Scalable Packet Classification

Florin Baboescu and George Varghese, *Member, IEEE*

***Abstract*—Packet classification is important for applications such as firewalls, intrusion detection, and differentiated services. Existing algorithms for packet classification reported in the literature scale poorly in either time or space as filter databases grow in size. Hardware solutions such as TCAMs do not scale to large classifiers. However, even for large classifiers (say, 100 000 rules), any packet is likely to match a few (say, 10) rules. This paper seeks to exploit this observation to produce a scalable packet classification scheme called Aggregated Bit Vector (ABV). It takes the bit vector search algorithm (BV) described in Lakshman and Stidialis, 1998 (which takes linear time) and adds two new ideas, recursive aggregation of bit maps and filter rearrangement, to create ABV (which can take logarithmic time for many databases). We show that ABV outperforms BV by an order of magnitude using simulations on both industrial firewall databases and synthetically generated databases.**

I. Introduction

EVERY Internet router today can forward entering Internet messages (packets) based on the destination address. The 32-bit IP destination address is looked up in a table which then determines the output link on which the packet is sent. However, for a competitive advantage, many routers today choose to do additional processing for a specific subset of packets. Such additional processing includes providing differentiated output scheduling (e.g., Voice over IP packets are routed to a high priority queue), taking security-related actions (e.g., dropping packets sent from a certain subnet), load balancing (e.g., routing packets to different servers) and doing traffic measurement (e.g., measuring traffic between subnet pairs).

Although the details of the additional processing can vary greatly, a common requirement of all the functions above is that routers be able to *classify* packets based on packet headers into equivalence classes called *flows*. A flow is defined by a rule—for example the set of packets whose source address starts with prefix bits S, whose destination address is D, and which are sent to the server port for web traffic. Associated with each flow is an action which defines the additional processing—example actions include sending to a specific queue, dropping the packet, making a copy, etc.

Thus, packet classification routers have a database of rules, one for each flow type that the router wants to process differently. The rules are explicitly ordered by a network manager (or protocol) that creates the rule database. Thus, when a packet arrives at a router, the router must find a rule that matches the packet headers; if more than one match is found, the first matching rule is applied.

A. Scalable Packet Classification

This paper is about the problem of performing scalable packet classification for routers at wire speeds even as rule databases increase in size. Forwarding at wire speeds requires forwarding minimum sized packets in the time it takes to arrive on a link; this is crucial because otherwise one might drop important traffic before the router has a chance to know it is important [14]. With Internet usage doubling every six months, backbone link speeds have increased from OC-48 to OC-192 (2.4–10 Gb/s), and speeds up to OC-768 (40 Gb/s) are projected. Even link speeds at the network edge have increased from Ethernet (10 Mb/s) to Gigabit Ethernet.

Further, rule databases are increasing in size. The initial usage of packet classification for security and firewalls generally resulted in fairly small databases (e.g., the largest database in a large number of Cisco rule sets studied by [11] is around 1700). This makes sense because such rules are often entered by managers. However, in the very popular Differentiated Services [7] proposal, the idea is to have routers at the edge of the backbone classify packets into a few distinct classes that are marked by bits in the TOS field of the IP header. Backbone routers then only look at the TOS field. If, as seems likely, the DiffServ proposal reaches fruition, the rule sets for edge routers can grow very large.

Similarly, rulesets for edge routers that do load balancing [5] can grow very large. Such rulesets can potentially be installed at routers by a protocol; alternately, a router that handles several thousand subscribers may need to handle say, 10 rules per subscriber that are manually entered. It may be that such customer aggregation is the most important reason for creating large classifiers. Thus, we believe rule databases of up to 100 000 rules are of practical interest.

II. Previous Work

Previous work in packet classification [11], [12], [14], [20], [21] has shown that the problem is inherently hard. Most practical solutions use linear time [14] to search through all rules sequentially, or use a linear amount of parallelism (e.g., Ternary-CAMs [15]). Ternary CAMs are Content Addressable Memories that allow wildcard bits. While Ternary-CAMs are very common, such CAMs have smaller density than standard memories, dissipate more power, and require multiple entries to handle rules that specify ranges. Thus, CAM solutions are still expensive for very large rule sets of, say, 100 000 rules, and are not practical for PC-based routers [16]. Solutions based on caching [22] do not appear to work well in practice because of poor hit rates and small flow durations [18].

Another practical solution is provided by a seminal paper that we refer to as the Lucent bit vector scheme [14]. The idea is to first search for rules that match each relevant field F of the packet header, and to represent the result of the search as a bitmap of

Manuscript received August 7, 2002; revised January 2, 2004; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor E. Zegura. The work of F. Baboescu and G. Varghese was supported by the National Science Foundation under NSF Grant ANI 0074004.

The authors are with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: baboescu@cs.ucsd.edu; varghese@cs.ucsd.edu).

Digital Object Identifier 10.1109/TNET.2004.842232

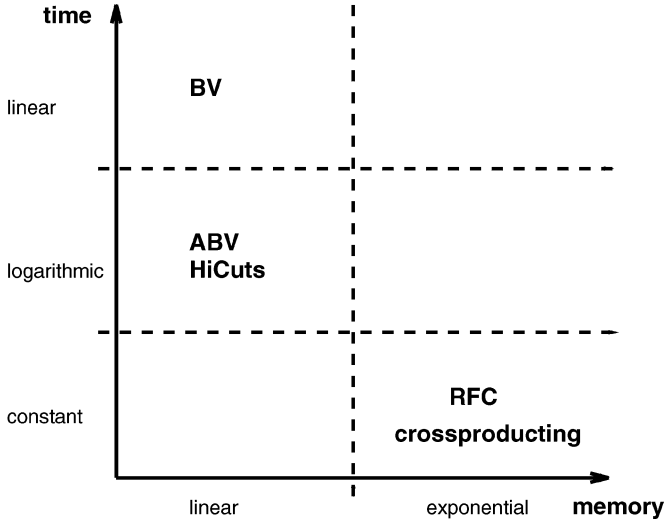


Fig. 1. Time-memory complexity for algorithmic solutions for the packet classification problem.

Filter	*RFC*	*HiCuts*	*BV*
DB_1	268,504	14,624	12,529
DB_2	444,287	48,347	9,627
DB_3	135,779	20,995	1,703
DB_4	173,037	18,027	4,487

Fig. 2. Total memory space occupied by the search structures in RFC [11], HiCuts [12], and the bit vector scheme (BV) [14]. The size is in memory words, one memory word is 32 bits. The filter databases are described in Fig. 9.

Filter	*RFC*	*HiCuts*	*BV*
DB_1	12	71	260
DB_2	12	99	150
DB_3	12	122	85
DB_4	12	122	70

Fig. 3. Total number of memory accesses for a worst case search in RFC [11], HiCuts [12] and the bit vector scheme (BV) [14]. One memory access is one word. One word is 32 bits. The filter databases are described in Fig. 9.

rules that match the packet in field F. Then the rules that match the full header can be found by taking the intersection of the bitmaps for all relevant fields F. While this scheme is still linear in the size of the rule set, in practice searching through a bitmap is fast because a large number of bits (up to 1000 in hardware, up to 128 bits in software) can be retrieved in one memory access. While the Lucent scheme can scale to around a reasonably large number of rules (say, 10 000) the inherently linear worst case scaling makes it difficult to scale up to large rule databases.

From a theoretical standpoint, it has been shown [14] that in its fullest generality, packet classification requires either $O(\log N^{k-1})$ time and linear space, or $\log N$ time and $O(N^k)$ space, where N is the number of rules, and k is the number of header fields used in rules. Thus, it comes as no surprise that the solutions reported in the literature for $k > 2$ either require large worst case amounts of space (e.g., crossproducting [20], RFC [11], HiCuts [12])[1] or time (e.g., bit vector search [14], backtracking [20]).

However, the papers by Gupta and McKeown [11], [12] introduced a major new direction into packet classification research. Since the problem is unsolvable in the worst case, they look instead for heuristics that work well on common rule sets. In particular, after surveying a large number of rule sets [11], they find that *multiple rule intersection* is very rare. In other words, it is very rare to have a packet that matches multiple rules. Since the examples that generate the worst case bounds entail multiple rule sets that intersect, it is natural to wonder whether there are schemes that are provably better given some such structural assumption on real databases.

Among the papers that report heuristics [11], [12], [21], the results on real databases are, indeed, better than the worst case bounds. Fig. 1 shows the time-memory relation for these type of schemes. As expected RFC occupies a memory space that is exponential in the number of rules. The HiCuts algorithm uses a memory space that is linear in the number of rules. However, the size of this space may be multiplied with a large constant in the case that the lists stored in the leaf nodes has a large number of duplications. The bit vector search algorithm has also a linear memory space utilization given by storing the bit vectors that are used in representing the matching rules associated with each prefix node in the search structures. The worst case complexity of the searches is linear in the number of rules in the case of the bit vector search algorithm. The complexity of the search is logarithmic in the number of rules in the case of HiCuts. RFC and crossproducting have a constant search time complexity, independent on the number of rules. Fig. 2 shows the memory-time tradeoff for RFC, HiCuts and the bit vector search scheme in the case of four different real life firewall databases that are described in Fig. 9.

