This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2007 proceedings.

Memory-Efficient 5D Packet Classification At 40 Gbps

Ioannis Papaefstathiou
ECE Department, Technical University of Crete,
Kounoupidiana, Chania, Crete, GR73100, Greece
ygp@ece.tuc.gr

Vassilis Papaefstathiou
Institute of Computer Science – FORTH,
Vassilika Vouton, Heraklio, Crete, GR71110, Greece
papaef@ics.forth.gr

***Abstract*—Packet classification is one of the most important enabling technologies for next generation network services. Even though many multi-dimensional classification algorithms have been proposed, most of them are precluded from commercial equipments due to their high memory requirements. In this paper, we present an efficient packet classification scheme, called Bloom Based Packet Classification (B2PC). B2PC comprises of an innovative 5-field search algorithm that decomposes multi-field classification rules into internal single field rules which are combined using multi-level Bloom filters. The design of B2PC is optimized for the common case based on analysis of real world classification databases. The hardware implementation of this scheme handles 4K rules by involving only 530KB of memory for its data structures, while it supports network streams at a rate of 15Gbps even in the worst case, and more than 40Gbps in the average case. This system covers 1.3 mm^2 in a 0.18μm CMOS technology. We show that given a certain memory budget and silicon cost, the B2PC is the most efficient hardware-based approach to the classification problem.**

***Index Terms*— Packet classification, QoS, Hardware Scheme**

I. INTRODUCTION

It is well established that multi-dimensional packet classification is a difficult problem [1], [8]. However, it is a necessity in order to support the next generation networking services incorporating certain Quality of Service (QoS) and security. Moreover, the ever growing speed of the interconnection technologies and the trend for low-cost networking equipments put additional pressure to the packet classification schemes. In particular, in order for such a scheme to be used in a real-world networking environment, it should support the current state-of-the-art networking speeds (i.e. OC-768 at 40Gpbs) while it should not be prohibitively expensive. Since packet classification is a very memory intensive task the latter is mainly translated to either use of inexpensive DRAM memories, or of small amounts of SRAMs; the use of TCAMs, although seems optimal from a performance perspective, is considered inadequate due to their very high cost and power consumption.

In general, packet classification requires searching a table of filters for the highest priority or the most specific filter that matches a certain incoming packet. Filters (or rules as they are frequently called) map a flow or a set of flows to a *FlowID*. Those filters consist of several fields and many different kinds of matches are supported (e.g. exact value, prefix and range matches, etc). Each filter or rule may also have an associated priority to allow more fine grained flow identification when a certain packet matches more than one rule.

Specifically, the packet classifiers currently employed in real systems are 5-dimensional and they use the following fields: (i) Source IP address (32-bits), (ii) Destination IP address (32-bits), (iii) Source Port (16-bits), (iv) Destination Port (16-bits) and (v) Protocol (8-bits). A filter in a classifier may specify any or all of those fields with prefixes, ranges, exact values or wildcards.

Given the fact that single field searching is a well studied problem and many efficient solutions have been proposed, decomposing a multiple field search problem into several instances of single field searches seems to be the most practical approach to the classification problem. However, this decomposition results in a number of complications. The primary challenge is to efficiently aggregate and combine the results of the single field searches. Moreover, the single field search engines should not only return the longest matching prefix for a given filter field, since the best matching multi-dimensional filter may contain a field which would not necessarily comprise of all the longest single-field matching prefixes. The majority of the techniques employing decomposition try to take advantage of certain filter set characteristics that allow them to limit the number of intermediate results. In general, the decomposition approach can provide very high throughput due to its parallel nature. However, this high lookup performance very often comes at the cost of memory overhead.

In this paper we propose a classification engine, called B2PC, which follows a similar approach by decomposing multi-field classification rules into internal single-field rules, which are then combined using multi-level Bloom filters [13]. B2PC is optimized for filter-sets with a few thousand rules and its data structures are handling very efficiently the common-cases identified in a large set of real-world classifiers. It uses the BOS single field searching scheme which has been proved to be very efficient [11]. The main advantages of B2PC are (i) a highly pipelined organization which results in processing rates of more than 40Gbps, in the average case, (ii) the innovative memory structures allowing it to support those rates with only 530KB of off-chip standard SRAM, (iii) the small hardware footprint, since its implementation covers only 1.3 mm^2 in a 0.18μm CMOS

0743-166X/07/$25.00 ©2007 IEEE

technology and (iv) the ability to very efficiently support incremental updates. In general, as the performance section demonstrates, given a certain memory budget, this scheme provides the highest throughput compared with all the systems that have been *implemented in hardware*.

