Fast Packet Classification Using Bit Compression with Fast Boolean Expansion*

CHIEN CHEN, CHIA-JEN HSU AND CHI-CHIA HUANG
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan

To support applications such as Internet security, virtual private networks, and Quality of Service (QoS), Internet routers need to quickly classify incoming packets into flows. Packet classification uses information contained in the packet header and a predefined rule table in the routers. In general, packet classification on multiple fields is a difficult problem. Hence, researchers have proposed a variety of classification algorithms. This paper presents a novel packet classification algorithm, the bit compression algorithm. As with the best-known classification algorithm, bitmap intersection, bit compression is based on the multiple dimensional range lookup approach. Since bit vectors of the bitmap intersection contain many "0" bits, the bit vectors could be compressed. We compress the bit vectors by preserving only useful information and removing the redundant bits of the bit vectors. An additional index table would be created to keep track of the rule number associated with the remaining bits. Additionally, the wildcard rules enable an extensive improvement in the storage requirement. A novel Fast Boolean Expansion enables our scheme to obtain better classification speed even under a large number of wildcard rules. Compared to the bitmap intersection algorithm, the bit compression algorithm reduces the storage complexity in the average case from $O(dN^2)$ (for bitmap intersection) to $\theta(dN \cdot \log N)$, where d denotes the number of dimensions and N represents the number of rules. The proposed scheme cuts the cost of packet classification engine and increases classification performance by accessing less memory, which is the performance bottleneck in the packet classification engine implementation using a network processor.

Keywords: router, packet classification, bitmap intersection, bit compression, Boolean expansion, network processor

1. INTRODUCTION

The accelerated growth of Internet applications has increased the importance of the development of new network services such as security, virtual private network (VPN), Quality of Service (QoS), and accounting. All of these mechanisms generally require routers to be able to categorize packets into different classes called flows. The categorization function is termed packet classification.

An Internet router classifies incoming packets into flows using information contained in the packet header and a predefined rule table in the router. A rule table maintains a set of rules specified based on the packet header fields such as the network source address, network destination address, source port, destination port, and protocol type.

Received February 2, 2007; accepted July 13, 2007.
Communicated by K. Robert Lai, Yu-Chee Tseng and Shu-Yuan Chen.
* This work was supported by the New Generation Broadband Wireless Communication Technologies and Applications Project of Institute for Information Industry and sponsored by MOEA, R.O.C.

The rule field can be a prefix (*e.g.*, a network source/destination address), a range (*e.g.*, a source/destination port), or an exact number (*e.g.*, a protocol type).

When a packet arrives, the packet header is extracted and compared with the corresponding fields of the rule in the rule table. A rule matching in all corresponding fields is considered a matched rule. The packet header is compared with every rule in the rule table, and the matched rule with the highest priority yields the best-matching rule. Finally, the router performs an appropriate action associated with the best-matching rule.

The d-dimensional packet classification problem (PC problem) is formally defined as follows. The rule table has a set of rules $R = \{R_1, R_2, \ldots, R_n\}$ over d dimensions. Each rule comprises d fields $R_i = \{F_{1,i}, F_{2,i}, \ldots, F_{d,i}\}$, where $F_{j,i}$ denotes the value of field j in rule i. Each rule also has a cost (priority). A packet P with header field $(p_1, p_2, \ldots, p_d)$ matches rule R_i if all the header fields p_j, j from 1 to d, of the packet match the corresponding fields $F_{j,i}$ in R_i. If P matches multiple rules, the minimal cost (highest priority) rule is returned.

The general packet classification problem can be viewed as a point location problem in multidimensional space [1]. Rules have a natural geometric interpretation in d dimensions. Each rule R_i can be considered a "hyper-rectangle" in d dimensions, obtained by the cross product of $F_{j,i}$ along each field. The set of rules R thus can be considered a set of hyper-rectangles, and a packet header represents a point in d dimensions.

A good packet classification algorithm must quickly classify packets with minimal memory storage requirements. This study proposes a novel bit compression packet classification algorithm. This algorithm succeeds in reducing the memory storage requirements in the bitmap intersection algorithm [8], proposed by Lakshman and Stiliadis. The bitmap intersection algorithm converts the packet classification problem into a multidimensional range lookup problem and constructs bit vectors for each dimension. Since the bit vectors contain many "0" bits, the bit vectors could be compressed. We compress the bit vectors by preserving only useful information and removing the redundant bits of the bit vectors. An additional index table is created to track the rule number associated with the remaining bits. Additionally, the wildcard rules enable more extensive improvement. The bit compression algorithm reduces the storage complexity on average from $O(dN^2)$, for bitmap intersection, to $\theta(dN \cdot \log N)$, where d denotes the number of dimensions and N represents the number of rules, without sacrificing the classification performance. Although the authors of bitmap intersection proposed a scheme, called incremental read, which can reduce the storage complexity from $O(dN^2)$ to $\theta(dN \cdot \log N)$, it requires more memory accesses than its original scheme. The incremental read takes an advantage of the fact that any two adjacent bit vectors differ by only one bit. Therefore, instead of storing all the bit vectors for each interval, it stores the position of the single bit between these two bit vectors. However, when a complete bit vector of an interval needs to be reconstructed, the incremental read will access not only multiple bit positions but also a complete bit vector as a final reference. Another famous scheme is the aggregated bit vector (ABV) algorithm [9]. Even though the ABV algorithm has much fewer memory accesses, close to that of the bit compression algorithm, it demands larger memory storage than the bit compression algorithm. The ABV algorithm attempts to reduce the number of memory accesses by adding smaller bit vectors called ABVs, which partially capture information from the complete bit vectors. An ABV is created along with an original bitmap vector to speed up packet classification performance by accessing only corresponding chunks of bits in the regular bit vector identified by the ABV.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the bit compression algorithm. Section 4 summarizes the performance results. Conclusions are made in section 5.

2. RELATED WORKS

The simplest packet classification algorithm involves a linear search of each rule. Linear search is efficient in terms of memory, but requires a large search time for a large number of rules. The data structure is simple and easy to update in response to rule changes. A tries-based data structure called 'Grid of tries' [2] was proposed by Srinivansan *et al.* This scheme is a good solution if the rules are restricted to just two fields, but it could not be applied to more fields to solve the general problem. The investigation proposes another general solution called 'Cross-producting'. This approach creates a table of all possible field value combinations (cross-products) and calculates the best-matching rule matching for each cross-product in advance. Unfortunately, the size of the cross-product table grows significantly with the number of rules and fields. To reduce memory consumption on the extra rules needed to represent the cross-products, recently Dharmapurikar *et al.* [18] proposed an architecture solution to apply the "Cross-producting Algorithm" to multiple subsets of rules. They introduce an overlay-free grouping to divide the rules into multiple subsets. Then, a cross-product table without extra rules can be constructed for each subset. However, to avoid performing multiple lookups in multiple tables, the Bloom filter is introduced to sustain high throughput, as in the original cross- producting algorithm. The 'tuple-space search' [3] is another general approach. This scheme partitions the rules into different tuple categories based on the number of specified bits in each dimension, and then uses hashing among the rules within the same tuple. This scheme has a fast average search time and update time, but suffers a disadvantage by using hashing, which leads to lookups and updates with non-deterministic durations. The Recursive Flow Classification (RFC) [4] proposed by Gupta *et al.* is one of the earliest heuristic approaches. This approach attempts to map an S-bit packet header into a T-bit identifier, where $T = \log N$ (N is the number of rules) and $T << S$. This approach employs cross-producting in stages. It groups intermediate results into equivalent classes to reduce storage requirements. RFC works fast, but this speed comes at the price of substantial memory use, and it does not support efficient updating. Gupta *et al.* also propose another heuristic algorithm called 'Hierarchical Intelligent Cuttings' (HiCuts) [5]. This algorithm attempts to partition the search space in each dimension, and then it establishes a decision-tree data structure by careful preprocessing of the rule table. Each leaf node stores a small number of rules. Meanwhile, a linear search of these rules yields the desired matching. This scheme exploits the characteristics of real rule table, but these characteristics vary. The method used to find a suitable decision tree prevents effective scaling in large rule tables. The 'Extended Grid of Tries' (EGT) [6] and 'Hyper-Cuts' [7] are general packet classification algorithms that achieve high performance without extreme storage space requirements. EGT employs a slightly modified two-dimensional Grid-of-Tries for classification of source and destination addresses, followed by a linear search of the rules that match the two fields (source and destination addresses) at that point. HyperCuts, similar to HiCuts, uses multidimensional cuts at each step. Unlike HiCuts, in which each node in the decision tree represents a hyperplane, each node in the HyperCuts decision tree represents a k-dimensional ($k > 1$) hypercube.

