Fast Packet Classification Using Bit Compression with Fast Boolean Expansion

Chien Chen, Chia-Ren Hsu, Chi-Chia Huang and Chia-Sheng Chou
Department of Computer Science
National Chiao Tung University
HSINCHU 30050, TAIWAN, ROC
chienchen@cs.nctu.edu.tw, gis91597@cis.nctu.edu.tw, chifatty.cis91@nctu.edu.tw, gis93584@cis.nctu.edu.tw

***Abstract*—In order to support Internet security, virtual private networks, Quality of Service (QoS), etc., Internet routers need to classify incoming packets quickly 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 algorithms. This paper presents a novel packet classification algorithm, called bit compression algorithm. Like the previously best known algorithm, bitmap intersection, bit compression is based on the multiple dimensional range lookup approach. Since bit vectors of the bitmap intersection contain lots of '0' bits, the bit vectors could be compressed. We compress the bit vectors by preserving only useful information but removing the redundant bits of the bit vectors. An additional index table would be created to keep tract of the rule number associated with the remaining bits. Additionally, the wildcard rules also enable more extensive improvement in storage requirement. A novel Fast Boolean Expansion enable our scheme obtain much better classification speed even under large number of wildcard rules. Comparing with the bitmap intersection algorithm, the bit compression algorithm reduces the storage complexity in the average case from $\Theta(dN^2)$ of bitmap intersection to $\Theta(dN \cdot logN)$, where d denotes the number of dimensions and N represents the number of rules. With less memory requirement, the proposed scheme not only cut the cost of packet classification engine down, but also increases classification performance by accessing less memory which is the performance bottleneck in the packet classification engine implementation using a network processor.**

I. 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), accounting, and so on. All of these mechanisms generally require the router to be able to categorize packets into different classes called flows. The categorization function is termed packet classification.

An Internet router categorizes incoming packets into flows utilizing 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, protocol type and possibly other fields. 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 first and then compared with the corresponding fields of 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 rule table, and the matched rule with the highest priority yields the best-matching rule. Finally, the router performs an appropriate action associating with the best-matching rule.

The general packet classification problem can be viewed as a point location problem in multidimensional space [1][2]. 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 classify packets quickly 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 [3], proposed by Lakshman and Stiliadis. As shown in Fig. 1, 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 lots of '0' bits, the bit vectors could be compressed. We compress the bit vectors by preserving only useful information but removing the redundant bits of the bit vectors. An additional index table would be created to keep tract of the rule number associated with the remaining bits. Additionally, the wildcard rules also enable more extensive improvement. The bit compression algorithm reduces the storage complexity in average from $O(dN^2)$ of bitmap intersection to $\Theta(dN \cdot logN)$, 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 $\Theta(dN \cdot logN)$, however it requires more memory access than its original scheme. The incremental read takes an advantage form the fact that any two adjacent bit vectors different by only one bit. Therefore, instead of store the entire bit vectors for each interval, it store the position of single bit which different between theses two bit vectors. However, when a complete bit vector of an interval needs to be reconstructed the increment read will access

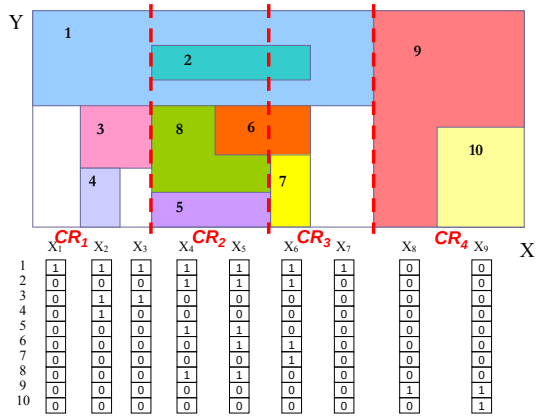


Fig. 1: The bitmap in dimension X of a 2-dimensional rule table with 10 rules.