II. Related Work

A complete review of the proposed approaches to the packet classification problem can be found in [2],[3],[8]. Each of the proposed schemes is very interesting from a certain perspective; many of them are optimized for software implementation taking advantage of certain features of the current CPUs, such as the Fat Inverted Segment tree (FIS-tree) [1] and the scheme in [12] which are taking advantage of the way the CPU caches work; others are tailored to very large filters sets (i.e. with more than 10^6 filters) and 2-dimensional searches (such as FIS-tree, Tuple Space Search [15], etc). The problem with all the software approaches is that they cannot support more than 1Gbps rates even when executed on the state-of-the-art network processors [9], [18].

In the sub-area of hardware-oriented approaches, Gupta and McKeown introduced *Recursive Flow Classification* (RFC) which provides high lookup rates at the cost of large amounts of memory [4]. The authors introduced a unique high-level view of the packet classification problem; essentially, packet classification can be viewed as the reduction of an *m*-bit string, defined by the packet fields, to a *k*-bit string specifying the set of matching filters for the packet or the action to be applied to the packet. For classification on the typical IPv4 5-tuple, *m* is 104 bits and *k* is typically in the order of 10 bits. The authors also performed a comprehensive study of real filter sets and extracted several useful properties. Specifically, they reported that the filter overlap and the associated number of distinct regions created in the multi-dimensional space is much smaller than the worst case of $O(n^d)$. For example for a filter set with 1734 filters, the number of distinct overlapping regions in a four-dimensional space was found to be 4316, as compared to the worst case which is approximately 10^{13}. The high performance of their presented scheme comes at the cost of large amounts of memory. Memory usage for less than 1000 filters ranged from a few hundred kilobytes to over one gigabyte depending on required performance. The authors propose a hardware architecture using two 64MB SDRAMs and two 4MB SRAMs that could perform 30 million lookups per second when operating at 125MHz. The index tables used for aggregation require significant pre-computation; which prohibits dynamic updates at high rates.

Lakshman and Stiliadis introduced another multiple field packet classification algorithm specifically designed for hardware implementation. Their technique is commonly referred to as the Lucent bit-vector scheme or Parallel Bit-Vectors (BV) [15]. The authors make the initial assumption that the filters are sorted according to priority. Parallel BV utilizes a geometric view of the filter set and maps filters into d-dimensional space. The authors implemented a five field version with five 128KB SRAMs. This configuration supports 512 filters and performs one million lookups per second. Baboescu and Varghese introduced the Aggregated Bit-Vector (ABV) algorithm which seeks to improve the performance of the Parallel BV technique by using statistical observations of real filter sets [7]. Simulations with real filter sets show that ABV reduced the number of memory accesses relative to Parallel BV by a factor of four. Simulations with synthetic filter sets show more dramatic reductions by a factor of 20 or more when the filters sets do not contain any wildcards. However, as wildcards increase, the reductions become much more modest. Moreover, no specific hardware-implementation for the ABV has been proposed or even sketched.

The main advantage of the tuple space search algorithm [16] is its very small memory requirements (O(N) where N is the number of rules). However, its search and update speed heavily depends on the number of active tuples and it is reported to be, in the worst case, forbiddingly high [10]. Moreover, this scheme supports up to 2-dimensional searches; it has not been simulated using large classification sets or 5-dimensional searches, and it is optimized for software implementation, since the hardware scheme proposed do not scale for large database sets (i.e. containing more than a few hundreds of filters).

HiCuts [4] and HyperCuts [5] partition the multi-dimensional search space based on certain heuristics. Each query leads to a leaf node in a search tree which stores a small number of rules that can be searched sequentially to find the best match. The characteristics of the decision tree (depth, degree of each node, and search criteria applied to each node) are configured during a preprocessing phase based on the performance and cost requirements. The main disadvantage of HiCuts is its high memory requirements (1MB of SRAM for only 1700 rules), while it needs 20 memory accesses to find a specific rule. HyperCuts reduces both the memory accesses needed and the memory requirements of HiCuts significantly. Unfortunately, no hardware implementation of this latter scheme is reported, and the one which is sketched needs a large number of independent memories (i.e. they mention that they need at least 10 SRAMs working in parallel).