Another interesting solution devised by Lakshman *et al.* [8] is the bitmap intersection scheme, which uses the concept of divide-and-conquer, dividing the packet classification problem into k sub-problems and then combining the results. This scheme uses the geometrical space decomposition approach to project every rule on each dimension. For N rules, a maximum of $2N + 1$ non-overlapping *intervals* are created on each dimension. Each interval is associated with an N-bits *bit vector*. Bit j in the bit vector is set if the projection of the rule range corresponding to rule j overlaps with the interval. On packet arrival, for each dimension, the interval to which the packet belongs is found. Taking conjunction of the corresponding bit vectors in each dimension to a resultant bit vector, the highest priority entry in the resultant bit vector can be determined. Since the rules in the rule table are assumed to be sorted in terms of decreasing priority, the first set bit found in the resultant bit vector is the highest priority entry. The rule corresponding to the first set bit is the best matching rule applied to the arriving packet. This scheme employs bit-level parallelism to match multiple fields concurrently, and can be implemented in hardware for fast classification. However, this scheme is difficult to apply to large rule tables, since the memory storage scales quadratically each time the number of rules doubles. The same study describes a variation that reduces the space requirement at the expense of higher execution time. Fig. 1 shows the bitmap for a 2-dimensional rule table in dimension X. The ten rules are represented by 2-dimensional rectangles. The bitmap intersection starts with projecting the edges of the rectangles to the X axis and the ten rectangles then create nine intervals. Subsequently, associate a bit vector with each interval. For example, the bit vector in interval X_2 is “1011000000” since the first, third, and fourth rules overlap X_2.

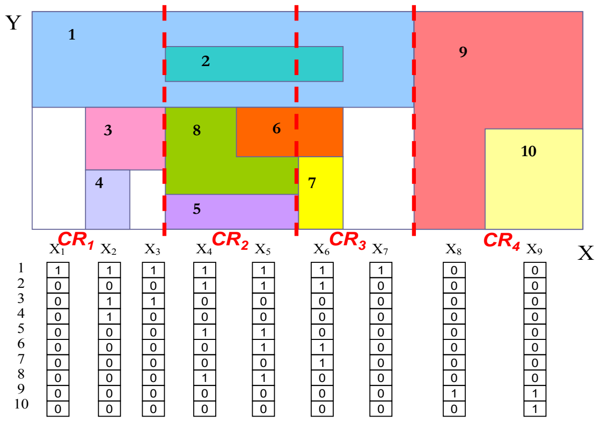


Fig. 1. The bitmap in dimension X of a 2-dimensional rule table with 10 rules.

The Aggregated Bit Vector (ABV) algorithm [9] designed by Baboescu *et al.* is an improvement on the bit map intersection scheme. The authors made two observations: there are sparse set bits in bit vectors and a packet matches few rules in the rule table.

Building on these two observations, two key ideas are extended – aggregation of bit vectors and rule rearrangement. Aggregation attempts to reduce memory access time by adding smaller bit vectors called Aggregate Bit Vectors (ABVs), which partially capture information from whole bit vectors. The length of an ABV is defined as $\lceil N/A \rceil$, where A denotes aggregate size. Bit k is set in ABV if there is at least one bit set in group k, which is from $((k-1) * A + 1)^{th}$ bit to $(k * A)^{th}$ bit in the original bit vector; otherwise, bit k is cleared. Aggregation reduces the search time for bitmap intersection, but produces another unfavorable effect, false matching, a situation where the result of the conjunctions of all ABV returns a set bit, but no actual match exists in the group of rules identified by the aggregate. False matching may increase memory access time. Rule rearrangement can alleviate the probability of false matching. Although ABV outperforms bitmap intersection for memory access time by an order of magnitude, it does not ameliorate the main problem of bit map intersection, which is the rapid exploitation of memory space. In fact, it uses more space. A detailed survey of packet classification schemes can be found in [15-17].

In a nutshell, for the general classification problem of large numbers of rules, we find that existing solutions do not balance well between performance and memory space. Our paper uses the bitmap intersection scheme as a foundation since it already is scalable in search performance. Our bit compression scheme adds a new idea by removing a large number of bits in bit vectors. The results show that our scheme not only resolves the issue of large memory requirement in bitmap intersection, but also cuts the search speed of bitmap intersection by several times when implemented on Intel IXP 1200/2400 network processors.

3. BIT COMPRESSION ALGORITHM

3.1 Motivation

As mentioned in the previous section, bitmap intersection is a hardware-oriented scheme with rapid classification speed, but suffers from a crucial drawback in that the storage requirements increase exponentially with the number of rules. The space complexity of bitmap intersection is $O(dN^2)$, where d denotes the number of dimensions and N represents the number of rules. Even though the ABV algorithm improves the search speed, it requires even more memory space than the bitmap intersection algorithm. For a packet classification hardware solution, memory storage is an important performance metric. Decreasing the required storage will reduce costs correspondingly. The question thus arises whether any method exists to resolve the extreme memory storage requirement for a large rule table. Observing the bit vectors produced by each dimension, as mentioned in [9], the set bits ("1" bits) are very sparse in the bit vectors of each dimension, and there are considerable clear bits ("0" bits). The authors of [9] used this property to reduce memory access time, but this property can also be applied to reduce memory storage requirements. As Fig. 2 shows, a space saving of approximately 60% can be achieved by removing redundant "0" bits. The shaded parts of Fig. 2 illustrate the removable "0" bits. Our challenge is how to represent the compressed format of bit vector. We segment each dimension into several sub-ranges. We call a sub-range a "Compressed Region" (CR), where a CR denotes the range of a series of consecutive intervals. In each

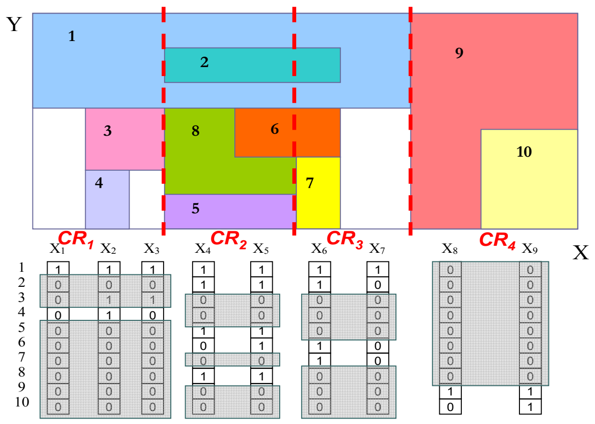


Fig. 2. pace saving by removing redundant ‘0’ bits.

CR, only an extremely small number of rules are overlapped, while the corresponding bits of the non-overlapped rules in this CR are all “0” bits. If a packet falls into a CR, denoted by CR_m, only the overlapped rules need to be considered. The corresponding bits of the non- overlapped rules in CR_m are all “0” bits. Neglecting the non-overlapped rules means those “0” bits corresponding to the non-overlapped rules of the bit vectors in CR_m can be removed. This study calls a bit vector after redundant “0” bits are removed a Compressed Bit Vector (CBV).