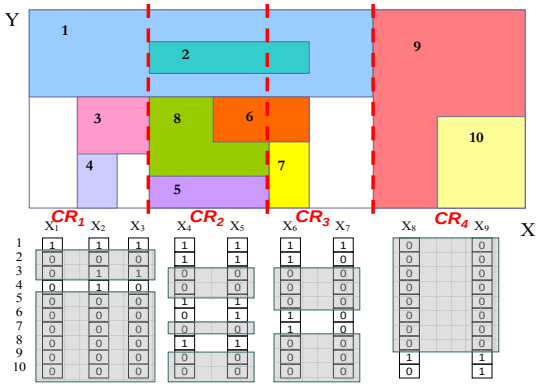


Fig. 2: Space saving by removing redundant '0' bits.

not only multiple bit positions but also a complete bit vector as a final reference. Another famous scheme is aggregated bit vector algorithm (ABV) [4]. Even though ABV has much less memory access close to bit compression algorithm, it demands larger memory storage than bit compression algorithm. The ABV attempts to reduce number of memory access by adding smaller bit vectors called ABVs, which partially captures information from the complete bit vectors. An ABV is created along with a original bitmap vector to speed up packet classification performance by accessing only corresponding chunk of bits in the regular bit vector identified by the ABV.

The rest of this paper is organized as follows. The bit compression algorithm is described in Section 2. Section 3 summarizes the performance results. Conclusions are finally made in Section 4.

II. Bit Compression Algorithm

A. Motivation

As mentioned in the previous section, bitmap intersection is a hardware oriented scheme with rapid classification speed, but suffers from the crucial drawback 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, but requires even more memory space than bitmap intersection algorithm. For a hardware solution of packet classification, memory storage is an important performance metric. Decreasing the required storage will reduce costs correspondingly. The question thus arises whether any method exists way of solving the extreme memory storage of a large rule table. Observing the bit vectors produced by each dimension, as mentioned in [4], the set bits ("1" bits) are very sparse in the bit vectors of each dimension, there are considerable clear bits ("0" bits). The authors of [4] used this property to reduce memory access time, but this property can also be applied to reduce memory storage requirements. For the example of Fig. 2, an approximately 60% space saving can be achieved by removing redundant '0' bits. The shaded parts of Fig. 2 illustrate the removable '0' bits.

Therefore, our challenge is how to represent compressed format of bit vector. We try to segment each dimension into several sub-ranges. We call the sub-range "Compressed Region" (CR), where a CR denotes the range of a series of consecutive intervals. In each CR, only an extreme 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 taken into consideration, while the non-overlapped rules do not. The corresponding bits of the non-overlapped rules in CR_m are all 0 bits. Neglecting the non-overlapped rules means these 0 bits corresponding to the non-overlapped rules of the bit vectors in CR_m can be removed. This study calls the bit vector after removing redundant 0 bits CBV (Compressed Bit Vector).

For example, consider the two-dimensional rule table like 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 forth bits of the bit vectors while removing the '0' bits of the non-overlapped rules in CR_1 are 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 corresponds 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. In order to keep track of the rule number of the three remaining bits, an index list [1, 3, 4] is appended in CR_1. Each CR associates an index list, and the index table comprising four index lists shows in Fig. 3.

After removing redundant 0 bits, we build the CBVs and index list for each CR. Because the length (number of bits) of CBV is related to the number of overlapped rules in the corresponding CR, the length of the CBVs is different for each CR, For example as Fig. 3, the length of CBVs in CR_1 should be three bits, while the length of

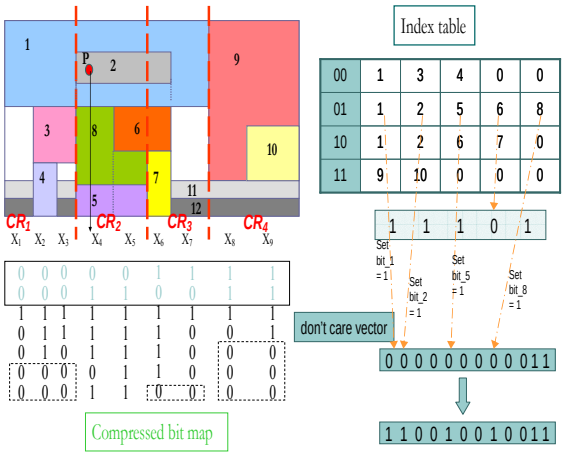


Fig. 3: An example of bit compression algorithm.