The scheme with the smallest memory requirements, proposed so far, is the one by Sun et. Al [10]. The proposed algorithm has a memory ratio (i.e. the ratio of the total amount of memory used to that needed to just store the classification rules) of 2. However, the performance results demonstrated are based on artificial 2-tuple filter sets, while they mention that in order to be efficient, it terms of speed, they have to use the very expensive and power hungry TCAMs. Moreover, it is not clear how this scheme can scale to 5-tuple classification and what the silicon-cost of the proposed highly pipelined and parallel hardware architecture would be.

Our B2PC scheme focuses on today's 5-tuple filter-sets with a few thousand entries whereas special care has been taken so as to be efficiently implemented in hardware, and to demand moderate amounts of inexpensive (i.e. pure SRAM) memory. Moreover, the proposed device is capable of supporting the state-of-the-art network rates of 40Gbps and beyond, while its silicon cost is very low.

III. B2PC DESIGN

The design of B2PC is driven by the observations of Gupta and McKeown [8], described in the last section, as well as our analysis of the real-world filter sets of [14]. The key issues affecting our design decisions are mainly the following:

1) Current filter sets' sizes are small, ranging from tens of filters to less than 5000. However, it is not clear if the size limitation is due to the networking applications or it has been imposed by the limited performance of current classification solutions.
2) The protocol field is restricted to a small set of values; TCP, UDP and commonly used wildcards (covering more than 95% of the cases).
3) Filters specify a limited number of unique transport Port ranges. The specifications for port ranges vary and have definitions like 'greater than 1023' or '20 to 23'.
4) The number of unique address-prefix rules matching a given source or destination address is usually five or less.
5) The number of single field filters matching a given packet is typically five or less.
6) Different multi-dimensional rules very often share a number of single-field values.
7) The number of single field values is significantly less than the number of overall filters.

A. Single Field Operations

Given that B2PC follows the decomposition approach, it is essential to employ a very efficient single-field scheme supporting both exact and prefix matches at very high speeds, while utilizing small amounts of memory. Those requirements are fulfilled by the BOS scheme described in [11]. Since our single-field lookup mechanism should not only report the longest prefix match but, instead, all the prefixes that match, we have altered the BOS scheme so as to provide us with All-Prefix-Matches (APM) and for each match the associated match length, as described in [17]. Moreover, and since BOS supports prefix matches, a certain mechanism transforming the range-based Source and Destination Port rules into prefix-rules has been employed utilizing the algorithm of [1]. Additionally, the BOS engine that supports those Port Fields has been fine-tuned since the original BOS supports 32-bit values while in the port fields we have 16-bit values. For the Protocol field, in order to perform the necessary 8-bit searches, we use a 256-entry directly indexed table (PRO_TBL).

Based on the observations described in the last subsection the proposed scheme supports up to 4K 5-tuple rules, therefore, each filter can be identified by a 12-bit *FlowID*. A general overview of the B2PC scheme is presented in Fig. 1 where all the discrete components are shown.

B. Internally Represented Filters

In order to reduce the memory requirements we take advantage of the fact that many rules share the same field values. In order to cope with this value-sharing issue we decided to have a special internal representation of the various filters where each particular *field* (sub-rule) is assigned an internal ID during rule insertion. The internal ID of each field is the originally given *Flow ID* value of the whole rule. If two or more rules share the same field-value their internal ID is equal to the first inserted Flow ID. Table II illustrates how the rules presented in Table I are kept internally in B2PC. This information is kept in the 4K entry RULES_TBL which is directly indexed by the 12-bit flow ID.