For example, consider the two-dimensional rule table in Fig. 1. By dividing dimension X into four CRs, CR_1, CR_2, CR_3, and CR_4. Only R_1, R_3, and R_4 are overlapped with CR_1. Therefore, if a packet falls into CR_1, only R_1, R_3, and R_4 have to be considered. Consequently, maintaining the first, third, and fourth bits of the bit vectors while removing the “0” bits of the non-overlapped rules in CR_1 is sufficient to looking for matching rules.

However, recall that in the bit map intersection, the bit order of a bit vector indicates to the rule order (i^{th} bits in a bit vector correspond to i^{th} rule in rule table). “0” bits are removed from a bit vector in such a way that it is no longer known which remaining bits represent what rules. To solve this problem, this study claims an “index list” with each CR, which stores the rule number associated with the remaining bits. Collections of the “index list” form an “index table”. For example in Fig. 1, after removing the redundant “0” bits, the bit vectors in CR_1 remain three bits. To keep track of the rule number of the three remaining bits, an index list [1, 3, 4] is appended to CR_1. Each CR associates an index list. The index table comprising four index lists is shown in Fig. 4.

After removing redundant “0” bits, we build the CBVs and index list for each CR. Because the length (number of bits) of the CBV is related to the number of overlapped rules in the corresponding CR, the length of the CBVs is different for each CR. As the example in Fig. 4 shows, the length of CBVs in CR_1 should be three bits, while the length of CBVs in CR_2 should be five bits. However, the bitmap intersection is a hardware-oriented algorithm. Our improvement scheme is also hardware-oriented. For convenience of memory access, the length of bit vectors should be fixed, and the CBVs maintaining fixed length are also desired. Therefore, the length of all CBVs should be based on the

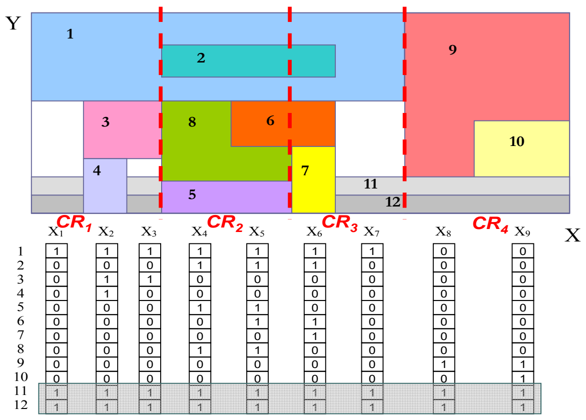


Fig. 3. The bitmap in dimension X of a 2-dimensional rule table that has two wildcard rules R_{11} and R_{12} in dimension X.

longest (maximum bits) CBV, and fill up “0” bits to the end of the CBVs which are shorter than the longest CBV. Note that a similar idea is applied to the index lists, where the index lists should have the same number of entries.

Furthermore, rules are considered to have wildcards. This study notes that if rule R_i is a wildcard in dimension j, the i^{th} bit of each bit vector in dimension j is set, thus forming a series of “1” bits over dimension j. As an example, Fig. 3 illustrates the rule table with two wildcard rules in dimension X, rule R_{11} and R_{12}. The last two bits of each bit vector are set because the ranges of R_{11} and R_{12} cover all intervals in dimension X. In [4], the authors mentioned that in the destination and source address fields, approximately half the rules are wildcards. Consequently, half of each bit vector in the destination field (or source field) is set to “1” because of the wildcards. Intrinsically, lots of these “1” bits are redundant. The idea is that for each dimension j, regardless of the interval in which a packet falls, the packet always matches the rules with wildcards in dimension j. Thus, there is no need to set corresponding “1” bits in every interval for recording these wildcard rules. Instead these rules are stored just once. Additional bit vectors, here called “Don’t Care Vectors” (DCV), are used to separate the wildcard and non-wildcard rules. A DCV is established for each dimension. Removing the redundant “1” bits caused by wildcard rules helps further reduce storage space. A DCV resembles a bit vector. Note that in a bit vector, bit j in the bit vector is set if the projection of the rule region corresponding to rule j overlaps with the related interval. In the DCV, bit j is set if the corresponding rule j is a wildcard; otherwise, bit j is clear. For example, the last two bits of each bit vector in Fig. 3 could be removed and DCV “000000000011” added instead, which indicates that the 11th and 12th rules are wildcards and others are not.

3.2 Bit Compression Algorithm

Using the above ideas, this study proposes an improved approach to bitmap intersection, called “bit compression.” Before describing the proposed bit compression scheme, this study presents some denotations and definitions.

For a k-dimensional rule table with N rules, $I_{i,j}$ denotes the i^{th} non-overlapping interval on dimension j and $ORN_{i,j}$ denotes the overlapped rule numbers for each interval $I_{i,j}$. Furthermore, $BV_{i,j}$ denotes the bit vectors associated with the interval $I_{i,j}$ and $CBV_{i,j}$ represents the corresponding compressed bit vector. DCV_j is the "Don't Care Vector" for dimension j and $DCV_{i,j}$ is the i^{th} bit in DCV_j.

Definition 1 For a k-dimensional rule table with N rules, "maximum overlap" for dimension j, denoted as MOP_j, is defined as the maximum $ORN_{i,j}$ for all intervals in dimension j.

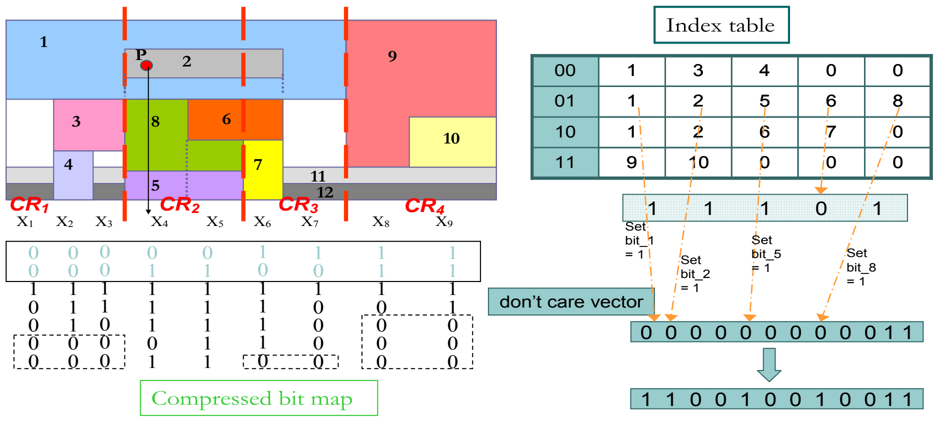
The preprocessing part of the bit compression algorithm is as follows. For each dimension j, $1 \leq j \leq k$,

1. Construct DCV_j. For n from 1 to N, if R_N is wildcarded on dimension j then $DCV_{n,j}$ is set; otherwise, $DCV_{n,j}$ is clear.
2. Calculate the value of MOP_j and segment the entire range of dimension j into t CRs, CR_1, CR_2, …, CR_t . The rules, where the rule projection overlaps with CR_i, $1 \leq i \leq t$, form a rule set RS_i, where the entry number of each rule set should be smaller than or equal to MOP_j (according to the subsequently described "region segmentation" algorithm).
3. For each CR CR_i, $1 \leq i \leq t$, construct a compressed bit vector and corresponding index list based on RS_i. Then gather the index lists to compose an index table. Furthermore, use $list_i$ to denote the index list related to CR_i and $list_{x,i}$ to represent the x^{th} entry of $list_i$.
4. For each CR CR_i, $1 \leq i \leq t$, append "index table lookup address" (ITLA), which is the binary of $(i - 1)$, in front of each CBV. For convenient hardware processing, the number of bits of the ITLA in each CR are the same.