Finally, there are several algorithms that are specialized for the case of rules on two fields (e.g., source and destination IP address only). For this special case, the lower bounds do not apply (they apply only for $k > 2$); thus hardly surprisingly, there are algorithms that take logarithmic time and linear storage. These include the use of range trees and fractional cascading [14], grid-of-tries [20], area-based quad-trees [4], and FIS-trees [8]. While these algorithms are useful for special cases, they do not solve the general problem. While the FIS trees paper [8] sketches an extension to $k > 2$ and suggests the use of clustering to reduce memory, there is a need to experimentally evaluate their idea on real (even small) multidimensional classifiers.

In summary, for the general classification problem on three or more fields, we find that existing solutions do not scale well in one of time or storage. Our paper uses the Lucent bit vector scheme as a point of departure since it already scales to medium size databases, and is amenable to implementation using either hardware or software. Our Aggregated Bit Vector (ABV) scheme adds two new ideas, *rule aggregation and rule rearrangement*, to enhance scalability.

A. *Previous Work in Efficient Representation of Sparse Sets*

Bit vectors are a natural way to represent sparse sets. However, operations such as set intersection take $O(n)$ using bit vectors, where n is the size of the universe which is represented.

[1] The tree search structure of the HiCuts [12] algorithm occupies a linear space in the number of rules. However each leaf node stores a list of rules that are a possible match. This list needs to be traversed in order to identify the matching rule of the search. In order to provide a good search throughput each of these lists must be stored in different memory spaces. As a result the information related to a rule may get duplicated into multiple lists and therefore increases the overall memory space.

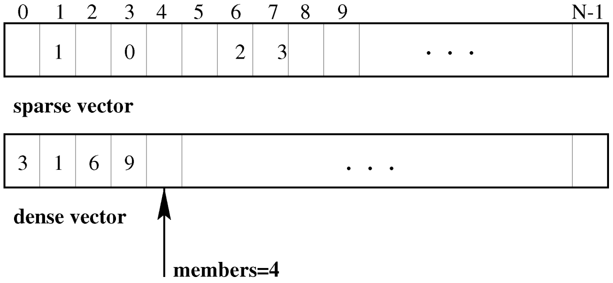


Fig. 4. Brigg's representation for a set made up of four elements {1, 3, 6, 9}. The total number of elements which may be represented is N.

An efficient sparse set representation for compiler applications (which allows set intersection in time proportional to the actual sizes of the sets being intersected) is described in [3].

The sparse set representation used in [3] has three components: two vectors, each with the length N, and a scalar that records the number of members in the set. The two vectors are called *sparse* and *dense*. The scalar value identifies the number of elements in the set as well as the number of elements in the *dense* vector. Fig. 4 shows the representation of a set made up of four elements; the maximum number of elements which may be represented is N. The values in the *dense* vector point to members in the *sparse vector*, which point back into the *dense* vector. If an element k is a member of the set, it must satisfy two conditions:

- $0 \leq sparse[k] < members$;
- $dense[sparse[k]] = k$.

We call this *Brigg's representation*.

The representation provides constant time implementations for operations like *find a member*, *add a member*, or *delete a member*. It also provides an asymptotic complexity of $O(n)$ where n is the number of members, for operations like *copy a set*, *compare sets*, *union*, *intersection*, and *difference of sets*. For these operations, bit vector representation takes $O(u)$ time where u is the size of the universe.

These results suggest that Brigg's representation could replace the bit vectors in the Lucent bit vector scheme and reduce time complexity. However, in Section VI we show that our ABV scheme can do much better than the Briggs scheme without being much worse in the worst case.

III. Problem Statement

Assume that information relevant to lookup is contained in k distinct packet *header fields*, denoted by $H_1, H_2, \ldots, H_k$, where each field is a bit string. For instance, the relevant fields for an IPv4 packet could be the Destination Address (32 bits), the Source Address (32 bits), the Protocol Field (8 bits), the Destination Port (16 bits), the Source Port (16 bits), and TCP flags (8 bits). Thus, the combination (D, S, TCP-ACK, 80, 2500), denotes the header of an IP packet with destination D, source S, protocol TCP, destination port 80, source port 2500, and the ACK bit set. Note that many rule databases allow the use of other header fields besides TCP/IP such as MAC addresses, and even Application (e.g., http) headers.

The *rule database* of a router consists of a finite sequence of rules, $R_1, R_2 \ldots R_N$. Each rule is a combination of k values, one for each header field. Each field in a rule is allowed three kinds of matches: *exact match, prefix match,* or *range match*. In an exact match, the header field of the packet should exactly match the rule field, for instance, this is useful for protocol and flag fields. In a prefix match, the rule field should be a prefix of the header field; this is useful for blocking access from a certain subnetwork. In a range match, the header values should lie in the range specified by the rule; this is useful for specifying port number ranges.

Each rule R_i has an associated action act^i, which specifies how to forward the packet matching this rule. The action specifies if the packet should be blocked. If the packet is to be forwarded, it specifies the outgoing link to which the packet is sent, and perhaps also a queue within that link if the message belongs to a flow with bandwidth guarantees.

We say that a packet P *matches* a rule R if each field of P matches the corresponding field of R—the match type is implicit in the specification of the field. For instance, let $R = (1010{*}, {*}, \text{TCP}, 1024 - 1080, {*})$ be a rule, with $act = drop$. Then, a packet with header $(10101\ldots111$, $11110\ldots000$, TCP, 1050, 3) matches F, and is therefore dropped. The packet $(10110\ldots000$, $11110\ldots000$, TCP, 80, 3), on the other hand, does not match R. Since a packet may match multiple rules, we define the matching rule to be the *earliest* matching rule in the sequence of rules[2].

We wish to do packet classification at wire speed for minimum sized packets and thus speed is the dominant metric. Because both modern hardware and software architectures are limited by memory bandwidth, it makes sense to measure speed in terms of memory accesses. It is also important to reduce the size of the data structure that is used to allow it to fit into the high speed memory. The time to add or delete rules is often ignored, but it is important for dynamic rule sets, that can occur in real firewalls. Our scheme can also be modified to handle fast updates at the cost of slightly increased search time.

IV. Towards a New Scheme

We introduce the ideas behind our scheme by first describing the Lucent bit vector scheme as our point of departure. Then, using an example rule database, we show our two main ideas: aggregation and rule rearrangement. In the next section, we formally describe our new scheme.

A. Bit Vector Linear Search

The Lucent bit vector scheme is a form of divide-and-conquer which divides the packet classification problem into k subproblems, and then combines the results. To do so, we first build k one-dimensional tries associated with each dimension (field) in the original database. We assume that ranges are either handled using a range tree instead of a trie, or by converting ranges to tries as shown in [20], [21]. An N-bit vector is associated with each node of the trie corresponding to a valid prefix. (Recall that N is the total number of rules).

Fig. 6 illustrates the construction for the simple two-dimensional example database in Fig. 5. For example, in Fig. 5, the second rule F_1 has 00* in the first field. Thus, the leftmost node

[2] Sometimes we refer to the lowest cost rule instead of the first matching rule. The two definitions are equivalent if the cost of a rule is its position in the sequence of rules

$Rule$	$Field_1$	$Field_2$
F_0	$00*$	$00*$
F_1	$00*$	$01*$
F_2	$10*$	$11*$
F_3	$11*$	$10*$
F_4	$0*$	$10*$
F_5	$0*$	$11*$
F_6	$0*$	$0*$
F_7	$1*$	$01*$
F_8	$1*$	$0*$
F_9	$11*$	$0*$
F_{10}	$10*$	$10*$

Fig. 5. A simple example with 11 rules on two fields.