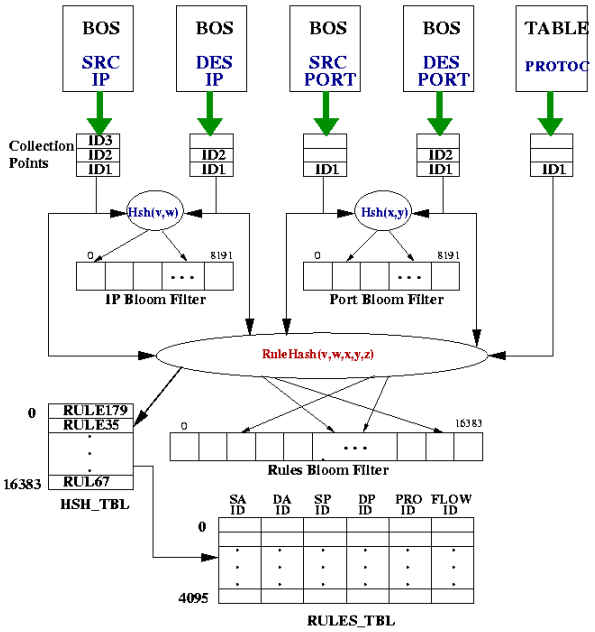

Fig. 1. Overall Architecture of B2PC

A side-effect of this ID sharing scheme is that a certain internal-ID cannot be deleted unless all the rules employing it are deleted. In order to cope with this problem, we keep a reference count for each internal ID on each field. The maximum number of distinct internal IDs is obviously equal to the maximum number of supported Flow IDs (i.e. 4K); each internal ID may be referenced from at most 4K rules and therefore we need 4K 12-bit counters for each field. Therefore, in total we need 5 x 4096 12-bit counters in order to efficiently support incremental updates. Obviously, when a newly inserted rule references an internal ID, we increment the appropriate counter and when such a rule is deleted we decrement the counter. The original single field value is only deleted when the corresponding counter reaches zero.

C. Combining Results

Given the 5 fields of a packet, B2PC has to find which of the existing rules best matches all of them. In the first stage, the five single-field engines provide a number of matching prefixes and the associated IDs. The IP address fields, namely Source IP and Destination IP, are prefix-based and may provide at most 33 matches each; 32 possible matches for the 32 possible prefix lengths and 1 for the zero-length wildcard. Similarly, the port fields may provide at most 17 matches. In the protocol field we have only exact-value and "match-all" searches so this sub-engine returns either a match on the value itself or the wildcard; therefore we have at most 2 matches.

TABLE I
EXAMPLE FILTER SET

No	Src IP	Dest IP	Src Port	Dest Port	Protocol	Flow ID
1	139.91.70.*	147.52.16.*	*	*	TCP	10
2	139.91.*.*	147.102.*.*	*	21	TCP	14
3	139.91.*.*	147.27.*.*	< 1024	*	*	17
4	*.*.*.*	139.91.*.*	*	80	UDP	26
5	139.91.70.33	147.52.16.33	135	< 1024	TCP	31
6	139.91.70.36	147.27.*.*	< 1024	21	*	45
7	*.*.*.*	147.52.*.*	*	23	*	47
8	139.91.*.*	147.52.*.*	135	135	TCP	50
9	139.*.*.*	147.*.*.*	*	80	TCP	54
10	139.91.*.*	147.52.*.*	*	135	TCP	55

TABLE II
INTERNAL REPRESENTATION OF THE EXAMPLE FILTER SET

No	Src IP	Dest IP	Src Port	Dest Port	Protocol	Flow ID
1	10	10	10	10	10	10
2	14	14	10	14	10	14
3	14	17	17	10	17	17
4	26	26	10	26	26	26
5	31	31	31	31	10	31
6	45	17	17	14	17	45
7	26	47	10	47	17	47
8	14	47	31	50	10	50
9	54	54	10	26	10	54
10	14	47	10	50	10	55

TABLE III
INCOMING PACKET HEADER FIELDS

Src IP	Dest IP	Src Port	Dest Port	Protocol
139.91.62.39	147.52.17.25	5000	80	TCP

MATCHED RESULTS IN COLLECTION POINTS

14	47	10	26	10
54	54	-	31	17
26	-	-	10	-

TABLE IV
TOTAL POSSIBLE PERMUTATIONS

No	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID
1	14	47	10	26	10
2	14	47	10	26	17
3	14	47	10	31	10
.	.	.	.	.	.
.	.	.	.	.	.
18	54	47	10	10	17
19	***54***	***54***	***10***	***26***	***10***
20	54	54	10	26	17
.	.	.	.	.	.
.	.	.	.	.	.
35	26	54	10	10	10
36	26	54	10	10	17

The internal IDs and the matching lengths returned by each of the five single field engines are gathered in certain collection points, one for every field, and they are then forwarded to the engine that combines all those results. The collection points are taking the matched prefixes from all the single-field modules and keep them in decreasing length order. Each collection point gives the longest prefix match first and proceeds with the less specific matches.

The results from every single-field engine should be combined, so as to cover all the possible permutations, and then it should be determined which of these permutations are actually valid (i.e. whether such a multi-field rule exists). Although the possible number of permutations could be large, in real-world databases, as it was described in Section III, the maximum number of matches in each field is typically less than five and the rules that match a certain incoming packet are usually less than five, as well. In the vast majority of the existing network applications, the best matching rule is the rule that has the most specific value. In order to address this issue, we first check whether the combination of the internal IDs that come from the longest single field matches, as the collection points provide them, is indeed valid; then we continue on checking the less specific matches. This searching order has increased the performance of our scheme, when compared to the random order approach, by up to 22%! Moreover B2PC assigns priorities to the fields that are taken into account when the permutations are generated. In particular, the permutations are generated by keeping the current matched value of the most significant field, at each time, and producing the combinations of the values coming from the less significant fields. Based on analysis of real-world networking applications the significance of fields in decreasing order is: Source IP, Destination IP, Source Port, Destination Port and Protocol.