The classification steps of a packet are as follows. For each dimension j, $1 \leq j \leq k$,

1. Find the interval $I_{i,j}$ to which the packet belongs and obtain the corresponding compressed bit vector $CBV_{i,j}$ and ITLA.
2. Use ITLA to look up the index table to obtain the corresponding index list, assume $list_m$.
3. Read the DCV_j into the final bit vector. Subsequently, read the index list found in step 2 entry by entry. If the x^{th} bit in $\text{CBV}_{i,j}$ is "1", then access $list_{x,m}$ and set the corresponding bit in the final bit vector.
4. Take the conjunction of the final bit vector associated with each dimension and determine the highest priority rule.

Fig. 4 illustrates the bit compression algorithm. First, construct the DCV "00000000 0011" for dimension X. As shown in Fig. 1, dimension X is divided into four CRs. In $CR_{1,}$ t, the corresponding rule set RS_1 is $\{R_1, R_3, R_4\}$, and thus the CBV in CR_1 is constructed by considering R_1, R_3, R_4 only and the index list $list_1$ is [1, 3, 4]. An ITLA "00" then is appended in front of the CBVs in CR_1. As mentioned previously, additional "0" bits are filled up in the CBVs and index table for the convenience of hardware implementation. Furthermore, similar steps are manipulated for CR_2, CR_3, and CR_4.



00	1	3	4	0	0
01	1	2	5	6	8
10	1	2	6	7	0
11	9	10	0	0	0

Fig. 4. An example of the bit compression algorithm.

Consider an arriving packet p shown in Fig. 4, which falls into interval X_4. The ITLA “01” and CBV “11101” associated with X_4 are accessed. The ITLA “01” serves as the lookup address in the index table to access index list [1, 2, 5, 6, 8]. The bits of “11101” then are known to represent R_1, R_2, R_5, R_6, and R_8, respectively. Read DCV “000000000011” and set the first, second, fifth, and eighth bits to form the final bit vector. The final bit vector in dimension X is “110010010011,” the same as the bit vector of interval X_4 produced by the bitmap intersection scheme in Fig. 3. Similar processes are operated to form the final bit vector of dimension Y. Take the conjunction of the final bit vectors in dimension X and Y, and the matched highest priority rule is obtained.

3.3 Maximum Overlap Statistics

Because the bit compression scheme does not change the number of bit vectors in bit intersection, the storage space saving is influenced by the length of the CBV. A shorter CBV length leads to greater space saving. However, as mentioned previously, the length of the CBV is limited by the value of maximum overlap. Consequently, the space saving increases with decreasing value of the maximum overlap, The bit compression scheme requires extra storage for the index table. If the value of the maximum overlap is sufficiently large, the overhead for the index table may exceed the profit from compression.

To determine if the number of redundant “0” bits removed from the bitmap is large enough to exceed the extra storage overhead (index table) in the bit compression algorithm, this study performs experiments on the destination field to measure statistics of the maximum overlap. This study employs two approaches to create the one-dimensional rule table with 0.5K, 1K, 5K, and 10K numbers of rules. In the first approach, the rule tables are generated by randomly picking prefixes from a Mae-West routing database [11]. In the second approach, the type of rule table is determined by the prefix length distribution probability based on five publicly available routing tables in [9] and probability, β, [10], which denotes the probability that prefix P_A is a prefix of P_B, where P_A and P_B are randomly selected from the rule table. β is an important parameter for the second rule type for controlling rule overlapping probability, and overlapping probability increases with increasing β. In [10], the authors calculated the value of β and acquired results about 10^{-5} for several real-life routing tables obtained from the Mae-West routing database. This study considers four different values of β (10^{-2}, 10^{-3}, 10^{-4}, and 10^{-5}).

Table 1 presents the statistical results of the first and second approaches, respectively. The maximum and average values of maximum overlap of 1,000 statistics are calculated. For real-life Mae-West routing tables, the ratio of maximum overlap to rule numbers shown in Table 1 is low (below 0.006). For rule tables created by the second approach, as expected, maximum overlap increases with β. Table 1 also shows that even with large β (= 10^{-2}), the ratio of maximum overlap to rule numbers remains low (below 0.044). The statistics confirm that considerable storage in the bitmap intersection is saved through bit compression. Note that the maximum overlap for the Mae-West routing table lies between the statistical results of the second approach with $\beta = 10^{-4}$ and $\beta = 10^{-5}$.

Table 1. Maximum and average values of "maximum overlap" for different β and real-life Mae-West routing table.

Number of Rules	$\beta = 10^{-2}$		$\beta = 10^{-3}$		$\beta = 10^{-4}$		$\beta = 10^{-5}$		Real-life Mae-West routing table	
	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.
0.5 K	22	15.308	7	4.827	4	2.544	3	2.014	3	2.111
1 K	36	25.879	10	6.777	6	3.205	4	2.119	4	2.561
5 K	114	98.39	26	18.183	9	5.844	5	3.158	7	4.734
10 K	253	189.08	43	29.487	11	8.178	6	3.742	9	6.58

3.4 Region Segmentation

This section describes the "region segmentation" algorithm. As mentioned previously, the region segmentation algorithm is used to segment the range of each dimension into CRs and then to group rules overlapping with each CR. Moreover, as in the bit compression algorithm, each CR is associated with an index list that contains the same number of entries (the maximum overlap). So, if more CRs are constructed, more storage space is needed to store the index list. Minimizing the number of CRs can help the bit compression algorithm save further space. Consequently, the objective of the region segmentation algorithm is to segment the range of dimension into the minimum number of CRs such that the number of rules overlapping within each CR is smaller than or equal to the maximum overlap.

The region segmentation algorithm transforms this problem into a graph model according to rule dependency. The definition of rule dependency is as follows.

Definition 2 Projecting all rule regions to one dimension, rules R_i and R_j are considered dependent if they overlap; otherwise, they are considered independent.

Accordingly, the region segmentation algorithm constructs an undirected graph $G(V, E)$ first, where $V = \{v_1, v_2, \ldots, v_N\}$, each vertex v_i corresponds to a rule R_i, and an edge is constructed between v_i and v_j if rules i and j are dependent. Graph G would include several connected components, where each connected component represents a group of mutually dependent rules that form a separated CR. The corresponding rules of a connected component form a rule set. If the vertex numbers of a connected component (rule numbers of rule set) are less than or equal to the maximum overlap, the connected component (rule set) is desired. Otherwise, the maximum degree vertex (which means neglect the rule overlapped with the maximum number of rules) and the related edges are removed, and the search for the desired connected components continues. In fact, the rule set corresponding to a connected component should consider the removed vertices originally connected with the connected component. The process is repeated until the vertex numbers of each connected component are less than or equal to the maximum overlap.

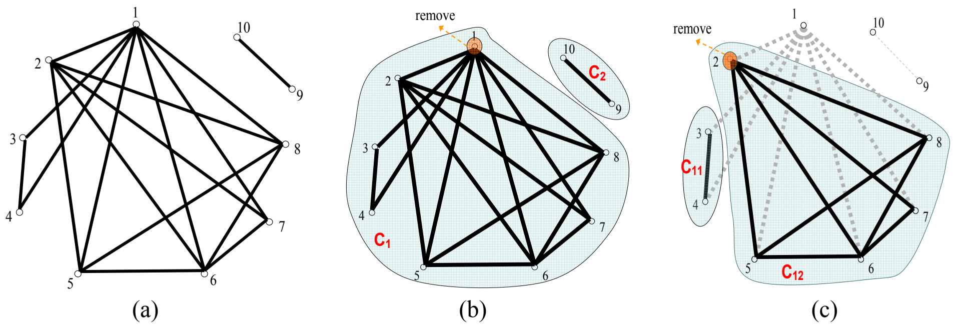
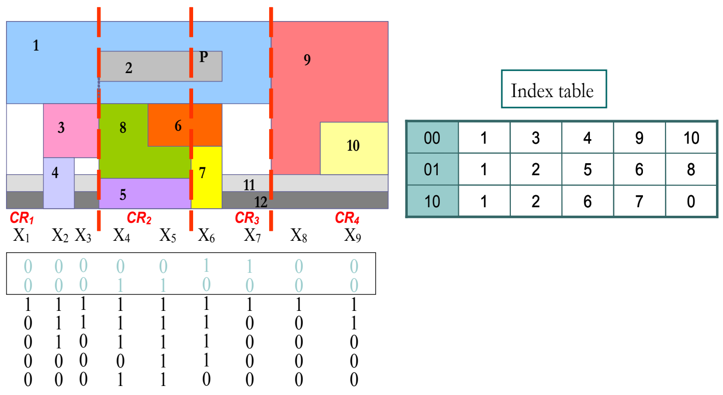


Fig. 5. (a) Graph model of the rule table in Fig. 1, and (b) and (c) illustration of the steps of "region segmentation" algorithm.