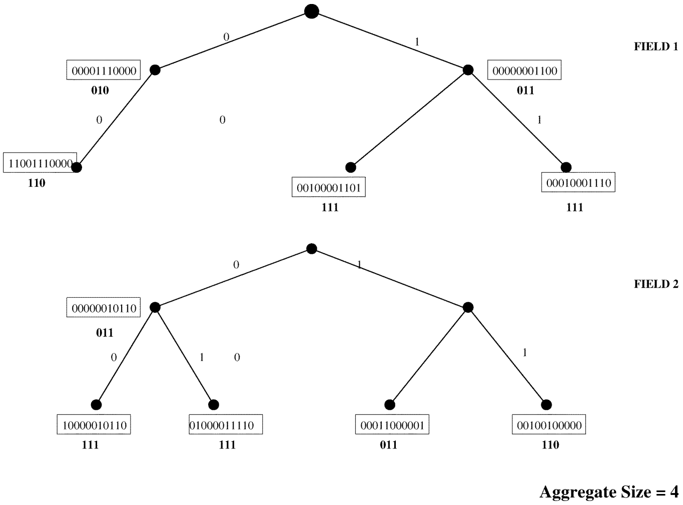


Fig. 6. Two tries associated with each of the fields in the database of Fig. 5, together with both the bit vectors (boxed) and the aggregate vectors (bolded) associated with nodes that correspond to valid prefixes. The aggregate bit vector has 3 bits using an aggregation size of 4. Bits are numbered from left to right.

in the trie for the first field, corresponds to 00*. Similarly, the Field 1 trie contains a node for all distinct prefixes in Field 1 of Fig. 5 such as 00*, 10*, 11*, 1*, and 0*.

Each node in the trie for a field is labeled with a N-bit vector. Bit j in the vector is set if the prefix corresponding to rule F_j in the database matches the prefix corresponding to the node. In Fig. 5, notice that the prefix 00* in Field 1 is matched by the values 00* and 0*, which correspond to values in rules 0, 1, 4, 5 and 6. Thus, the 11-bit vector shown behind the leftmost leaf node in the top most trie of Fig. 6 is 11001110000. For now, only consider the boxed bit vectors and ignore the smaller bit vectors below each boxed bit vector.

When a packet header arrives with fields $H_1, \ldots, H_k$, we do a longest matching prefix lookup (or narrowest range lookup) in each field i to get matches M_i and read off the resulting bit vectors $S(M_i)$ from the tries for each field i. We then take the intersection of $S(M_i)$ for all i, and find the lowest cost element of the intersection set. If rules are arranged in nondecreasing order of cost, all we need to do is to find the index of the first bit set in the intersected bit vector. However, these vectors have N bits in length; computing the intersection requires $O(N)$ operations. If W is the size of a word of memory than these bit operations are responsible for $(N \times k)/W$ memory accesses in the worst case. Note that the worst case occurs very commonly when a packet header does *not* match a single rule in the database.

B. Reducing Accesses by Aggregation

Recall that we are targeting the high cost in memory accesses which essentially scales linearly ($O(N)$) except that the constant factor is scaled down by the word size of the implementation. With a word size of up to 1000 in hardware, such a "constant" factor improvement is a big gain in practice. However, we want to do better by at least one order of magnitude, and remove the linear dependence on N. To this end, we introduce the idea of *aggregation*.

The main motivating idea is as follows. We hope that if we consider the bit vectors produced by each field, the set bits will be very sparse. For example, for a 100 000 rule database, if there are only 5 bits set in a bit vector of size 100 000, it seems a waste to read 100 000 bits. Why do we believe that bit vectors will be sparse? We have the following arguments:

- **Experience:** In the databases we have seen, every packet matches at most four rules. Similar small numbers have been seen in [12] for a large collection of databases up to 1700 rules.
- **Extension:** How will large databases be built? If they are based on aggregating several small classifiers for a large number of classifiers, it seems likely that each classifier will be disjoint. If they are based on a routing protocol that distributed classifiers based on prefix tables, then prefix containment is quite rare in the backbone table and is limited to at most six [21]. Again, if a packet matches a large number of rules, it is difficult to make sense of the ordering rules that give one rule priority over others.

The fact that a given packet matches only a few rules does not imply that the packet cannot match a large number of rules in all dimensions (because only a few matches could align properly in all dimensions). However, assume for now there is some dimension j whose bit vector is sparse.[3] To exploit the existence of such a sparse vector, our modified scheme, appends the bit vector for each field in each trie with an *aggregate bit vector*. First, we fix an aggregate size A. A is a constant that can be tuned to optimize the performance of the aggregate scheme; a convenient value for A is W the word size. Next, a bit i is set in the aggregate vector if there is at least one bit k set, $k \in [i \times A, (i+1) \times A]$. In other words, we simply aggregate each group of A bits in the Lucent bit vector into a single bit (which represents the OR of the aggregated bits) in the aggregate bit vector.

Clearly, we can repeat the aggregation process at multiple levels, forming a tree whose leaves are the bits in the original Lucent bit vector for a field. This can be useful for large enough N. However, since we deal with aggregate sizes that are at least 32, two levels of hierarchy can handle $32*32*32 = 32K$ rules. Using larger aggregate sizes will increase the N that can be handled further. Thus, for much of this paper, we will focus on one level (i.e., a single aggregate bit vector) or two levels (for a few synthetically generated large databases). We note that the only reason our results for synthetic databases are limited to 20 000 rules is because our *current testing* methodology (to check the worst case search time for all packet header combinations) does not scale.

[3] If this is not the case, as is common, then our second technique of rearrangements can make this assumption more applicable.

$Rule$	$Field_1$	$Field_2$
F_1	X	A_1
F_2	A_1	Y
F_3	X	A_2
F_4	A_2	Y
F_5	X	A_3
F_6	A_3	Y
F_7	X	A_3
$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$
F_{60}	A_{30}	Y
F_{61}	X	Y

Fig. 7. Example of a database with two-dimensional rules for which the aggregation technique without rearrangement behaves poorly. The size of the aggregate $A = 2$.

Why does aggregation help? The goal is to efficiently construct the bit map intersection of all fields without examining all the leaf bit map values for each field. For example, suppose that a given packet header matches only a small constant number of rules in each field. This can be determined in constant time, even for large N, by examining the top level aggregate bit maps; we then only examine the leaf bit map values for which the aggregate bits are set. Thus, intuitively, we only have to examine a constant number of memory words per field to determine the intersection because the aggregate vectors allow us to quickly filter out bit positions where there is no match. The goal is to have a scheme that comes close to taking $O(\log_A N)$ memory accesses, even for large N.

Fig. 6 illustrates the construction for the example database in Fig. 5 using an aggregate size $A = 4$. Let us consider a packet with Field 1 starting with bits 0010 and Field 2 starting with bits 0100. From Fig. 6 one can see that the longest prefix match is 00 for the first field and 01 for the second one. The associated bit vectors are: 11001110000 and 01000011110 while the aggregate ones (shown in bold below the regular bit vectors) are: 110 and 111. The AND operation on the two aggregate vectors yields 110, showing that a possible matching rule must be located only in the first 8 bits. Thus, it is not necessary to retrieve the remaining 4 bits for each field.

Notice that in this small example, the cost savings (assuming a word size of 4) is only two memory accesses, and this reduction is offset by the two memory accesses required to retrieve the bit maps. Larger examples show much bigger gains. Also, note that we have shown the memory accesses for *one* particular packet header. We need to efficiently compute the *worst case* number of memory accesses across *all* packet headers.

While aggregation does often reduce the number of memory accesses, in some cases a phenomenon known as *false matches* can increase the number of memory accesses to being slightly higher (because of the time to retrieve the aggregates for each field) than even the normal Lucent bit vector search technique.

Consider the database in Fig. 7 and an aggregation size $A = 2$. $A_1, \dots, A_{30}$ are all prefixes having the first five bits different from the first five bits of two IP addresses X and Y. Assume the arrival of a packet from source X to destination Y. Thus, the bit vector associated with the longest matching prefix in the Field 1 (source) trie is 1010101…101 and the corresponding bit vector in the Field 2 (destination) trie is 0101010…011. The aggregate bit vectors for both fields both using $A = 2$ are 111…1. However, notice that for all the ones in the aggregate bit vector (except the last one) the algorithm wrongly assumes that there might be a matching rule in the corresponding bit positions.

This is because of what we call a false match, a situation in which the result of an AND operation on an aggregate bit returns a one but there is no valid match in the group of rules identified by the aggregate. This can clearly happen because an aggregate bit set for field 1 corresponding to positions $p, \dots, p+A-1$ only means that *some* bit in those positions (e.g., $p+i$, $i < A$) has a bit set. Similarly, an aggregate bit set for field 2 corresponding to positions $p, \dots, p+A-1$ only means that some bit in those positions (e.g., $p+j$, $j < A$) has a bit set. Thus, a false match occurs when the two aggregate bits are set for the two fields but $i \neq j$. The worst case occurs when a false match occurs for every aggregate bit position.

For this particular example there are 30 false matches which makes our algorithm read 31×2 bits more than the Lucent bit vector linear search algorithm. We have used an aggregation size $A = 2$ in our toy example, while in practice A will be much larger. Note that for larger A, our aggregate algorithm will only read a small number of bits more than the Lucent bit vector algorithm even in the worst case.