Note that when all the collection points provide the same internal ID, then we surely know that this permutation belongs or used to belong to our set; the same value for all the internal IDs, in a permutation, denotes that the values in all fields are the initially inserted ones for this specific rule. The only thing we have to investigate in this case is whether this rule has been deleted and the values found have only been kept in the database due to references from other rules.

To illustrate how these permutations are generated, we show in Table III the header fields of an incoming packet and the matched results in the collection points. This example assumes the rules of Table I, while the matched results are stored in order, from the most specific to the less specific.

The total number of possible permutations is equal to the overall product of the number of matches in every field:

$Total_{perm}$ = #Src IP IDs * #Dest IP IDs * #Src Port IDs * #Dest Prt IDs * #Proto IDs.

Hence for the matches shown in Table III the total number of permutations is: $Total_{perm} = 3 * 2 * 1 * 3 * 2 = 36$

These 36 generated permutations are shown in Table IV and the permutation that corresponds to an existing ruleset entry is shown in bold.

D. Set Membership Queries with Bloom Filters

One of the most important challenges of B2PC, if not *the* most important, is how to identify that a permutation belongs to the given set of rules. Sequential access to the rule table is prohibitively slow since we may need to access every single entry of it. Therefore, a data structure that can efficiently represent a given ruleset and support quick set membership queries is needed. Hash tables and B-Trees are widely used for this type of queries but there are also the Bloom Filters that have received renewed attention in network applications [13]. The main advantage of those filters, when compared to the

other data structures, is that they can easily be implemented in hardware while supporting set-membership queries at extremely high rates. The disadvantage of Bloom filters is that they may report that a certain item is part of the set, when this is not the case (i.e. false-positive error).

In order to efficiently support classification databases with up to 4K rules, B2PC employs suitable Bloom Filters. A very important characteristic of the Bloom Filter is that its false positive rate can be tuned, as discussed in [13]. In order to keep this rate low, we have carefully chosen the size of the Bloom filter bit-vector and then calculated the corresponding optimal number of hash functions that set the filter's individual bits. Based on an analysis presented in detail in [17] we ended-up with a bit vector which is 2^{14} bits wide; based on the latter analysis, the optimal number of hashing functions that set this vector is 4. Given those parameters, the produced Bloom Filter has a theoretical false positive probability of 6.2%.

The bit-vector of the Bloom filter is relatively large to be kept in registers/flip-flops, and therefore it is stored in a memory array. Having four hash functions means that we have to set four bit positions in the bit vector and test four bits at each access; due to the fact that the bit-vector is to be stored in a memory array we may require up to four memory accesses to locate each bit. Thus, in order to avoid sequential accesses, and since the array is quite small and can easily be kept on-chip, we split this bit-vector into four equal sub-vectors of 4K bits each and assign each hash function to one of those sub-vectors. This allows us to implement the accesses in parallel and decide in a single parallel memory access if the current permutation belongs to our set. Additionally, this splitting prevents the hash functions from setting the same bit.

Since certain bits of the Bloom filter may be shared by many rules in the ruleset, we cannot delete a bit if other rules depend on this. Therefore, and in order to efficiently support incremental updates, we keep counters for every bit of the Bloom filter. Hence, for the 16K bit-vector of our Bloom filter we need 16K counters. Each counter is at most 12-bits since this is the maximum number of rules supported. Accordingly, a bit from the vector is deleted only when the corresponding counter reaches zero. Since its sub-vectors comprise of 4K entries, the hash function produces a 12-bit value. Moreover, based on our analysis of real filter sets, these hash functions should use all of the ID information so as to provide discrete values for each permutation. Inherently, the IDs we use are the actual Source IP (SIP), Destination IP (DIP), Source Port (SPO), Destination Port (DPO) and Protocol (PRO) IDs.