For example, Fig. 5 (a) presents a graph model constructed using the rule table of Fig. 1, in which the maximum overlap equals 5. First, two connected components, C_1 and C_2, are found, as shown in Fig. 5 (b). The corresponding rule set of C_2 is $\{R_9, R_{10}\}$, where the number of rules is 2, less than the maximum overlap, and thus the rule set is desired. On the other hand, the rule set corresponding to connected component C_1 has eight vertices, so the maximum degree vertex of C_1, vertex v_1, and the related edges are removed. Subsequently, two new connected components, C_{11} and C_{12}, are obtained, as shown in Fig. 5 (c). The corresponding rule set of C_{11} and C_{12} should take R_1 into consideration. Therefore, the rule set of C_{11} should be $\{R_1, R_3, R_4\}$ and the rule set of C_{12} should be $\{R_1, R_2, R_5, R_6, R_7, R_8\}$. $\{R_1, R_3, R_4\}$ is a desired rule set, while the maximum degree vertex, v_2, is neglected for C_{12} and the process continues. Finally, four desired rule sets $\{R_1, R_3, R_4\}$, $\{R_1, R_2, R_5, R_6, R_8\}$, $\{R_1, R_2, R_6, R_7\}$, and $\{R_9, R_{10}\}$ are obtained.

The region segmentation algorithm achieves the objective of minimizing the number of CRs by merging the rule sets. Two rule sets can be merged together if the rule numbers of the merged rule sets are smaller than or equal to the maximum overlap. The merged CRs then share the same index list. After merging the rule sets, the required number of index lists can be reduced so as to save the space of the index table. For example, Fig. 6 merges the rule sets $\{R_1, R_3, R_4\}$ and $\{R_9, R_{10}\}$, so CR_1 and CR_4 can be considered as the same CR, which uses only an index list [1, 3, 4, 9, 10]. The CBVs in regions 1 and 4 then employ ITLA "00" to access the same index list. Compared with Fig. 4, merging rule sets helps the index table in Fig. 6 save 25% in storage space.



00	1	3	4	9	10
01	1	2	5	6	8
10	1	2	6	7	0

Fig. 6. An example of the bit compression algorithm after merging rule sets.

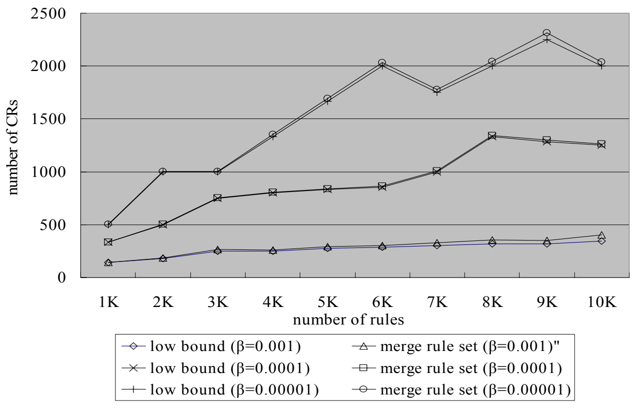


Fig. 7. Performance comparison of the number of CRs between low bound and region segmentation with merging

To observe the performance of the region segmentation algorithm with merging rule sets, we compare the number of CRs constructed by merging rule sets with the low bound. The low bound for the number of CRs is determined by the number of rules and maximum overlap, since the maximum number of rules in each CR equals the maximum overlap. Therefore, the low bound of the number of CRs for N-rules rule table is N/MOP_j. Fig. 7 presents the statistical results under three different β (10^{-3}, 10^{-4}, and 10^{-5}). The number of CRs constructed by region segmentation with merging rule sets is very close to the low bound.

3.5 Fast Boolean Expansion (FBE)

One can see that more memory access for bit compression algorithm, compared to bitmap intersection, is needed when processing an arriving packet with wildcard rules. For each dimension, our scheme needs not only to access the CBV and the index list, but also to access extra wildcard rules knowledge, DCV, whose size is identical with the bit vector of the bitmap intersection algorithm. In this section, we introduce an improving scheme modified from the original bit compression scheme described in the previous section. The intent is to minimize the amount of memory access by accessing the appropriate parts of the DCV instead of accessing the complete DCV.

In the bit compression algorithm, the DCV is used to keep track of every wildcard. If rules are not considered to have wildcards, the lookup performance of bit compression outperforms the bitmap intersection scheme as shown later in simulation. However, from [4], we know that approximately half of the rules are wildcards in the destination and source address fields. The question now arises: How to reduce the performance degradation caused by accessing DCV?

To gain some insight, the two-dimension rule table is considered. We translate the steps of the classification scheme into a Boolean expression:

$$(CBV_s + DCV_s) \cdot (CBV_d + DCV_d) \tag{1}$$

where $CBV_s + DCV_s$ and $CBV_d + DCV_d$ are from steps 1 through 3 of the bit compression algorithm in section 3.2 for source dimension and destination dimension, respectively. The notation " · " means the conjunction of the final vector associated with each dimension, *i.e.*, step 4. In Eq. (1), several interesting observations come to attention. First, full-length DCV_j is always accessed whatever the length of CBV_j. Second, we consider the relation between set bits in CBV_i and their corresponding bit in DCV_j, as well as if any matching existed between CBV_i and CBV_j, where $i \neq j$. it is futile to conjoin two complete vectors if the probability that the i^{th} bit is set in both vectors is low. This inspires us: if, early in the process, we can combine CBV_s and essential parts of DCV_s appropriately to obtain the integrated matchings, then the only thing what we should do is select the highest priority matching from those integrated matchings. Therefore, based on this idea, we expand the original Boolean expression to a new form:

$$(CBV_s \cdot CBV_d) + (CBV_s \cdot DCV_d) + (DCV_s \cdot CBV_d) + (DCV_s \cdot DCV_d) \tag{2}$$

In Eq. (2), a matching rule for a packet is obtained by comparing the priority of the four rules generated from the four clauses of Eq. (2). Processing $(CBV_s \cdot CBV_d)$ takes few memory accesses since CBV_s and CBV_d are compressed bit vectors. To reduce the number of memory accesses, while conjoining $(CBV_s \cdot DCV_d)$ and $(DCV_s \cdot CBV_d)$, we only extract the essential bits from DCV that are corresponding to the set bits of CBV instead of reading the complete DCV. Moreover, there is no need to process $(DCV_s \cdot DCV_d)$ since we know the conjunction of DCV_s and DCV_d is the default rule. Based on this modification, the bit compression algorithm can obtain much better classification speed as shown in simulation even under a large number of wildcard rules.