C. Why Rearrangement of Rules can Help

Normally, in packet classification it is assumed that rules cannot be rearranged. In general, if Rule 1 occurs before Rule 2, and a packet could match Rule 1 and Rule 2, one must never rearrange Rule 2 before Rule 1. Imagine the disaster if Rule 1 says "Accept," and Rule 2 says "Deny," and a packet that matches both rules get dropped instead of being accepted. Clearly, the problem is that we are rearranging overlapping rules; two rules are said to overlap if there is at least one packet header that can match both rules.

However, the results from [11] imply that in real databases rule overlap is rare. Thus, if we know that a packet header can never match Rule 1 and Rule 2, then it cannot affect correctness to rearrange Rule 2 before Rule 1; they are, so to speak, "independent" rules. We can use this flexibility to try to group together rules that contribute to false matches into the same aggregation groups, so that the memory access cost of false matches is reduced.

Better still, we can rearrange rules arbitrarily *as long as we modify the algorithm to find all matches and then compute the lowest cost match.* For example, suppose a packet matched rules Rule 17, Rule 35, and Rule 50. Suppose after rearrangement Rule 50 becomes the new Rule 1, Rule 17 becomes the new Rule 3, and Rule 35 becomes the new Rule 6. If we compute all matches the packet will now match the new rules 1, 3, and 6. Suppose we have precomputed an array that maps from new rule order number to old rule order number (e.g., from 1 to 50, 3 to 17, etc.). Thus, in time proportional to the number of matches, we can find the "old rule order number" for all matches, and select the earliest rule in the original order. Once again the crucial assumption to make this efficient is that the number of worst case rules that match a packet is small. Note also that it is easy (and not much more expensive in the worst case) to modify a bit vector scheme to compute all matches.

$Rule$	$Field_1$	$Field_2$
F_1	X	A_1
F_2	X	A_2
F_3	X	A_3
$\ldots$	$\ldots$	$\ldots$
F_{30}	X	A_{30}
F_{31}	X	Y
F_{32}	A_1	Y
F_{33}	A_2	Y
$\ldots$	$\ldots$	$\ldots$
F_{60}	A_{29}	Y
F_{61}	A_{30}	Y

Fig. 8. Example of rearranging the database in Fig. 7 in order to improve the performance of aggregation. The size of the aggregate $A = 2$.

For example, rearranging the rules in the database shown in the database in Fig. 7, we obtain the rearranged database shown in Fig. 8. If we return to the example of packet header (X, Y), the bit vectors associated with the longest matching prefix in the new database will be $111\ldots11000\ldots0$ and $000\ldots01111\ldots1$ having the first 31 bits 1 in the first bit vector and the last 31 bits 1 in the second bit vector. However, the result of the AND operation on the aggregate has the first bit that is set in the position 16. This makes the number of bits necessary to be read for the aggregate scheme to be $16 \times 2 + 1 \times 2 = 34$ which is less than the number of the bits to be read for the scheme without rearrangement: $31 \times 2 = 62$.

The main intuition in Fig. 8 versus Fig. 7 is that we have "sorted" the rules by first rearranging all rules that have X in Field 1 to be contiguous; having done so, we can rearrange the remaining rules to have all values in Field 2 with a common value to be together (this is not really needed in our example). What this does is to localize as many matches as possible for the sorted field to lie within a few aggregation groups instead of having matches dispersed across many groups.

Thus, our paper has two major contributions. Our first contribution is the idea of using aggregation which, by itself, reduces the number of memory accesses by more than an order of magnitude for real databases, and even for synthetically generated databases where the number of false matches is low. Our second contribution is to show how can one reduce the number of false matches by a further order of magnitude by using rule rearrangement together with aggregation. In the rest of this paper, we describe our schemes more precisely and provide experimental evidence that shows their efficacy.

V. The ABV Algorithm

In this section, we describe our new ABV algorithm. We start by describing the algorithm with aggregation only. We then describe the algorithm with aggregation and rearrangement.

A. Aggregated Search

We start by describing more precisely the basic algorithm for a two-level hierarchy (only one aggregate bit vector), and without rearrangement of rules.

For the general k-dimension packet classification problem our algorithm uses the N rules of the classifier to precompute k tries, T_i, $1 \le i \le k$. A trie T_i is associated with field i from the rule database; it consists of a trie built on all possible prefix values that are found in field i in any rule in the rule database.

Thus, a node in trie T_i is associated with a valid prefix P if there is at least one rule R_l in the classifier having $R_l^i = P$, where R_l^i is the prefix associated with field i of rule R_l. For each such node two bit vectors are allocated. The first one has N bits and is identical to the one that is assigned in the BV algorithm. Bit j in this vector is set if and only if rule R_j in the classifier has P as a prefix of R_j^i. The second bit vector is computed based on the first one using aggregation. Using an aggregation size of A, a bit k in this vector is set if and only if there is at least one rule R_n, $A \times k \le n \le A \times k + 1 - 1$ for which P is a prefix of R_n^i. The aggregate bit vector has $\lceil N/A \rceil$ bits.

When a packet arrives at a router, a longest prefix match is performed for each field H_i of the packet header in trie T_i to yield a trie node N_i. Each node N_i contains both the bit vector ($N_i.bitVector$) and the aggregate vector ($N_i.aggregate$) specifying the set of filters or rules which matches prefix H_i on the dimension i. In order to identify the subset S of filters which are a match for the incoming packet, the AND of $N_i.aggregate$ is first computed.

Whenever position j is 1 in the AND of the aggregate vectors, the algorithm performs an AND operation on the regular bit vectors for each chunk of bits identified by the aggregate bit j (bits $A \times j, \ldots, A \times (j+1) - 1$). If a value of 1 is obtained for bit m, then the rule R_m is part of set S. However, the algorithm selects the rule R_t with the lowest value of t.

Thus, the simplest way to do this is to compute the matching rules from the smallest position to the largest, and to stop when the first element is placed in S. We have implemented this scheme. However, in what follows we prefer to allow arbitrary rearrangement of filters. To support this, we instead compute *all* matches. We also assume that each rule is associated with a cost (that can easily be looked up using an array indexed by the rule position) that reflects its position before rearrangement. We only return the lowest cost filter, i.e., the filter with the smallest position number in the original database created by the manager. As described earlier, this simple trick allows us to rearrange filters arbitrarily without regard for whether they intersect or not.

The pseudocode for this implementation is:

```
Get Packet $P(H_1,\ldots,H_k)$;
for $i \leftarrow 1$ to $k$ do
  $N_i \leftarrow LPMNode(Trie_i, H_i)$;
$Aggregate \leftarrow 11\ldots1$;
for $i \leftarrow 1$ to $k$ do
  $Aggregate \leftarrow Aggregate \bigcap N_i.aggregate$;
$BestRule \leftarrow Null$;
for $i \leftarrow 0$ to $sizeof(Aggregate) - 1$ do
  if ($Aggregate[i] == 1$)
    for $j \leftarrow 0$ to $A - 1$ do
      if ($\bigcap_{l=1}^{k} N_l.bitVect[i \times A + j] == 1$)
        if ($R_{i \times A + j}.cost < BestRule.cost$)
          $BestRule = R_{i \times A + j}$;
return BestRule;
```

B. A Sorting Algorithm for Rearrangement

One can see that false matches reduce the performance of the algorithm introduced in the previous section, with lines 10 to 13 in the algorithm being executed multiple times. In this section, we introduce a scheme which rearranges the rules such that, wherever possible, multiple filters which match a specific packet are placed close to each other. The intent, of course, is that these multiple matching filters are part of the same aggregation group. Note that the code of the last section allows us to rearrange filters arbitrarily as long as we retain their cost value.

Recall that Fig. 8 was the result of rearranging the original filter database from Fig. 7 by grouping together the entries having X as a prefix on the first field and then the entries having Y as a prefix in the second field. After rearranging entries, a query to identify the filter which matches the header (X, Y) of a packet takes about half the time it would take before rearrangement. This is because regrouping the entries reduces the number of false matches to zero.

To gain some intuition into what optimal rule arrangement should look like, we examined four real life firewall databases. We noticed that there were a large number of entries having prefixes of either length 0 or 32. This suggests a simple idea: if we arbitrarily pick a field and group together first the entries having prefixes of length 0 (such wildcard fields are very common), then the prefixes of length 1, and so on until we reach a group of all size 32 prefixes. Within each group of similar length prefixes, we sort by prefix value, thereby grouping together all filters with the same prefix value. For the field picked, this will clearly place all the wildcard fields together, and all the length 32 prefixes together, and so on. Intuitively, this rule generalizes the transformation from Figs. 7 and 8. In the rest of the paper, we refer to this process of rearrangement as *sorting on a field*.

Suppose we started by sorting on field i. There may be a number of filters with prefix X. Of course, we can continue this process recursively on some other field j, by sorting all entries containing entry X using the same process on field j. This clearly leaves the sorting on field i unchanged.

Our technique of moving the entries in the database creates large areas of entries sharing a common subprefix in one or more fields. If there are entries having fields sharing a common subprefix with different lengths, it separates them at a comfortable distance such that false matches are reduced.