After careful analysis of the classification databases and the Bloom Filter properties, we have defined the hash functions by the use of XOR, SHIFT (>>,<<) and the reverse (REV) function according to the following formulas:

BLH1 = (SIP>>4) **xor** REV(DIP>>2) **xor** (SPO<<4) **xor** (DPO>>3) **xor** (PRO<<3)
BLH2 = SIP **xor** (DIP<<6) **xor** (SPO>>2) **xor** REV(DPO) **xor** PRO
BLH3 = (SIP<<3) **xor** REV(DIP) **xor** REV(SPO) **xor** DPO **xor** (PRO<<6)
BLH4 = REV(SIP) **xor** (DIP<<3) **xor** (SPO>>3) **xor** (DPO<<1) **xor** (PRO>>2)

The performance of these hash functions is studied and analyzed in the performance section of this paper.

E. Flow ID Resolving

Once we have a match in a set-membership query we should first determine whether it is a false positive match and in case it is not, we have to return the corresponding FlowID. To locate the FlowID we use a hash table of 16K entries (HSH_TBL) that gives us the matched FlowID. Once we have the FlowID we access the RULES_TBL, as described in the last subsection, and compare the stored IDs with the IDs of the current permutation. In case all IDs match, we have found the final result, otherwise this match is a false positive and we continue by testing the rest of the permutations.

Indexing the HSH_TBL requires a hash function and obviously this function may produce collisions. Resolving these collisions is trivial by using variable size blocks (such as in [11]) that hold the colliding FlowIDs. If more than one FlowIDs are stored in a specific HSH_TBL entry then we have to sequentially check all of them. The hash function proved to produce the optimal, for our case, results uses the already hashed values of BLH1, BLH2, BLH3 and BLH4 to indicate an entry in HSH_TBL. Its 14-bit value is defined as follows:

HSH_TBL_{index} = (BLH1,00) **xor** (00,BLH2>>4) **xor** (00,BLH3) **xor** (00,REV(BLH4))

The performance of this hash function is also studied and analyzed in the performance section.

F. Improving the Efficiency of Set Membership Queries

Following our simple approach, we have to check every generated permutation for actual membership despite the fact that a certain pair of source-destination addresses or a pair of source-destination ports may not be part of the ruleset. To avoid these useless queries we have used two additional Bloom Filters that contain the information regarding the IP-Address pairs and the Port-pairs, respectively. This approach splits the membership queries problem into two sub-problems. This splitting is based on McKeown's observations [8] which state "that the IP address pairs characterize the actual network paths and the Port pairs characterize the network applications". So in order to speed up the processing time, in the common case, the additional Bloom filters are checked first, and if they both provide a match then we query the "Main" Bloom filter (i.e. the one that holds the actual full rules). After an analysis of the databases, we claim that for each of the two Bloom Filters, the optimal approach is to use an 8K entry bit-vector with two hash functions. We again split each bit-vector into two equal sub-vectors and store them in separate on-chip tables so as to exploit parallelism. Moreover, accessing the Bloom filters of the IP-pair and Port-pair can be done in parallel and simultaneously with the accesses to the Main Bloom Filter.

Based on our analysis, we have defined the hash functions for the IP and Port pairs by the use of XOR, SHIFT and reverse (REV) function according to the following formulas:

IP_BLH1 = { SIP(6:11) **xor** DIP(0:5) , SIP(0:5) **xor** DIP(6:11) }
IP_BLH2 = { SIP(0:5) **xor** DIP(6:11) , SIP(6:11) **xor** DIP(0:5) }
PR_BLH1 = SPO **xor** (DPO<<2)
PR_BLH2 = (SPO<<2) **xor** REV(DPO)

The performance of these hash functions is also studied and analyzed in the next section.

The number of the generated permutations for the IP-pairs and the Port-pairs is obviously significantly smaller, compared to the total number of 5-tuple permutations, and thus they can be checked for actual membership much faster. When both IP and Port queries are successful, the matched pairs along with the 2 possible Protocol matches are processed using the information contained in the Main Bloom filter. Using the results of Table III we illustrate, in Table V, which queries are performed in parallel in the three Bloom filters. The queries in both IP and Port Bloom Filters are started simultaneously. When at least one of the Bloom Filters returns a match (while the other may still process the incoming data) a query to the Main Bloom Filter is performed.

In general, breaking the problem into two stages allows us to better handle the required membership tests. Looking at the actual reasoning behind the searching order the IP-pair enquiry first determines whether a certain network path exists in the ruleset, while the Port-pair enquiry checks for certain network configurations; the final rule membership query clarifies whether those pairs match together in a rule. Searching these pairs independently distributes the queries efficiently and provides faster results as the next section clearly demonstrates.