CBVs in CR_2 should be five bits. However, the bitmap intersection is a hardware-oriented algorithm, and our improvement scheme is also hardware-oriented. For convenience of memory access, the length of bit vectors should be fixed, and thus CBVs maintaining fixed length are also desired. Therefore, the length of all CBV should be based on the longest (maximum bits) CBV, and fill up '0' bits to the end of the CBVs which are shorter than the longest CBV. Notably, 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 wildcard in dimension j, the i^{th} bit of each bit vector in dimension j is set, therefore forms a series of '1' bits over dimension j. For example, Fig. 4 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 of rules are wildcard. Consequently, half of each bit vector in the destination field (or source field) is set to '1' owing to wildcards. Intrinsically, lots of these '1' bits are redundant. The idea is that for each dimension j, regardless of the interval that a packet falls in, 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, and instead these rules are stored just once. Additional bit vectors, here called "Don't Care Vectors" (DCV), are utilized 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 wildcarded, otherwise bit j is clear. For example, the last two bits of each bit vector in Fig. 4 could be removed and DCV "000000000011" added instead, which indicates that the 11th and 12th rules are wildcarded and others are not.

B. Bit Compression Algorithm

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

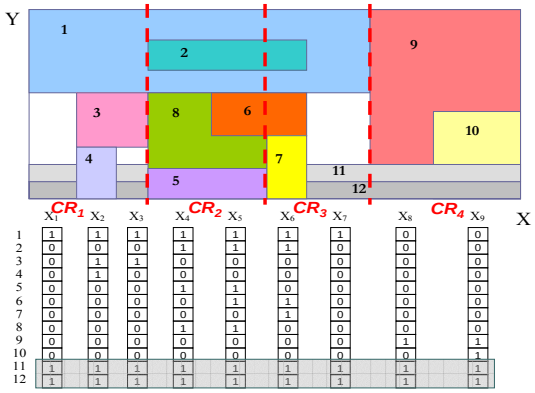


Fig. 4: The bitmap in dimension X of a 2-dimensional rule table which has two wildcarded rules R_{11} and R_{12} in dimension X. rule table with 10 rules.

For a k-dimensional rule table with N rules, let $I_{i,j}$ 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. Finally, DCV_j is the "Don't Care Vector" for dimension j and $DCV_{i,j}$ is the i^{th} bit in DCV_j.

Definition: 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 bit compression algorithm is as follows. For each dimension j, $1 \le j \le k$,

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, $1 \le i \le t$, 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, $1 \le i \le t$, 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, $1 \le i \le t$, 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 ITLA in each CR are all the same.

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

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 $CBV_{i,j}$ 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 then determine the highest priority rule implied to the packet.

Figure 3 illustrates the bit compression algorithm. First, construct the DCV "000000000011" for dimension X. As shown in Fig. 1, the dimension X is divided into 4 CRs. In CR_1, 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.

Consider an arriving packet p shown in Fig. 3, which falls into interval X_4. The ITLA "01" and CBV "11101" associated with X_4 thus 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. 1. 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 then the matched highest priority rule is obtained.

C. Fast Boolean Expansion (FBE)

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

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

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

$$(CBV_s+DCV_s)\cdot(CBV_d+DCV_d) \quad (1)$$

, where CBV_s+DCV_s and CBV_d+DCV_d means the step 1 to 3 of bit compression algorithm in section III.B for source dimension and destination dimension respectively, and the notation "$\cdot$" means the conjunction of the final vector associated with each dimension, i.e. step 4. In (1), several interesting observations come to our notice. First, full-length DCV_j is always accessed whatever the length of CBV_j will be. Secondly, what we consider is the relation between set bits in CBV_i and their corresponding bit in DCV_j, as well as whether any matching existed between CBV_i and CBV_j, where $i \neq j$. Finally, it is unworthy to conjoin two complete vectors if the probability of the i_{th} bit is set in both vectors is rare. This inspires us: if we can early combine *CBVs* and essential parts of *DCVs* appropriately to obtain the intergraded matchings, then the only thing what we should do is to select the highest priority matching from those intergraded matchings. Therefore, basing on this idea, we expand the original Boolean expression to a new form:

$$(CBV_s\cdot CBV_d) + (CBV_s\cdot DCV_d) + (DCV_s\cdot CBV_d) + (DCV_s\cdot DCV_d) \quad (2)$$

In (2), a matching rule for a packet is obtained by comparing the priority of the four rules generating from the four clauses of (2). Processing $(CBV_s\cdot CBV_d)$ takes few memory accesses since CBV_s and CBV_d are compressed bit vectors. In order to reduce the number of memory access, while conjoining $(CBV_s\cdot DCV_d)$ and $(DCV_s\cdot CBV_d)$, we only extract the essential bits from DCV which are corresponding to set bits of CBV instead of reading complete DCV. Moreover, it has no need to process $(DCV_s\cdot DCV_d)$ since we have known 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 large number of wildcard rules.

