8)	1.004	0.989
(32, 256, 16)	1.034	1.004
(32, 256, 32)	1.094	1.034
(32, 256, 64)	1.216	1.095
(32, 256, 128)	1.458	1.216

TABLE V. KEYS IN FIGURE 6 - 9

Parameter / Keys	Aggregation Size	Folding Range	Folding Size	
			Single folding	Double folding
BV	N/A	N/A	N/A	N/A
ABV-32	32	N/A	N/A	N/A
ABV-64	64	N/A	N/A	N/A
AFBV-1-64-8	64	64	8	(3,5)
AFBV-2-64,16	64	128	16	(7,9)
AFBV-R-64,64	64	All	64	(25,39)

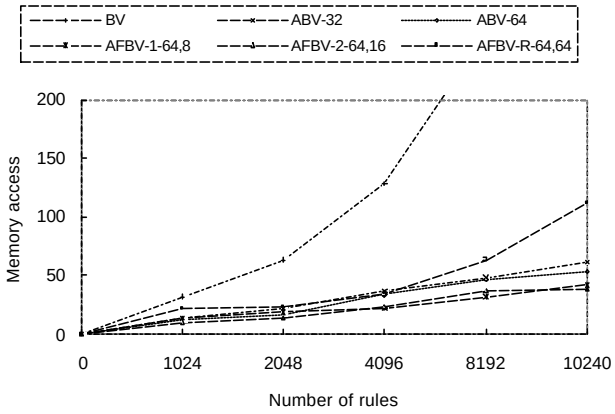

Figure 6. Performance on average in single folding

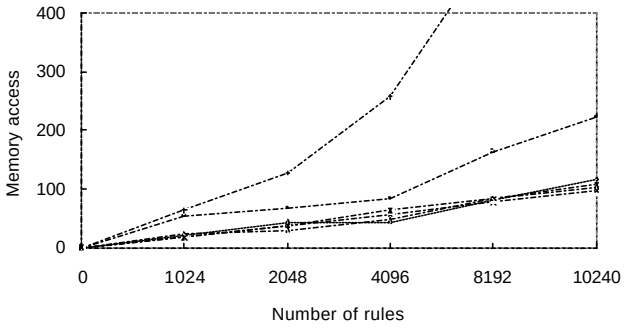

Figure 7. Performance in the worse case in single folding

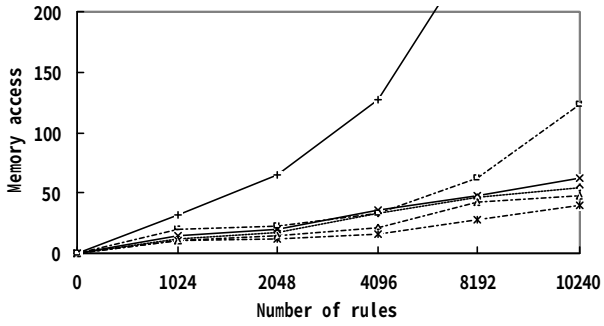

Figure 8. Performance on average in double folding

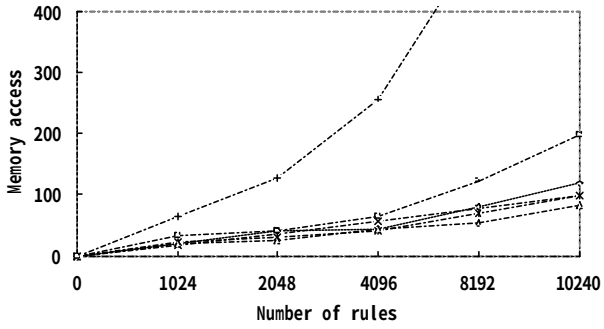

Figure 9. Performance in the worse case in double folding

summing the number of bytes accessed by all the cross products of bit vectors and then dividing the result by the number of cross products. The figure shows that the average performance of local folding and regional folding (m = 2) is much better than that of ABV, but the performance of the global folding (AFBV-R-64,64) is much worse than ABV. This is due to the large folding size of the global folding scheme. In this case, 64 bits are fetched for each match in the intersection of the aggregate bit vectors. As the number of rules increases, local folding and regional folding still work better than ABV, but global folding performs even worse.

Among the three folding methods, global folding has the worst performance. With a big folding range, global folding does not work if the folding size is too small because the folded vectors will be full of set bits; however, if the folding size is large, it requires much memory access to check the folded bit vectors for each match in the aggregate bit vectors. Local folding and regional folding with m = 2 perform similarly, and much better than global folding especially when the database grows larger.

Fig.7 shows the performance in the worst case. In this case, the advantage of the AFBV is not as obvious as Fig.6. Usually the worst case for AFBV happens when there are several contiguous set bits in the original bit vector and the first match position is near the end of the original bit vector. So AFBV does not performs much better than ABV, sometimes even worse. As to the three folding methods, global folding is still the worst due to the same reasons stated above.

AFBV in Fig.8 and Fig.9 performsa little better than that in Fig.6 and 7 under the same storage requirements. This implies that multiple folding does better than single folding in detecting false matches and locating match positions.

D. Parameters discussion

1) Folding size selection

In multiple folding, the folding size is important to performance. The straightforward idea is that two folding sizes should have no common divisor. We test double folding under local folding on a database with 2048 rules. The aggregate size is 128 bits, and the folding range also 128 bits. In this table, (5, 11), (7, 9), (11, 21) perform better than the others with the same sum of folding sizes. This implies that the difference between the couple of sizes should not be too much or too little. The key is to make two folded bit vectors contain unrelated information of the original bit vector.