4. PERFORMANCE RESULTS

For a d-dimensional rule table with N rules, the query time of the proposed bit compression scheme comprises the time required for interval lookup, T_{IL}, and the time to access ITLAs, CBVs, index lists, and DCVs. The time complexity is $\theta(d \cdot (T_{IL} + (\log r + n$

$+ n \cdot \log N + N)/W)$), where r denotes the number of index lists, n represents the value of maximum overlap, W is the memory bandwidth, and the time complexity of bitmap intersection algorithm is $\Theta(d \cdot (T_{RL} + N/W))$.

The space requirement of the bit compression algorithm consists of four parts – ITLAs, CBVs, index tables, and DCVs. This study neglects the space complexity of the DCV because it has a much smaller size than the other three parts. On average, the memory space complexity is $\Theta(d \cdot N \cdot (\log r + n + \log N))$. The storage complexity is reduced from $O(dN^2)$ of the bitmap intersection algorithm to $\Theta(dN \cdot \log N)$.

In the worst case, for N rules, a maximum of $2N + 1$ non-overlapping intervals are created on each dimension. Each interval is associated with an N-bit bit vector; therefore, the storage consumption is $\Theta(N^2)$. For the bit compression algorithm, the storage consumption is calculated as: for each dimension, the ITLA has a $\log|CR|$-bit index for each interval, *i.e.*, $(2N + 1) \cdot \log|CR|$. The CBV has a MOP-bit vector for each interval. In the worst case, MOP is N, so it consumes $N \cdot (2N + 1)$ bits. The index table consumes $|CR| \cdot MOP \cdot \log N = |CR| \cdot N \cdot \log N$ bits. The DCV consumes N bits. Therefore, the total space consumption is $\Theta(d((2N + 1) \cdot \log|CR| + N \cdot (2N + 1) + |CR| \cdot N \cdot \log N + N))$. If we take $|CR|$ as a constant, this is equivalent to $O(dN^2)$. Even though the theoretical space consumption is not improved, the actual memory requirement can be reduced, as Figs. 8, 9, and 10 shown.

We implement the bitmap intersection and bit compression algorithms with Microengine C. Experiments are conducted on the Intel IXP 1200/2400 (Internet Exchange Processor) Developer WorkBench [13]. IXP 1200/2400 are the network processors, which consist of a core processor, StrongARM, and 6/8 microengines [12]. Memory hierarchies in IXP 1200/2400 consist of multiple memories. We focus on three primary storage devices (Scratchpad memory, SRAM, and SDRAM). We use six microengines to receive, classify, and route packets and use the remaining two microengines for packet transmission. The StrongARM core handles routing table management and exceptions. Three types of memory storage are used in IXP 2400: a SRAM configuration of 64MB, a SDRAM of 128MB, and a scratchpad memory of 16kB. IXP 1200 has less memory. Its three memory sizes are 8MB, 64MB, and 4KB respectively. In bitmap intersection, SRAM or SDRAM is used to store the bit vectors according to space requirements. Moreover, the relative memory access time and the bus bandwidth of the IXP 2400 memory is Scratchpad: 10 with a 32-bit bus; SRAM: 14 with a 32-bit bus; and SDRAM: 20 with a 64-bit bus [14]. In the bit compression scheme, SRAM is used to store compressed bit vectors, while the scratchpad is used to store index table and DCV.

This study considers the complexity of storage requirements and classification performance. We compare the proposed bit compression scheme with the bitmap intersection scheme. This study focuses on the two-dimensional rule table, IP destination address, and IP source address. The proposed scheme randomly generates two field rules to create a synthesized rule table, as previous experiments consider the prefix length distribution and β. Recall that for the real-life routing table, the value of β is approximately 10^{-5} and maximum overlap is measured to be between $\beta = 10^{-4}$ and 10^{-5}. Therefore, this study lists the experimental performance statistics with $\beta = 10^{-4}$ and 10^{-5} and even a larger $\beta = 10^{-3}$.

Figs. 8, 9, and 10 compare the memory requirements (based on $\log_2$) for the bitmap intersection and bit compression schemes. Note that since the bitmap intersection and bit compression algorithms use the same size of memory storage to store interval boundaries,

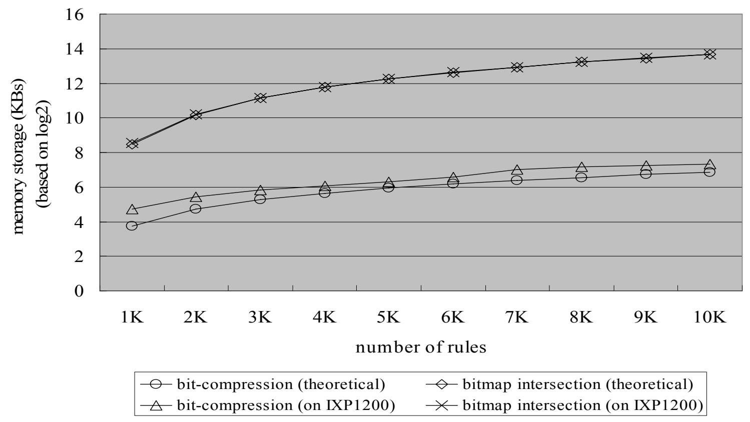


Fig. 8. Memory requirements under $\beta = 10^{-3}$.

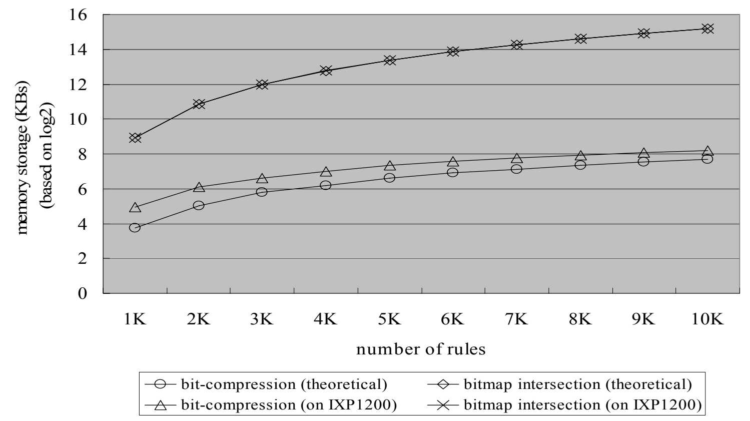


Fig. 9. Memory requirements under $\beta = 10^{-4}$.

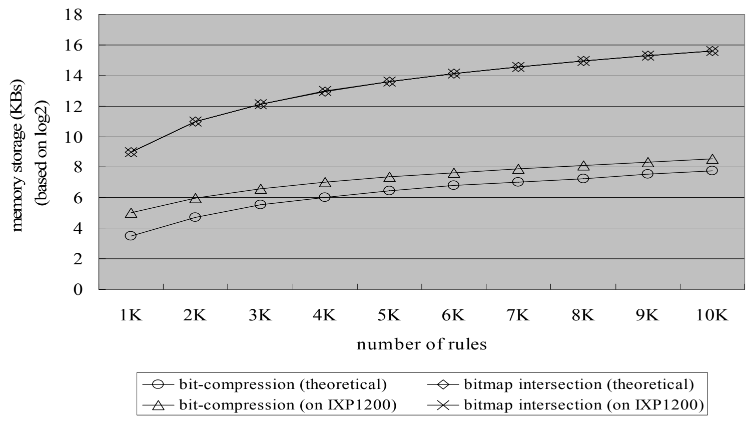


Fig. 10. Memory requirements under $\beta = 10^{-5}$.

we omit the memory storage of the interval boundary in memory requirements. The experimental results demonstrate that the proposed scheme performs better than the bitmap intersection algorithm. Under $\beta = 10^{-5}$, with the rule table size of 5k, we need only 164 KBytes for the bit compression algorithm compared to 12.5 MBytes required by the bitmap intersection algorithm. With the rule table size of 10K, 374 Kbytes are needed for the bit compression algorithm compared to 48 MBytes required by the bitmap intersection algorithm. When the rule number doubles, the memory consumption of the bit compression algorithm increases (374/164) = 2.28 times, which approximates (10K/5K) · (log10K/log5K) = 2 · $\log_{5K}$ 10K = 2 · $\log_{5K}$ 5K · 2 = 2 · ($\log_{5K}$ 5K + $\log_{5K}$ 2) = 2 · (1 + $\log_{5K}$ 2) ≅ 2. The difference is caused by storing the index table and related information. The memory consumption of the bitmap intersection algorithm increases (48/12.5) = 3.84 times, which approximates $N^2 = 4$. The simulation result shows the bit compression algorithm significantly decreases memory consumption while the rule number increases in the proportion we expect. The memory storage requirement for the bitmap intersection algorithm scales quadratically each time the number of rules doubles, while the memory storage requirement for the bit compression algorithm scales twice as the number of rules doubles. The bit compression algorithm prevents memory explosion with large rule tables. The difference between theoretical measurement and implementation on the IXP 1200 is that the lengths of the CBV, index list, and DCV are multiples of 32 bits, wasting a certain amount of space. The memory storage with implementation on the IXP1200 is higher than the theoretical storage requirements.