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

The space requirement of the bit compression comprises four parts – ITLAs, CBVs, index table and DCVs. This study neglects the space complexity of the DCV because of having much smaller size than the other three parts. In average case, the memory space complexity is $\Theta(d\cdot N\cdot(logr+n+logN))$. The storage complexity is reduced from $\Theta(dN^2)$ of bitmap intersection to $\Theta(dN\cdot logN)$.

In 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 bit compression the storage consumption is calculated as: for each dimension, the ITLA has a log|CR|-bit index for each intervals, i.e. $(2N+1)\cdot log|CR|$; the CBV has a MOP-bit vector for each intervals, in the worst case, MOP is N, so it consumes $N\cdot(2N+1)$ bits; index table consumes $|CR|\cdot MOP\cdot logN=|CR|\cdot N\cdot logN$ bits; DCV consumes N bits. Therefore, the total space consumption is $\Theta(d((2N+1)\cdot log|CR|+N\cdot(2N+1)+|CR|\cdot N\cdot logN+N))$, if we take |CR| as a constant, this is equivalent to $\Theta(dN^2)$. Even though the theoretical space consumption is not improved, the actual memory requirement can be reduced, as is seen in Fig. 5.

This study considers the complexity of storage requirement 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 [5], where is a controlling rule overlapping probability. The overlapping probability increases with increasing of β. This study considers $\beta=10^{-5}$[8].

.

We implement the bitmap intersection and bit compression algorithm with Microengine C. Experiments are conducted on the Intel IXP2XXX (Internet Exchange Processor) Developer WorkBench [7]. IXP 2400 is a network processor, which consists of a core processor, StrongARM, and eight microengines [6]. Memory hierarchy in IXP2400 consists of multiple memories, and three primary storage devices (Scratchpad memory, SRAM and SDRAM) are focused on.

Figures 5 compare the memory requirements (based on log_2) for the bitmap intersection and bit compression schemes. Notably, since the bitmap intersection and bit compression use the same size of memory storage to store interval boundary, we omit the memory storage of interval boundary in memory requirements. The experimental results demonstrate that the proposed scheme performs better than bitmap intersection. Under $\beta=10^{-5}$, with the rule table size of 5K, we need only 164 Kbytes to store the bit compress algorithm compared with 12.5 Mbytes by bitmap intersection. And with the rule table size of 10K, 374 Kbytes is needed to store the bit compress algorithm compared with 48 Mbytes of bitmap intersection. When the rule number doubles, the memory consumption of bit compression increases (374/164) = 2.28 times, which approximates $N\cdot logN=2\cdot log2=2$. The difference is caused by storing the ndex table and related information. The memory consumption of bitmap intersection increases (48/12.5)=3.84 times, which approximates $N^2=4$. The simulation result shows our bit compression algorithm significantly decreases memory consumption while rule number increases and the proportion presents as we expect. The memory storage for bitmap intersection scales quadratically each time the number of rules doubles, while our bit compression is almost with rule numbers. Bit compression algorithm prevents memory

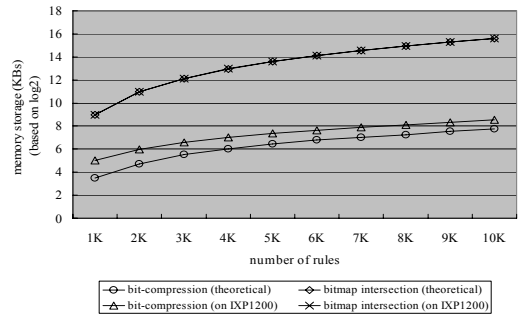


Fig. 5. Showing the variation of the memory requirements between the bit compression and the bitmap intersection algorithm under $\beta=10^{-5}$

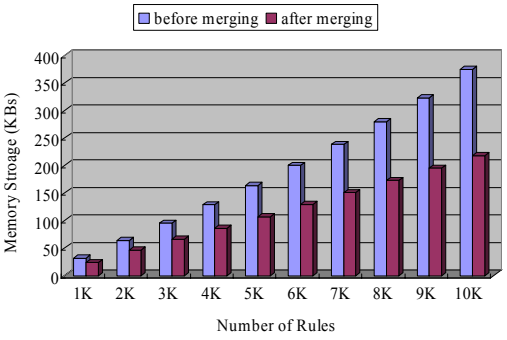


Fig. 6. Showing the improvement of memory storage by merging rule sets under $\beta=10^{-5}$.

exploration with large rule tables. The difference between theoretical measurement and implementation on IXP2400 is that the lengths of CBV, index list and DCV are a multiple of 32 bits when stored on IXP2400 for convenient memory access, creating a certain amount of space wastage. The memory storage with implementation on IXP2400 is higher than the theoretical storage requirements.