A question each rearrangement scheme should address is correctness. In other words, for any packet P and any filter database C which, after rearrangement is transformed into a database C', the result of the packet classification problem having as entries both (C, P) and (C', P) should be the same. One can see that the ABV algorithm guarantees this because an entry is selected based on its cost. Note that (by contrast) in the BV scheme an entry is selected based on its position in the original database.

Our rearranging scheme uses a recursive procedure which considers the entries from a subsection of the original database identified through the *first* and *last* element. The rearrangement is based on the prefixes from the field *col* provided as an argument. The procedure groups the entries based on the length of the prefixes; for example first it considers the prefixes from field 1, and creates a number of groups equal to the number of different prefix lengths in field 1. Each group is then sorted so that entries having the same prefix are now adjacent. The entries having the same prefix then create subgroups; the procedure continues for each subgroup using the next fields that needs to be considered; the algorithm below considers fields in order from 1 to k. Note that one could attempt to optimize by considering different orders of fields to sort. We have not done so yet because our results seem good enough without this further degree of optimization.

A pseudocode description of the algorithm is given below. The algorithm is called initially by setting the parameters $first = 1, last = N, col = 1$

```
ARRANGE-ENT($first, last, col$)
if(there are no more fields) or ($first == last$) then return;
for (each valid size of prefixes) then
    Group together all the elements with the same size;
    Sort the previously created groups.
    Create subgroups made up of elements having the same prefixes on the field $col$
    for (each subgroup S with more than two elements) then
        Arrange-Ent($S.first, S.last, col + 1$);
```

VI. Aggregated Bit Vectors Versus Brigg's Representation for Sparse Sets

Ref. [3] introduced an efficient way to represent sparse sets which we describe in Section II-A. In this section, we analytically compare the two schemes.

Let W be the word size, P be the size of a pointer to a position in the *dense* vector in the Brigg's scheme (e.g., probably 16 bits to cover bitmaps of size greater than 256) and N be the number of rules (called size of universe in [3]). Recall that our model is different from Brigg's model: *we only count memory references*. Thus, if we read a word with 6 bits set and have to chase down each bit set, this operation still only has a cost of 1. This is because hardware (and even software) logic operating on registers is much faster than a memory access.

We first investigate the time complexity of a search operation.

Lemma 6.1: There exists rule sets (moderately dense) for which the Brigg's representation takes a factor of W^2/C more memory accesses than ABV, where C is a small constant no greater than 4.

For example, with $W = 32$ (typical word size) and $C = 4$, there are executions where ABV is 256 times faster than using Brigg's representation.

Proof: Consider a case in which we are intersecting two 64-bit vectors A and B. (The case may be generalized for other values of W) A has its first 32 bits set while B has the next 32 bits set. Therefore their intersection is null. By using aggregation with an aggregate size equal to 32, ABV gives an answer using only two memory accesses, reading the two aggregates and intersecting them. Let us consider next two N bit vectors A, B. A has the first half of its bits set, while B has the second half set. The sets they designate are disjoint. ABV produces an answer using $2 * ((N/W)/W)$ accesses, while using Brigg's representation the answer requires reading $N * (W/P)$ words. We consider $C = W/2P$.[4] □

Lemma 6.2: There is no execution on which ABV is more than a factor $\log_W N - 1$ *times worse than using Brigg's representation.*

The worst case executions occur when the bit vector is sparse. For example, with $W = 32$ and $N = 32K$ (pretty much the largest sizes one should consider and considered in our paper), this means that the Brigg's method can beat ABV by at most a factor of 2. Notice that for larger W, easily achieved in hardware, the comparison favors ABV further.

Proof: If ABV examines a word in the sparse bitmap, it must be that both aggregate bits indicate a 1, so both bitmaps have a nonzero position in that word, so using Brigg's representation must pay two memory accesses as ABV does. Thus, ABV can only pay more accesses by reading the aggregates. However for every bit set (a position that must be examined using Brigg's representation) ABV pays at most $\log_W N - 1$ memory accesses (ignoring the root which is an extra 1 that cancels with the scalar field in Briggs). □

Corollary 6.3: For $W = 32$ *and* $N < 32K$, *ABV is never more than a factor of two worse than Brigg's representation, while using Brigg's representation can be* 256 *times worse than using ABV.*

The following lemmas investigate the memory occupied by both implementation: ABV and the one based on Brigg's representation. We consider first the situation in which the rule set is dense. As we will see in the Section VII it is quite common to have sets which contain more than 20% of the total universe.

Lemma 6.4: There exists rule sets (very dense) for which the Brigg's representation takes a factor of 16 *more memory (using 16 bit pointers when every bit is set) than ABV.*

Proof: Immediate, considering that the dense vector contains pointers which are represented using P bits. In this case $P = 16$. □

Lemma 6.5: There exists no rule set for which ABV is more than a factor of $1/(W-1)$ *worse in memory size than using Brigg's representation.*

Proof: Consider the case in which there is only one element in the set. For this case using Brigg's representation one only needs to store a pointer to that element. However, ABV pays for storing the aggregate. The overall size of the aggregate is $(N/W)/W$. Therefore the overall memory space occupied by ABV is $(N/W) + (N/W)/W$ while using Brigg's representation the memory size is $(N/W) + 1$. □

Corollary 6.6: For $W = 32$, *using Brigg's representation can be* 16 *times worse in storage than, while for any database of rules using ABV uses at most* 3% *more memory than using Brigg's representation.*

So, which of the representations should be used for the multidimensional packet classification problem? The authors in [11] noticed that despite an increase in the number of rules in the packet classification, there are only a small, limited number of matching rules. Therefore in a set representation one can only say that the result set is sparse. However, in each individual dimension we have observed multiple matches because of lots of wildcards.

Therefore the result set in each dimension is not sparse; thus it does not favor the use of Brigg's representation. It is very common to have about 20% matching rules in each dimension, because of a large number of zero length (wildcarded) prefixes. As a result we conclude that ABV is a better solution for multidimensional packet classification.

VII. Evaluation

In this section we consider how the ABV algorithm can be implemented, and how it performs on both real firewall databases and synthetically created databases. Note that we need synthetically created databases to test the scalability of our scheme because real firewall databases are quite small.

First, we consider the complexity of the preprocessing stage and the storage requirements of the algorithm. Then, we consider the query performance and we relate it to the performance of the BV algorithm. The speed measure we use is the worst case number of memory accesses for search across *all possible packet headers.* This number can be computed without considering all possible packets because packet headers fall into equivalence classes based on distinct cross products [20]; a distinct cross-product is a unique combination of longest matching prefix values for each header field.

Since each packet that has the same cross-product is matched to the same node N_i (in trie T_i) for each field i, each packet that has the same cross-product will behave identically in both the BV and ABV schemes. Thus, it suffices to compute worst case search times for all possible cross-products. However, computing all crossproducts for a database of 20 000 rules took 6 hours on a modern SPARC. We improved the testing algorithm from hours to minutes using a clever idea used in the RFC scheme [11] to equivalence cross-products while computing crossproducts pairwise. Note that these large times are the times required to certify the worst case behavior of our algorithm, not the time for a search.

We have seen that false matches can cause our ABV algorithm (in theory) to have a poorer worst behavior than BV. However through our experiments we show that *ABV outperforms BV by more than an order of magnitude on both real life databases and synthetic databases*.

A. ABV Preprocessing

We consider the general case of a k dimension classifier.We build k tries T_i, $1 \leq i \leq k$, one for each dimension. Each trie has two different types of nodes depending if they are associated or not with valid prefixes. The total number of nodes in the tries is on the order of $O(N \times k)$, where N is the number of entries in the classifier (i.e., rule database). Two bit vectors are associated with each valid prefix node. One bit vector is identical with the one used in BV scheme and requires $\lceil N/WordSize \rceil$ words of data. The second bit vector is the aggregate of the first one; it contains $\lceil N/A \rceil$ bits of data which means that it requires $\lceil N/(A \times WordSize) \rceil$ words of memory (A is the size of the aggregate). Building both bit vectors requires an $O(N)$ pass through the rule database for each valid node of the trie. Thus, the preprocessing time is $O(N^2 k)$.

[4] This example also suggests why rearrangement helps. It allows us to pay a cost of only one memory access to look at an aggregate pointing to a word containing lots of 1's that we do not need to examine.

Filter	*Number of rules specified by:*	
	Range	*Prefix*
DB_1	266	1640
DB_2	279	949
DB_3	183	531
DB_4	158	418

Fig. 9. Sizes of the firewall databases we use in the experiments.

One can easily see from here that the memory requirements for ABV are slightly higher than that of BVS; however for an aggregate size greater than 32 (e.g., software), ABV differs from BV by less than 3%, while for an aggregate size of 500 (e.g., hardware), it is below 0.2%.