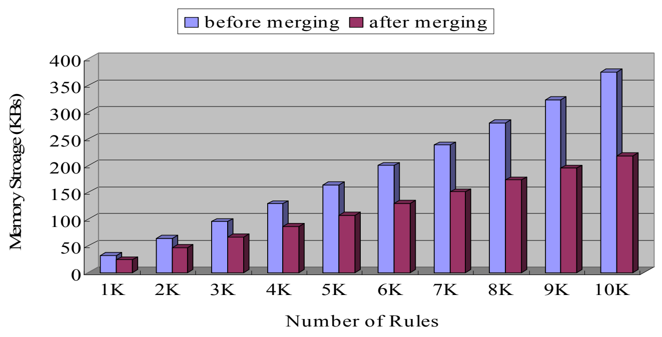


Fig. 11. Improvement of memory storage by merging rule sets under $\beta = 10^{-5}$.

As noted previously, the space of the index table can be further reduced by merging the rule sets. Fig. 11 displays the total memory space consumed by the rule table of the bit compression scheme with and without merging. The required space is reduced around 25%-40% after merging the rule sets.

In the bitmap intersection scheme, the rule table is expected to store in SRAM. But the memory storage increases rapidly such that the required storage exceeds the size of SRAM (8MB) in IXP 1200. For example, under $\beta = 10^{-5}$, the required storage space for the rule table with rules more than 4K exceeds 8MB. Thus the rule table of more than 4K rules must be stored in SDRAM. In the bit compression scheme, the memory explosion is prevented. For a two-dimensional rule table with 10K rules, the bit compression scheme stores the bit vector and index table using SRAM.

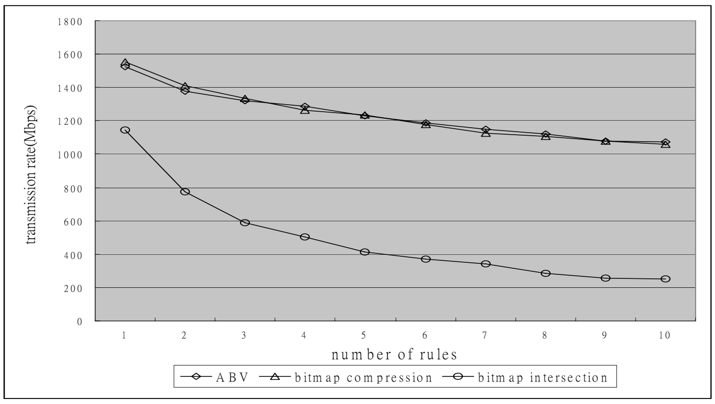


Fig. 12. Transmission rate vs. number of rules without wildcards.

As mentioned above, the bit compression algorithm needs less memory access than the bitmap intersection algorithm. Compared with bitmap intersection, the bit compression algorithm requires decompressing the CBV to full bit vector. Extra processing time for decompression is required, which will degrade the classification performance of the bit compression algorithm. However, the time for decompression is actually much less than the memory access time. The memory access time dominates the classification performance. Therefore, even the bit compression scheme requires extra processing time for decompression. The bit compression algorithm can still outperform the bitmap intersection algorithm, as Fig. 12 illustrates. Fig. 12 presents the packet transmission rates for the bitmap intersection, aggregated bit vector, and bit compression schemes with different sizes (from 1K to 10K) of rule tables without wildcards under $\beta = 10^{-5}$ on the IXP 2400. The minimum size packets (46 Bytes) were created as arriving data. The number of memory access for reading a CBV and index list is less than used for reading a full bit vector, although extra processing time for decompression is required for the bit compression scheme. The bit compression scheme outperforms the bitmap intersection scheme. Moreover, since the lengths of CBVs and index lists remain an almost fixed value (according to maximum overlap), the transmission rate of the bit compression scheme remains constant. In contrast, the transmission rate of the bitmap intersection decreases linearly with the number of rules.

Figs. 13 (a) and (b) demonstrate the performance of the bitmap intersection algorithm, aggregated bit vector algorithm, and proposed bit compression algorithm with and without Fast Boolean Expansion under different amounts of rules (from 1K to 10K) with 20% and 50% percent of wildcards. When rules are not considered to have wildcards, the results shown in Fig. 12 demonstrate that the proposed bit compression algorithm outperforms the bitmap intersection algorithm and is better than the aggregated bit vector. As previously mentioned, DCV is used to reserve the wildcard information if the rule database is considered to have wildcards. Therefore, in practice, we can even omit the DCV and the need to access it if the rule database does not contain wildcards such as in Fig. 12.

However, the result is contrary if the rule database is considered to have wildcards. Figs. 13 (a) and (b), with 20% and 50% wildcards, respectively, indicate the performance

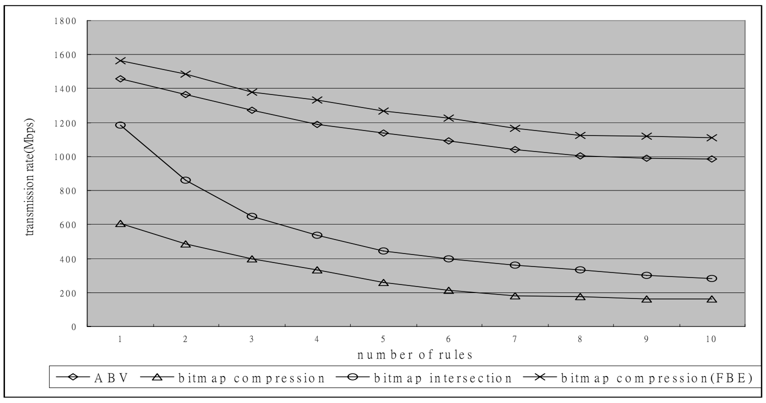


Fig. 13. (a) Rule table contains 20% wildcard.

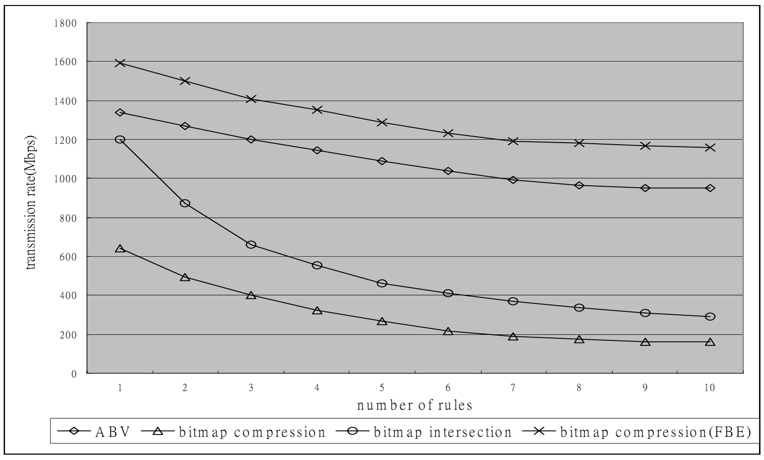


Fig. 13. (b) Rule table contains 50% wildcard.