IV. Simulation Results

In order to measure the efficiency of our scheme we employed realistic filter sets and test patterns. In particular we have used Taylor's ClassBench [14] which is a suite of tools for performance evaluation of classification algorithms and is publicly available. ClassBench contains a filter set generator that uses seeds from ***real-world*** filter sets in order to provide synthetic databases which model real filters in an *extremely accurate* manner. Moreover, it includes a packet header generator that produces a sequence of packet headers to exercise a given filter set; this generator uses the Pareto Distribution which is the best available statistical model for Internet traffic. One of the strong points of our work, when compared with the related work of Section II, comes from the fact that we are among the first to use such traces, which model the real-world classification environment much more accurately than the artificial filter-sets based on routing tables that have been used in the past.

The efficiency of B2PC was measured using 8 filter sets of various sizes. Before we use those filters we analyzed their properties so as to be sure that they are compliant with the features described in Section II. Then, we estimated the efficiency of our approach based on the average and worst-case number of memory accesses needed for classifying a network packet, as well as its memory requirements.

In order to accurately model the real-world environment we generate filter sets that represent the most common classification applications, namely Access Control List (ACL), Firewall (FW) and IP Chain (IPC). We used the real-filter's seeds and generate 8 such sets; for each one of them all the features of Section III hold.

A. Hashing Functions and False Positives

As it was described in the last section we incorporate many hash functions in B2PC in order to either index specific bits of the Bloom filters or to identify the final FlowID. Looking at the Bloom Filters' functions the most important property is to produce several distinct values and minimize the number of references to each filter-bit. After a thorough analysis of different such functions we ended up with those described in subsections III.D and III.F; Table VI shows the number of bits set by them, in each of the Bloom filters, as well as the number of rules that reference these bits. Those results demonstrate that our hashing functions behave efficiently, since they set a large number of distinct bits and the number of references per bit is certainly not high. More specifically, in the Port Bloom filter, the higher number of references comes from the fact that we have a small number of common values as we have described in the last subsection. The Main Bloom Filter has many bits set with a small average number of references to each bit, due to the scheme we are using for creation of the internal IDs which produces many distinct values. Moreover, the average number of references in the IP Bloom filter is a little higher than in the Main Bloom filter as an effect of the small number of unique field values in the latter compared to the size of the set (i.e. feature III.7); in other words many rules in a set share the same Source and

TABLE V
PARALLEL BLOOM FILTER QUERIES

Query Number	IP Pair Permutation		Port Pair Permutation		Rule Permutation				
	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID
1	14	47	10	26	-	-	-	-	-
2	**14**	**47**	10	31	14	47	10	26	10
3	**14**	**47**	10	10	14	47	10	26	17
4	14	54	10	26	14	47	10	10	10
5	54	47	**10**	**26**	14	47	10	10	17
6	54	54	**10**	**26**	-	-	-	-	-
7	**54**	**54**	10	31	***54***	***54***	***10***	***26***	***10***
8	**54**	**54**	10	10	54	54	10	26	17
9	26	47	10	26	54	54	10	10	10
10	**26**	**47**	**10**	**26**	54	54	10	10	17
11	**26**	**47**	10	31	26	47	10	26	10
12	**26**	**47**	10	10	26	47	10	26	17
13	26	54	10	26	26	47	10	10	10
14	-	-	-	-	26	47	10	10	17

Destination IP address. It should be noted that the choice of the Hash functions is indeed *crucial*, since different such functions gave us maximum and average reference numbers which were up to 3 times higher than the presented ones.

Another important characteristic of the Bloom Filters' hash functions is the number of false positives they trigger; the bit-vector size of the filters influences this same metric. Using the same filter sets and the corresponding packet headers we measured the false positive rates shown Table VII.

The observed false positive rate in B2PC is close to the theoretical 6.2% value for 4K active rules, while it is very low for small filter sets. The high rate of false positives in IP and Main Bloom filters for ACL1 and ACL2 filter sets can be justified by the fact that our hashing functions have produced higher maximum and average reference counts as shown in Table VI. More specifically, these filter-sets have an increased number of matched values per field and thus they produce more permutations for which the Bloom filters are queried. On the other hand ACL3, which is the largest database, has a very low rate of false positives despite the fact that we have observed the highest maximum and average reference counts. This is due to the fact that it incorporates a small number of matches in each of the single fields and therefore fewer permutations are generated and forwarded to the Bloom filters.