The time required for insertion or the deletion of a rule in ABV is of the same complexity as BV. This is because the aggregate bit vector is updated each time the associated bit vector is updated. Note that updates can be expensive because adding a filter with a prefix X can potentially change the bit maps of several nodes. However, in practice it is rare to see more than a few bitmaps change, possibly because filter intersection is quite rare [11]. Thus, incremental update, though slow in the worst case, is quite fast on the average.

B. Experimental Platform

We used two different types of databases. First we used a set of four industrial firewall databases. For privacy reasons we are not allowed to disclose the name of the companies or the actual databases. Each entry in the database contains a 5-*tuple* (source IP prefix, destination IP prefix, source port number(range), destination port number(range), protocol). We call these databases $DB_1, \ldots, DB_4$. The database characteristics are presented in Fig. 9.

The third and fourth field of the database entries are represented by either port numbers or range of port numbers. We convert them to valid prefixes using the technique described in [20]. The following characteristics have important effects on the results of our experiments.

1) Most prefixes have either a length of 0 or 32. There are some prefixes with lengths of 21, 23, 24 and 30.
2) No prefix contains more than four matching subprefixes for each dimension.
3) The destination and source prefix fields in roughly half the rules were wildcarded (by contrast, [8] only assumes at most 20% of the rules have wildcards in their experiments), and roughly half the rules have $\geq$ 1024 in the port number fields. Thus, the amount of overlap within each dimension was large.
4) No packet matches more than four rules.

The second type of databases are randomly generated two and five field (sometimes called two- and five-dimensional) databases using random selection from five publicly available routing tables ([13]). We used the snapshot of each table taken on September 12, 2000. An important characteristic of these tables is the prefix length distribution, described in Fig. 10.

Routing Table	*Prefix Lengths:*					
	8	*9 to 15*	*16*	*17 to 23*	*24*	*25 to 32*
$Mae-East$	10	133	1813	9235	11405	58
$Mae-West$	15	227	2489	11612	16290	39
$AADS$	12	133	2204	10144	14704	55
$PacBell$	12	172	2665	12808	19560	54
$Paix$	22	560	6584	28592	49636	60

Fig. 10. Prefix length distribution in the routing tables, September 12, 2000.

Recall that the problem is to generate a synthetic database that is larger than our sample industrial databases to test ABV for scalability. The simplest way to generate a two-dimensional classifier of size N would be to iterate the following step N times: in each step, we randomly pick a source prefix and a destination prefix from any of the five routing tables. This generation technique is unrealistic because real routing databases have at most one prefix of length 0. Thus, simple random generation is very unlikely to generate rules with zero length prefixes, whereas zero length prefixes are very common in real firewall rule databases.

For more realistic modeling, we also allow a controlled injection of rules with zero length prefixes, where the injection is controlled by a parameter that determines the percentage of zero length prefixes. For example, if the parameter specifies that 20% of the rules have a zero length prefix, then in selecting a source or destination field for a rule, we first pick a random number between 0 and 1; if the number is less than 0.2 we simply return the zero length prefix; else, we pick a prefix randomly from the specified routing table.

A similar construction technique is also used in [8] though they limit wild card injection to 20%, while our experiments have used up to 50% wild card injection. [8] also uses another technique based on extracting all pairs of source-destination prefixes from traces and using these as filters. They show that the two methods differ considerably with the random selection method providing better results because the trace method produces more overlapping prefix pairs. However, rather than using an ad hoc trace, we prefer to stress ABV further by adding a controlled injection of groups of prefixes that share a common prefix to produce more overlapping prefix pairs.

When we inject a large amount of zero length prefixes and subprefixes, we find that ABV without rearrangement begins to do quite poorly, a partial confirmation that we are stressing the algorithm. Fortunately, ABV with rearrangement still does very well. Finally, we did some limited testing on synthetic five-dimensional databases. We generated the source and destination fields of rules as in the synthetic two-dimensional case; for the remaining fields (e.g., ports) we picked port numbers randomly according to the distribution found in our larger real database. Once again, we find that ABV scales very well compared to BV.

C. Performance Evaluation on Industrial Firewall Databases

We experimentally evaluate ABV algorithm on four industrial firewall databases described in Fig. 9. The rules in the databases are converted into prefix format using the technique described in [17]. The memory space that is used by each of them can be estimated based on the number of nodes in the tries, and the number of nodes associated with valid prefixes. We provide these values in Fig. 11. A node associated with a valid prefix carries a bit vector of size equal to $\lceil N/32 \rceil$ words and an aggregate bit vector of size $\lceil N/(32 \times 32) \rceil$ words. We used a word size equal to 32;

Filter	*No.of Nodes*	*No. of Valid Prefixes*
DB_1	980	188
DB_2	1242	199
DB_3	805	127
DB_4	873	143

Fig. 11. Total number of nodes in the tries and the total number of nodes associated with valid prefixes for the industrial firewall databases.

Filter	*BV*	*ABV*		
		unsorted	*One Field Sorted*	*Two Fields Sorted*
DB_1	260	120	75	65
DB_2	150	110	50	50
DB_3	85	60	50	50
DB_4	75	55	45	45

Fig. 12. Total number of memory accesses in the worst case scenario for the industrial firewall databases. Several cases are considered: databases with no rule rearrangement, databases sorted on one field only, and databases sorted on two fields.

we also set the size of the aggregate to 32. We used only one level of aggregation in this experiment.

Our performance results are summarized in Fig. 12. We consider the number of memory accesses required by the ABV algorithm once the nodes associated with the longest prefix match are identified in the trie in the worst case scenario. The first stage of finding the nodes in the tries associated with the longest prefix matching is identical in both algorithms ABV and BV (and depends on the longest prefix match algorithm used; an estimate for the fastest algorithms is around three to five memory accesses per field). Therefore, we do not consider it in our measurements. The size of a memory word is 32 bits for all the experiments we considered.

The results show that ABV without rearrangement outperforms BV, with the number of memory accesses being reduced by a factor of 27% to 54%. By rearranging the elements in the original database, the performance of ABV can be increased by further reducing the number of memory accesses by a factor of 40% to 75%. Our results also show that for the databases we considered it was sufficient to sort the elements using only one field.

D. Experimental Evaluation on Synthetic Two-Dimensional Databases

Thus, on real firewall databases our ABV algorithm outperforms the BV algorithm. In this section we evaluate how our algorithm might behave with larger classifiers. Thus, we are forced to synthetically generate larger databases, while injecting a controlled number of zero length prefixes as well as a number of prefixes that have subprefixes. As described earlier, we create our synthetic two-dimensional database of prefixes from publically available routing tables [13] whose characteristics are listed in Fig. 10. We show results for databases generated using MAE-EAST routing table. The results for databases generated using the other routing tables are similar and are not reproduced here.

Effect of Zero-Length Prefixes: We first consider the effect of prefixes of length zero on the worst case number of memory accesses. Entries containing prefixes of length zero are randomly generated as described earlier. The results are displayed in Fig. 13. The presence of prefixes of length zero randomly distributed through the entire database has a heavy impact on the number of memory accesses. If there are no prefixes of length zero in our synthetic database, the number of memory accesses for a query using ABV scheme is a factor of 8 to 27 times less than the BV scheme.

However, by inserting around 20% worth of prefixes of length zero in the database we found that the ABV scheme (without rearrangement) needed to read all the words from both the aggregate and the bit vector; in such a scenario, clearly the BV scheme does better by a small amount. Fortunately, by sorting the entries in the database using the technique described in Section V.B, the number of memory accesses for the worst case scenario for ABV scheme is reduced to values close to the values of a database (of the same size) without prefixes of length zero. Note that the sorted ABV scheme reduces the number of memory accesses by more than 20 compared to the BV scheme, with the difference growing larger as the database size gets larger.

Fig. 14 graphs the distribution of the number of memory accesses as a function of number of entries in the synthetic database. The databases are generated using randomly picked prefixes from the MAE-East routing table, and by random injection of prefixes of length zero. The line with stars represents the linear scaling of the Lucent (BV) scheme. Notice that *unsorted* ABV with more than 20% injection of zero length prefixes has slightly worse overhead than the BV scheme. However, the overhead of the sorted ABV scheme with up to 50% zero length injection (see the bottom lines) appears to increase very slowly, possibly indicating logarithmic scaling.

Injecting Subprefixes: A second feature which directly affects the overall performance of our algorithm is the presence of entries having prefixes which share common subprefixes. These prefix groups effectively create subtries whose root is is the longest common subprefix of the group. Let W be the depth of the subtrie, and consider a filter database with k dimensions. It is not hard to see that if we wish to stress the algorithm, we need to increase W. How do we generate a synthetic database for a given value of W?