As noted previously, the space of index table can be further reduced by merging the rule sets. Figure 6 displays the total memory space consumed by the rule table of the bit compression scheme with and without merging under. As a result, 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). 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 exploration is prevented. For 2-dimensional rule table with 10K rules, the bit compression scheme still store the bit vector and index table using SRAM without SDRAM. Moreover, because most memory access cost of the bit compression scheme is expended to access the DCVs, we take advantage of memory hierarchy to store the DCVs in the smallest (4KB) but fastest scratchpad memory rather than SRAM. Storing the DCVs in the scratchpad memory facilitate decreasing memory access time for our bit compression scheme, while the bitmap intersection can only use SRAM or SDRAM. Therefore, although the times of memory access of bit compression are more

than bitmap intersection for the same size rule tables, the memory access performance of bit compression is better.

As mentioned above, the bit compression needs less memory access time than bitmap intersection. But notably, 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 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 can still outperform bitmap intersection as illustrated in Fig. 7. Figure 7 presents the packet transmission rates for bitmap intersection, aggregated bit vector and bit compression scheme with different size of rule tables without wildcards under β $=10^{-5}$ on IXP 2400. The minimum size packets (46 Bytes) were created as arriving data. In the bitmap intersection scheme, the rule tables are stored in SDRAM. In the bit compression scheme, the rule table is stored in SRAM only. Because the memory access time for reading a CBV and index list is less than reading a full bit vector. Although extra processing time for decompression is required for bit compression scheme. Our bit compression scheme outperforms bitmap intersection scheme. Moreover, since the length of CBVs and index lists almost remain a fixed value (according to maximum overlap), the transmission rate of our bit compression scheme remain constant. In contrast, the transmission rate of bitmap intersection however decreases linearly with number of rules.

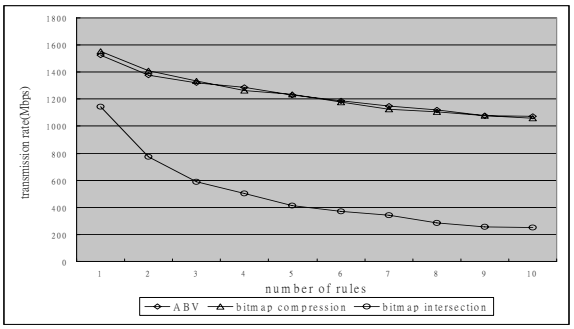
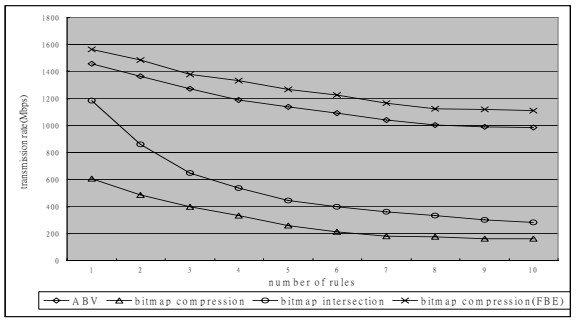
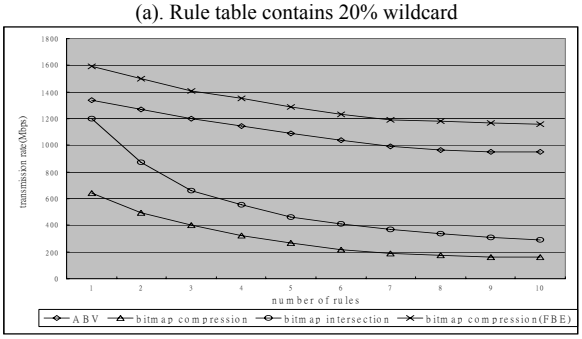


Fig 7. Showing the transmission rate of bitmap intersection algorithm, bit compress algorithm and aggregated bit vector algorithm where the rule table of various number of rule with no wildcard.