2) Folding size vs. Folding range

In our algorithm, the folding size and the folding range are the most important parameters. By now we determine these parameters by the storage requirement. In this section, without considering this requirement, we test the relationship between the two parameters by the number of memory access under several combinations of the two parameters on a database with 10240 rules, 2 fields with 16 bits for each field. The schemes are local folding and double folding. The aggregation size is 32 bits. The relationship can be seen clearly from Fig.10 and Fig.11. In Fig.10, when the folding range is 32 bits, the best folding size is 16 bits; when the folding range increases to 64 bits, 16 is still the best size, but the performance difference between the folding sizes 16 bits and 32 bits becomes smaller; when the folding range changes to 128 bits, the best performance is achieved with a folding size of 32 bits; when the folding range reaches 256 bits and 512 bits, 64 bits become the best folding size; when the folding range increases to more than 1024 bits, there is little difference between 8, 16 and 32, because they are insignificant compared to the folding range. The performance in the worse case is similar to that of average

TABLE VI. FOLDING SIZE SELECTION

DATABASE SIZE: 2048 RULES
AGGREGATION SIZE: 128 BITS

Folding size	Average Memory Access	Worst Memory Access
(3, 13)	16	72
(5, 11)	14	64
(7, 9)	14	62
(7, 25)	15	59
(9, 23)	15	44
(11, 21)	14	46
(13, 19)	16	56
(15, 17)	16	42

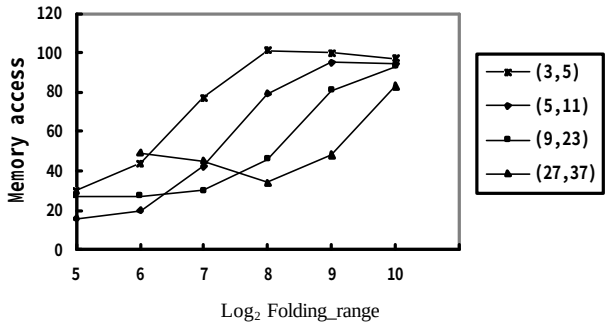

Figure 10. Performance on average

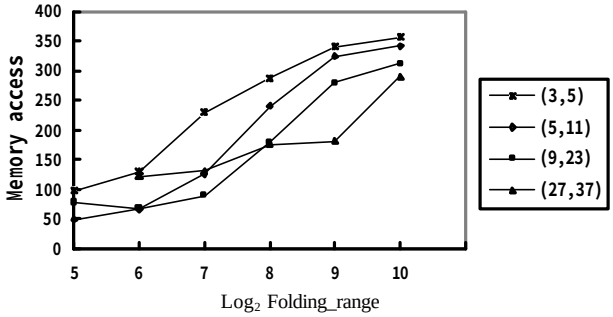

Figure 11. Performance in the worst case

case, shown in Fig.11.

From this experiment, the proper relationship between folding range B and folding size F is: $B = (4\sim8) \cdot F$, which implies about 10-20% additional storage.

VI. CONCLUSIONS

Packet classification is a central function of a number of network activities. This project is addressing the problem of improving performance in the selection of rules for classification of packet headers. There is significant previous work in this area. Our work is an improvement on the ABV scheme, which in turn is an improvement in the linear Lucent bit vector scheme. The Lucent bit vector scheme scales to medium size databases. ABV introduces the concepts of bit vector aggregation and rule rearrangement to make the scheme more scalable. Our investigation suggests that aggregation is an excellent idea but rule rearrangement has several problems.

In this paper we presented a novel approach for scalable packet classification by combining bit vector aggregation and folding. Folding helps to not only reduce false matches, but also locate real matches. Our algorithm avoids the limitations of rule rearrangement and effectively reduces false matches while preserving the benefits of bit vector aggregation.

In our simulations of relative performance we compare the BV algorithm, the ABV algorithm and our new AFBV algorithm, and evaluate both performance and space utilization. Preliminary results show that AFBV outperforms ABF on average, and can achieve the similar level of performance in worst cases, with about a 10% increase in memory utilization, for the additional, folded vectors.

REFERENCES

[1] F. Baboescu, G. Varghese. Scalable Packet Classification. In Proceedings of ACM SIGCOMM '01, Aug.2001, San Diego, California, USA.

[2] T. Lakshman and D.Stidialis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In Proc.ACM Sigcomm'98, Sep.1998.

[3] P. Gupta, N. McKeown. Packet classification on multiple fields. In Proceedings of ACM SIGCOMM '99 (Cambridge, Massachusetts, USA, Sept. 1999).

[4] M. Waldvogel etc. High-Performance Packet Classification for Filtering and Forwarding. IBM Zurich Research Laboratory, 2001.

[5] Lili Qiu, George Varghese, Subhash Suri. Fast Firewall Implementations for Software and Hardware-based Routers. 2001.

[6] P. Warkhede, S. Suri, G. Varghese. Fast Packet Classification for Two-imensional Conflict-Free Filters. IEEE INFOCOMM 2001.01

[7] V. Srinivasan, S. Suri and G. Varghese. Packet Classification using Tuple Space Search. Proc. of Sigcomm, 1999.

[8] M. Waldvogel. Multi-dimensional prefix matching using line search. In Proceedings of IEEE Local Computer Networks, Nov. 2000.

[9] F. Baboescu, G. Varghese. Aggregated Bit Vector Search Algorithms for Packet Filter Loo

[10] Kups. Technical Report cs2001-0673, June 2001.

[11] V. Srinivasan, et al. Fast scalable level four switching. In Proc. ACM Sigcomm'98 Sept. 1998.