To do so, we first extract a set of 20 prefixes having length equal to 24. We call this set L. L is chosen so no two elements in L share the same 16 – bit prefix. In the second step, for each element in L we insert eight other elements with prefix length in the range $(24 - W), \ldots, 23$. These elements are subprefixes of the element in L.

We generate the filter database by randomly picking prefixes from both the routing table and from the newly created set L. We can control the rate with which elements from L are inserted in the filter database. We measure the effect of different tries heights W as well as the effect of having different ratios of such elements. The results are displayed in Figs. 15, 16, and 18. For example, Fig. 18 compares the linear scaling of the Lucent (BV) scheme to the *sorted* ABV scheme. The figure shows that when the percentage of subprefixes sharing a common prefixes increases to very large values, the overhead of ABV also increases, though much more slowly than the BV scheme.

The tables show that, at least for a model of random insertion, *the height W does not have a large impact on the number*

DB Size	*BV*	*Percentage of prefixes of length zero; sorted(s)/unsorted(u)*												
		0	$1u$	$1s$	$2u$	$2s$	$5u$	$5s$	$10u$	$10s$	$20u$	$20s$	$50u$	$50s$
$1K$	64	8	12	10	26	10	54	10	66	12	66	12	66	10
$2K$	126	10	28	14	58	12	84	14	126	14	130	14	130	14
$5K$	314	16	50	18	76	18	216	20	298	20	324	22	324	18
$10K$	626	26	78	30	196	28	426	34	588	34	644	32	646	30
$20K$	1250	48	148	48	346	50	860	52	1212	54	1288	52	1292	52

Fig. 13. Worst case total number of memory accesses for synthetic two-dimensional databases of various sizes, with a variable percentage of zero prefixes. The databases were generated using the MAE-EAST routing table [13].

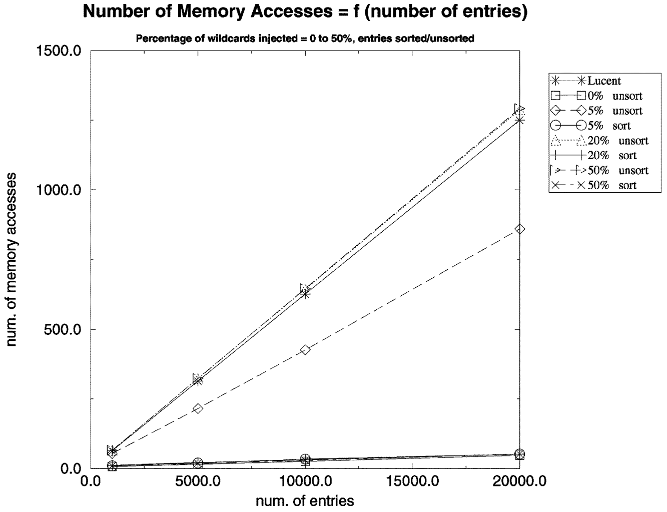


Fig. 14. Number of memory accesses as a function of the number of database entries. The ABV scheme outperforms the BV scheme by a factor greater than 20 on a sorted synthetic database having prefixes of length zero randomly inserted. The synthetic databases were generated using the MAE-EAST routing table [13].

of false matches. A slight increase in this number can be seen only when there are about 90% of such elements inserted in the measured database. We consider next the ratio of such elements to the total number of prefixes in the database. Their impact on the total number of memory accesses is lower than the impact of prefixes of length zero. When their percentage is roughly 50%, the number of memory accesses using the ABV algorithm (without sorting) is about 10 times lower than the number of memory accesses using the BV algorithm. This number is again improved by a factor of about 30% by sorting the original database. These numbers were for a database with $20K$ entries.

1) Evaluating ABV With Different Word Sizes: Our measurements until now have compared ABV versus BV using a word size equal to 32 bits. However, in hardware the clear power of BV is using a larger word size of up to 1000 bits using a wide internal bus. We analyzed the worst case scenario for both ABV and BV using different word sizes between 128 and 1024 bits. In all cases ABV outperformed BV. The results for a 20 000 rules two-dimensional synthetic generated database are given in Fig. 17. However, it is interesting that the worst case gain of ABV over BV seems to decrease from a factor of nearly ten (using 128 bit words) to a factor of two (using 1024 bit words). This makes intuitive sense because with larger bitmaps more bits can be read in a single memory access. We suspect that with larger word sizes one would see larger gains only when using larger rule databases.

2) Evaluating ABV With Two Levels of Aggregation: So far our version of ABV for two-dimensional databases has used only one level of aggregation. Even for a 32 000 rule database, we would use an aggregate bit vector of length equal to $32\,000/32 = 1000$. However, if only a few bits are set in such an aggregate vector, it is a waste of time to scan all 1000 bits. The natural solution, for aggregate bit vectors greater than A^2 (1024 in our example), is to use a *second* level of hierarchy. With $A = 32$, a second level can handle rule databases of size equal to $32^3 = 32K$. Since this approaches the limits of the largest database that we can test (for worst case performance), we could not examine the use of any more levels of aggregation.

To see whether two levels provides any benefit versus using one level only, we simulated the behavior of the two-level ABV algorithm on our larger synthetic databases. (It makes no sense to compare the performance of two levels versus one level for our small industrial databases.). For lack of space, in Fig. 19 we only compare the performance of two versus one level ABV on synthetic databases (of sizes 5000, 10 000, and 20 000) generated from MAE-EAST by injecting 0% to 50% prefixes of zero length. In all cases we use the ABV algorithm with rearrangement (i.e., the best case for both one and two levels).

The results show that using an extra level of aggregation reduces the worst number of memory accesses by 60% for the largest databases. For the smallest database (5000) the improvement is marginal, which accords with our intuition — although the algorithm does not touch as many leaf bits for the database of size 5000, this gain is offset by the need to read another level of aggregate bits. However, at a database size of 10 000 there is a clear gain. The results suggest that the number of memory accesses for a general multilevel ABV can scale logarithmically with the size of the rule database, allowing potentially very large databases.

E. Performance Evaluation Using Synthetic Five-Dimensional Databases

So far we have tested scalability only on randomly generated two-dimensional databases. However, there are existing schemes such as grid-of-tries and FIS trees that also scale well for this special case. Thus, in this section we briefly describe results of our tests for synthetic five-dimensional databases.

The industrial firewall databases we use have a maximum size of 1640 rules, making them infeasible for scalability tests. To avoid this limitation, we generated synthetic five-dimensional databases using the IP prefix addresses from MAE-EAST as in the two-dimensional case, and port number ranges and protocol fields using the distributions of values and ranges found in the industrial firewall databases.

DB Size	*BV*	$W = 4$					$W = 6$					$W = 8$				
		1	10	20	50	90	1	10	20	50	90	1	10	20	50	90
$1K$	64	8	10	20	40	52	8	12	26	38	56	8	12	20	36	52
$5K$	314	16	28	56	124	144	16	32	56	126	148	16	30	50	120	162
$10K$	626	28	54	96	228	214	26	50	96	244	234	26	50	94	194	226
$20K$	1250	48	88	168	308	254	48	90	154	274	292	48	92	176	304	326

Fig. 15. Worst case total number of memory accesses for synthetic two-dimensional databases having injected a variable percentage of elements which share a common subprefix. The databases are *not sorted*. W is is the depth of the subtrie created by these elements. The values below W denote the percentage of injection. The values labeled by BV estimate the number of memory accesses using the BV scheme. All the other values are associated with the ABV scheme. The synthetic databases were generated using the MAE-EAST routing table [13].

DB Size	$W = 4$					$W = 6$					$W = 8$				
	1	10	20	50	90	1	10	20	50	90	1	10	20	50	90
$1K$	6	12	16	34	54	8	12	18	36	48	8	12	16	36	48
$5K$	16	26	48	106	136	16	30	44	112	136	16	30	46	116	138
$10K$	26	46	82	176	154	26	52	80	166	176	26	48	84	198	178
$20K$	48	78	146	212	138	48	100	142	224	208	48	88	136	232	170

Fig. 16. Worst case total number of memory accesses for synthetic two-dimensional databases having injected a variable percentage of elements which share a common subprefix. The databases are *sorted*. W is is the depth of the subtrie created by these elements. The values below W denote the percentage of injection. All the values are associated with the ABV scheme. The synthetic databases were generated using the MAE-EAST routing table [13].

Word Size	*BV*	*ABV*
128	314	34
256	158	28
512	80	26
1024	40	20

Fig. 17. ABV versus BV scheme for a two-dimensional synthetic generated database with 20 000 rules. The synthetic database was generated using the MAE-EAST routing table. We consider an aggregate size of 32, and different word sizes between 128 and 1024 bits.