Figure 8 demonstrate the performance of bitmap intersection algorithm, aggregated bit vector algorithm and proposed bit compression algorithm with and without Fast Boolean Expansion in different amount of rules with a various percent of wildcard. When rules are not considered to have wildcards, the results shown in Figure 7 demonstrate that proposed bit compression algorithm outperforms bitmap intersection algorithms and is slide better than aggregated bit vector for the amount of rules 1000 to 10000. As previous mentioned DCV is used to reserve the wildcard information if rule database is considered to have wildcards. Therefore, in



(a). Rule table contains 20% wildcard



(b). Rule table contains 50% wildcard

Fig. 8. Showing the transmission rate of bitmap intersection algorithm, bitmap intersection algorithm (FBE), bit compress algorithm and aggregated bit vector algorithm where the rule table of various number of rule with various wildcard.

practice, we can even omit DCV and no need to access it if rule database do not comprise wildcards.

However, the result is contrary if the rule database is considered to have wildcards. Figure 8(a) and 8(b) with 20%, 50% wildcards respectively indicate the performance comparison between the three algorithms. As figure 8(a) and 8(b) indicate, expectably, bit compression algorithm has the poorest behavior compared to bitmap intersection algorithm and aggregated bit vector algorithm. In order to improve the performance, we employ the conception of Fast Boolean Expansion which is proposed in previous section. The results are also presented in figure 8(a) and 8(b).

As figure 8(a) and 8(b) indicates, bit compression algorithm with Fast Boolean Expansion has a better behavior than bit compression algorithm without Fast Boolean Expansion and bitmap intersection algorithm. It also outperforms aggregated bit vector algorithm slightly. This figure proves that proposed Fast Boolean Expansion indeed decreases the amount of memory access. For a 10000 rules with 50% wildcards example, bit compression algorithm with FBE takes at most 38 memory accesses since we had mentioned in previous section that there are at most 9 rule overlaps in each bit vector. But bitmap intersection algorithm takes 626 memory accesses and aggregated bit vector algorithm only need 30 memory accesses. Although aggregated bit vector algorithm has less memory access than proposed bit compression algorithm, the storage aggregated bit vector algorithm requires is huger than which proposed bit compression algorithm requires.

IV. CONCLUSION

Packet classification is an essential function of Internet security, virtual private networks, QoS and several network services. There are numerous various investigations have addressed this problem. This paper attempts to improve the original bitmap intersection algorithm, which has memory explosion problem for large rule table. This study introduces the notion of bit compression to significantly decrease the storage requirement, creating what we called 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 reducing the number of CRs further. For rule table with wildcared rules, the bit compression propose a novel idea, "Don't Care Vector" to save plenty 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 with bitmap intersection, the storage complexity is reduced from O (dN^2) of bitmap intersection to Θ ($dN \cdot logN$). In our experiment, our bit compression scheme only needs less than 380 Kbytes to store the 2-dimensional rule table with 10K rules, while bitmap intersection needs 48 Mbytes. Furthermore, comparing with memory access speed, our algorithm accesses average 96% less bits than bitmap intersection. Additionally, by exploiting the memory hierarchy to store the DCV and Fast Boolean Expansion, our bit compression scheme requires much less memory access time than bitmap intersection. Even though extra processing time for decompression is required for bit compression. The bit compression scheme with Fast Boolean Expansion still outperforms bitmap intersection scheme on the 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, no.3, pages 629-656, November 1996.

[2] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," Proc. of ACM Sigcomm, pages 147-160, September 1999.

[3] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," Proc. of ACM Sigcomm, pages 191-202, September 1998.

[4] F. Baboescu, G. Varghese, "Scalable Packet Classification," *Proc. of ACM Sigcomm,* pages 199-210, August2001.

[5] G. Zhang, H.J. Chao, J. Joung, "Fast Packet Classification Using Field-level Trie," Proc. of IEEE Globecom, vol. 6, 1-5, Pages 3201-3205 December 2003.

[6] "IXP2400 Network Processor: Datasheet", Part No. 301164-011 Fabruary 2004.

[7] "Intel IXP2XXX Product Line of Network Processors : Development Tools User's Guide", Part No. 278733-018, November 2004.

[8] C. R. Hsu, C. Chen, C. Y. Lin, "Fast Packet Classification Using Bit Compression," Proc. of IEEE Globecom, vol. 2, pages 739-743, November/December. 2005.