Fig. 13. Transmission rate of bitmap intersection, bit compression, bit compression with FBE, and aggregated bit vector vs. number of rules with wildcards.

comparison between the four algorithms. As Figs. 13 (a) and (b) indicate, expectably, the bit compression algorithm has the poorest behavior compared to the bitmap intersection algorithm and aggregated bit vector algorithm. The reason for this is that if the rule tables contain wildcard rules, most of the memory access cost of the bit compression scheme is expended to access the DCVs which are the same size as the bit vectors in the original bitmap intersection algorithm. Therefore, plus the CBVs and index table, the number of memory accesses of the bit compression algorithm are more than the bitmap intersection algorithm for the same size of rule tables with wildcards. Even we take advantage of memory hierarchy to store the DCVs in the smallest (16 KB) but fastest scratchpad memory rather than the SDRAM. However, in IXP 2400 the bus width of DRAM (64 bits) is twice as the bus width of scratchpad memory (32 bits). The memory access performance of the bit compression algorithm is worst than bitmap intersection if the accesses of DCVs are required. To improve the performance, we employ the concept of Fast Boolean

Expansion, which is proposed in the previous section. The results are also presented in Figs. 13 (a) and (b).

As Figs. 13 (a) and (b) indicate, the bit compression algorithm with Fast Boolean Expansion has the best classification performance. It also outperforms the aggregated bit vector algorithm. Because in ABV, to lower the probability of false matching, a new sorting scheme has been presented to concentrate the set bits in the bit vectors produced by the rules. However, the experiments indicate that reordering the 1st dimension decreased the concentration of the bits in the 2nd dimension. Hence, ABV's performance fell in respect to the rule table with a large percentage of wildcards, because a large percentage of wildcards raises the chance of false matching. This figure proves that the proposed Fast Boolean Expansion indeed decreases the amount of memory access and increases the classification speed. For 10K rules with 50% wildcard as an example, the bit compression algorithm with FBE takes at most 38 memory accesses, since as mentioned in the previous section, there are at most nine rule overlaps in each bit vector. But the bitmap intersection algorithm takes 626 memory accesses and the aggregated bit vector algorithm only needs 40 memory accesses. Although the aggregated bit vector algorithm has the similar memory accesses as the proposed bit compression algorithm, the storage requirements for the aggregated bit vector algorithm is much greater than that for the proposed bit compression algorithm.

5. CONCLUSION

Packet classification is an essential function of Internet security, virtual private networks, QoS, and various network services. Numerous investigations have addressed the problem of efficient packet classification. This paper attempts to improve the original bitmap intersection algorithm, which has a memory explosion problem for large rule tables. This study introduces the notion of bit compression to significantly decrease the storage requirement, creating what we call the CBV. Bit compression is based on the fact that "1"bits are sparse enabling redundant "0" bits to be removed. By region segmentation, the bit compression algorithm segments the range of dimension into CRs and then associates each CR with an index list. Merging rule sets reduces the number of CRs further. For rule tables with wildcard rules, the bit compression proposes a novel idea, "Don't Care Vector" to save plenty of storage space. The experiments for measuring maximum overlap led us to believe that plenty of redundant "0" bits exist, such that removing "0" bits can significantly improve memory storage.

Compared to the bitmap intersection algorithm, the storage complexity is reduced from $O(dN^2)$ for the bitmap intersection algorithm to $\theta(dN \cdot \log N)$. In our experiment, the bit compression scheme needs less than 380 Kbytes to store the two-dimensional rule table with 10K rules, while the bitmap intersection algorithm needs 48 MBytes. When comparing memory access speed, our algorithm accesses, on average, 96% fewer bits than the bitmap intersection algorithm under the rule tables without wildcards. For the rule tables with wildcards, by exploiting Fast Boolean Expansion, the bit compression scheme requires much less memory access time than the bitmap intersection algorithm, even though extra processing time for decompression is required for bit compression. Since memory access dominates the classification performance. From the experiments, our bit compression scheme with Fast Boolean Expansion outperforms the bitmap intersection and aggregated bit vector schemes on packet classification speed.

REFERENCES

1. M. H. Overmars and A. F. van der Stappen, "Range searching and point location among fat objects," *Journal of Algorithms*, Vol. 21, 1996, pp. 629-656.
2. V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Fast and scalable layer four switching," in *Proceedings of ACM SIGCOMM*, 1998, pp. 203-214.
3. V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proceedings of ACM SIGCOMM*, 1999, pp. 135-146.
4. P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of ACM SIGCOMM*, 1999, pp. 147-160.
5. P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," *IEEE MICRO*, Vol. 20, 2000, pp. 34-41.
6. F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: is there an alternative to CAMs," in *Proceedings of IEEE INFOCOM*, Vol. 1, 2003, pp. 53-63.
7. S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of SIGCOMM*, 2003, pp. 213-224.
8. T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of ACM SIGCOMM*, 1998, pp. 191-202.
9. F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Transactions on Networking*, Vol. 13, 2005, pp. 2-14.
10. G. Zhang, H. J. Chao, and J. Joung, "Fast packet classification using field-level trie," in *Proceedings of IEEE GLOBECOM*, Vol. 6, 2003, pp. 3201-3205.
11. http://www.merit.edu/ipma/routing_table.
12. "IXP2400 network processor: datasheet," Part No. 301164-011, 2004.
13. "Intel IXP2XXX product line of network processors development tools user's guide," Part No. 278733-018, 2004.
14. D. Comer, *Network System Design using Network Processors*, Intel IXP 2xxx version, Upper Saddle River, Pearson Prentice Hall, NJ, 2005.
15. P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network Special Issue*, Vol. 15, 2001, pp. 24-32.
16. D. Taylor, "Survey & taxonomy of packet classification techniques," Technical Report, No. WUCSE-2004-24, Department of Computer Science and Engineering, Washington University in Saint Louis, U.S.A., 2004.
17. G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*, Morgan Kaufmann Publishers, San Francisco, CA, 2005.
18. S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast packet classification using bloom filters," Technical Report, No. WUCSE-2006-27, Department of Computer Science and Engineering, Washington University in Saint Louis, U.S.A., 2006.

Chien Chen (陳健) received his B.S. degree in Computer Engineering from National Chiao Tung University in 1982 and the M.S. and Ph.D. degrees in Computer Engineering from University of Southern California and Stevens Institute of Technologies in 1990 and 1996. Dr. Chen hold a Chief Architect and Director of Switch Architecture position in Terapower Inc., which is a terabit switching fabric SoC startup in San Jose, before joining National Chiao Tung University as an Assistant Professor in August 2002. Prior to joining Terapower Inc., he is a key member in Coree Network, responsible for a next-generation IP/MPLS switch architecture design. He joined Lucent Technologies, Bell Labs, NJ, in 1996 as a Member of Technical Staff, where he led the research in the area of ATM/IP switch fabric design, traffic management, and traffic engineering requirements. His current research interests include wireless ad-hoc and sensor networks, switch performance modeling, and DWDM optical networks.

Chia-Jen Hsu (許嘉仁) received his B.S. degree in Computer Science and Information Engineering from National Chung Chen University in 2001 and the M.S. degree in Computer Science from National Chiao Tung University, Taiwan in 2003. His research interests include embedded systems and packet classification.

Chi-Chia Huang (黃啟嘉) received his B.S. degree in Computer Science from National Chiao Tung University in 2004. He is a master student in Department of Computer Science and Information Engineering of National Taiwan University now. His current research interests include artificial intelligent and computer networks.