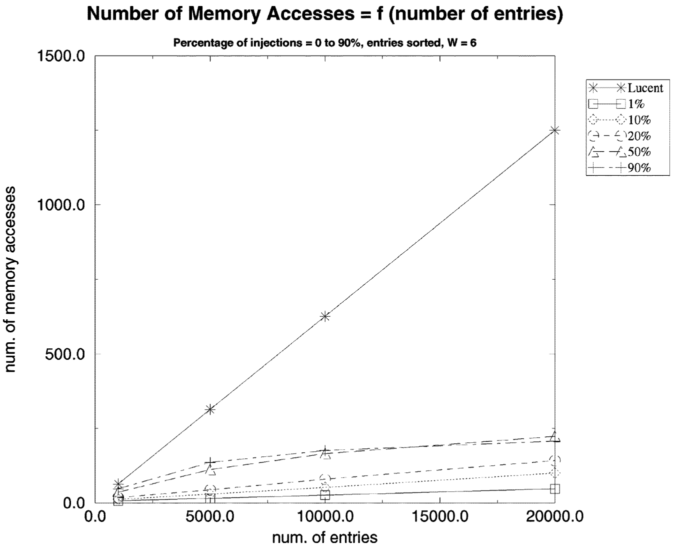


Fig. 18. Number of memory accesses as a function of the number of database entries. Synthetic databases generated using MAE-EAST routing table and by randomly inserting group of elements which are sharing a common subprefix. $W = 6$ is the depth of the subtrie created by these elements. The percentage of subprefixes injected varies from 0 to 90%. The ABV scheme outperforms the BV scheme by a factor of *2 to 7* if the databases are sorted.

Our results are shown in Fig. 20. The ABV scheme outperforms the BV scheme by more than one order of magnitude. The only results we have shown use no wildcard injection. The results for larger wildcard injections are similar to before (though sorting on multiple fields appears to be even more crucial). Note that for a five-dimensional database with 21 226 rules the Lucent (BV) scheme required 3320 memory accesses while ABV with an aggregation size of 32 required only 140 memory accesses.

VIII. Conclusions

While the Lucent BV scheme [14] is fundamentally an $(O(N))$ scheme, the use of an initial projection step allows the scheme to work with compact bitmaps. Taken together with memory locality, the scheme allows a nice hardware or software implementation. However, the scheme only scales to medium size databases.

Our paper introduces the notions of aggregation and rule rearrangement to make the Lucent BV scheme more scalable, creating what we call the ABV scheme. The resulting ABV scheme is at least an order of magnitude faster than the BV scheme on all tests that we performed. The ABV scheme appears to be equally simple to implement in hardware or software. In hardware, the initial searches on the individual tries can be pipelined with the remainder of the search through the bitmaps. The searches in the levels of the bitmap hierarchy can also be pipelined.

In comparing the two heuristics we used, aggregation by itself is not powerful enough. For example, for large synthetically generated databases with 20% of the rules containing zero length prefixes, the performance of ABV without rearrangement grew to be slightly worse than BV. However, the addition of sorting again made ABV faster than BV by an order of magnitude. A similar effect was found for injecting subprefixes. However, a more precise statement of the conditions under which ABV does well is needed.

We evaluated our implementation on both industrial firewall databases and synthetically generated databases. We stressed ABV by injecting prefixes that appear to cause bad behavior. Using only 32-bit memory accesses, we were able to do packet classification in a 20 000 rule random two-dimensional databases (with almost half the entries being wild cards) using 20 accesses using two levels of hierarchy. By contrast, the Lucent algorithm took 1250 memory accesses on the same database. Similarly, for a random five-dimensional database of 20 000 rules the Lucent scheme required 3320 memory accesses while ABV with one level of hierarchy required only 140 memory accesses. Taken together with wider memory accesses possible using either cache lines in software or wide busses in hardware, we believe our algorithm should have sufficient speed for OC-48 links even for large databases using SRAM.

Experiment	*No. Of Entries = 5000*		*No. Of Entries = 10000*		*No. Of Entries = 20000*	
	One Level	*Two Levels*	*One Level*	*Two Levels*	*One Level*	*Two Levels*
0% stars	16	14	26	14	46	18
1% stars	18	14	30	20	52	22
5% stars	20	14	30	18	52	26
10% stars	22	20	32	22	50	22
50% stars	20	18	30	18	50	20

Fig. 19. Number of memory accesses for the ABV algorithm with one and two levels of aggregation. The databases are *sorted* and are generated using the MAE-EAST routing table [13] using various percentages of wildcard injection and various sizes.

Size	*BV*	*ABV - 32*
3722	585	40
7799	1220	65
21226	3320	140

Fig. 20. ABV versus BV scheme for five-dimensional synthetically generated databases. The synthetic databases were generated using the MAE-EAST routing table, and using port number ranges and protocol numbers from the industrial firewall databases. All results use an aggregate size of 32.

While most of the paper used only one level of hierarchy, we also implemented a two-level hierarchy for the large synthetically generated databases. The second level of hierarchy does improve the number of memory accesses for large classifiers, which suggests that the scaling of ABV is indeed logarithmic. It also suggests that ABV is potentially useful for the very large classifiers that may be necessary to support such applications as DiffServ and content-based Load Balancing that are already being deployed.

Finally, the use of aggregate bitmaps may be useful in other networking and system contexts as well. For example, the *select()* mechanism in UNIX works well for small scale applications, but does not scale to the large number of file descriptors used by large web servers or proxies [2]. One reason for the poor performance of *select()* is that on each call the application must inform the operating system kernel about the set of descriptors of interest, where the set is encoded using bitmaps. For a large number of descriptors, searching through the bitmap for set bits can be time consuming. Aggregate bitmaps may reduce search and copy times.

References

[1] M. L. Bailey, B. Gopal, M. Pagels, L. L. Peterson, and P. Sarkar, "Pathfinder: A pattern-based packet classifier," in *Proc. 1st Symp. Operating Systems Design and Implementation*, Nov. 1994.

[2] G. Banga and J. C. Mogul, "Scalable kernel performance for internet servers under realistic loads," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 1998.

[3] P. Briggs and L. Torczon, "An efficient representation for sparse sets," *ACM Lett. Program. Lang. Syst.*, vol. 2, Mar.-Dec. 1993.

[4] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space decomposition techniques for fast layer-4 switching," in *Proc. Conf. Protocols for High Speed Networks*, Aug. 1999.

[5] Cisco ArrowPoint Communications (2000). [Online]. Available: http://www.arrowpoint.com

[6] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. ACM SIGCOMM*, Aug. 1996.

[7] IETF Differentiated Services (Diffserv) Working Group (2000). [Online]. Available: http://www.ietf.org/html.charters/diffserv-charter.html

[8] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE INFOCOM*, vol. 1, Mar. 2000, pp. 397–413.

[9] G. Malan and F. Jahanian, "An extensible probe architecture for network protocol measurement," in *Proc. ACM SIGCOMM*, Sep. 1998.

[10] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: A software architecture for next generation routers," in *Proc. ACM SIGCOMM*, Sep. 1998.

[11] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, Sep. 1999.

[12] ——, "Packet classification using hierarchical intelligent cuttings," in *Proc. Hot Interconnects VII*, Stanford, CA, Aug. 1999.

[13] Merit Inc. (2000) Ipma Statistics. [Online]. Available: http://nic.merit.edu/ipma

[14] T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM*, Sep. 1998.

[15] Memory-Memory (2000). [Online]. Available: http://www.memorymemory.com

[16] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proc. 17th ACM Symp. Operating Systems Principles*, Dec. 1999.

[17] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM*, Oct. 1997.

[18] C. Partridge, "Locality and route caches," in *Proc. NSF Workshop, Internet Statistics Measurement and Analysis*, Feb. 1999.

[19] D. Shah and P. Gupta, "Fast updates on ternary-cams for packet lookups and classification," in *Proc. Hot Interconnects VIII*, Stanford, CA, Aug. 2000.

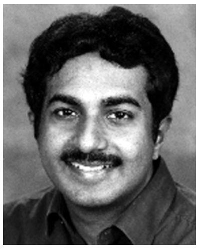
[20] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast scalable level four switching," in *Proc. ACM SIGCOMM*, Sep. 1998.

[21] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, Sep. 1999.

[22] J. Xu, M. Singhal, and J. Degroat, "A novel cache architecture to support layer-four packet classification at memory access speeds," in *Proc. IEEE INFOCOM*, Mar. 2000.

Florin Baboescu received the M.Sc. degree in computer engineering from the University Politehnica Bucharest and the Ph.D. degree in computer science from the University of California at San Diego.

He is an engineer in the Central R&D at STMicroelectronics Inc., where he works on hardware solutions for network search engines.



George Varghese (M'89) received the Ph.D. degree from the Massachusetts Institute of Technology, Cambridge, in 1992.

He is a Professor in the Computer Science Department, University of California at San Diego, where he does research in designing reliable protocols and efficient protocol implementations.

Dr. Varghese is a Fellow of the Association for Computing Machinery.