The B2PC also uses a hash function in order to identify the final FlowID of the matching permutation as described in subsection III.E. We illustrate the collisions produced by this hash function in Table VIII. As those results demonstrate, the function we chose seems ideal for such a classification framework and produces 34% less collisions in the average case than the worst hashing function we have studied.

B. Storage Requirements

As it was analytically described in the previous sections, one of our main concerns, when designing the proposed framework, was to be very memory efficient. In this subsection we present the storage requirements of B2PC for all the generated filter sets. To calculate the total storage for B2PC we measure the storage requirements of the B2PC tables and the storage of all the employed BOS engines.

As described in [11] each BOS engine employs a number of static tables and a group of memory blocks implementing its dynamic memory management (DMM) scheme. Since each BOS sub-system supports very few unique values the requirements of the DMM ranges from 2 to 5 KB for each of the filter sets. Moreover, every BOS engine requires 73KB for its static tables; so the total memory requirements of each BOS engine are at most 78KB.

For the total storage requirements of B2PC we have to calculate the size of the Bloom filters (including their associated counters), the counters for the internal IDs of each BOS engine, the protocol table (PRO_TBL), the hash table (HSH_TBL) and the rules table (RULES_TBL). For our calculations we assume standard 36-bit wide memory words. We also assume that two 12-bit counters, together with the associated information, are placed in a single 36-bit word and each rule entry needs 2 memory words. Accordingly, the storage requirements for all the B2PC components needed are shown in Table IX. As this table demonstrates all our filter sets can be supported given just 538KB of memory.

TABLE VI
NUMBER OF REFERENCES IN BLOOM FILTERS

Set Name	IP Bloom Filter (8192 bits)			Port Bloom Filter (8192 bits)			Main Bloom Filter (16384 bits)		
	# set bits	Max Refs	Avg refs	# set bits	Max Refs	Avg refs	# set bits	Max Refs	Avg refs
ACL1	2651	29	1,77	305	321	15,39	6566	16	1,43
ACL2	2912	32	2,04	396	336	15,02	7847	10	1,51
ACL3	2985	40	2,23	78	708	85,71	8468	9	1,57
FW1	418	8	1,34	107	47	5,27	1023	4	1,10
FW2	355	13	1,48	219	29	2,40	958	4	1,09
FW3	228	7	1,36	78	29	4,00	568	3	1,09
IPC1	2406	10	1,40	164	650	20,5	5251	5	1,28
IPC2	202	8	1,67	18	111	18,77	503	7	1,34

TABLE VII
OBSERVED FALSE POSITIVES RATE

Set Name	IP Bloom False Positives (%)	Port Bloom False Positives (%)	Main Bloom False Positives (%)
ACL1	3,2	0	5,1
ACL2	8,4	0	8,3
ACL3	0,005	0	0,01
FW1	3,7	0	0
FW2	1,5	0	0
FW3	2,0	0,7	0,2
IPC1	0,3	0	0,5
IPC2	0,1	0	0

TABLE VIII
HASH TABLE COLLISIONS

Set Name	Set Size	Max Collisions	Average Collisions
ACL1	2348	3	1,17
ACL2	2974	3	1,19
ACL3	3343	3	1,21
FW1	282	2	1,02
FW2	263	2	1,07
FW3	156	1	1
IPC1	1687	2	1,10
IPC2	169	3	1,29

TABLE IX
B2PC COMPONENTS MEMORY REQUIREMENTS

Component	Memory Words	Total (Kbytes)
BOS ID counters	10240	45
Bloom Filters counters	16384	72
HSH_TBL	16384	72
RULES_TBL	8192	36
PRO_TBL	256	1
Total (not including BOS)	**51456**	**226**
BOS Engines (x4)	69812	312
Total	**121268**	**538**

C. Memory Accesses

As described in the last section, the lookup performance of B2PC mainly depends on the lookup time of each BOS engine and on the set-membership queries in the Bloom filters.

The BOS scheme was introduced and analyzed in [11] whereas more details regarding its performance, when APM results are required, can be found in [17]. The BOS scheme can work either utilizing only one external SRAM device, in which case it performs its operations in a serial manner, or in

parallel mode where four SRAM devices are needed. Based on our simulations with the filter sets and the corresponding packets headers, the average number of memory accesses needed to perform a complete search in a parallel manner is 2,2 while the worst case observed is 6 memory accesses; when BOS operates in sequential mode we need 9,2 accesses on average and 25 in the worst case.

The second factor which influences the performance of B2PC is the number of sequential probes in its Bloom filters. As described in Section III we query the IP and Port pair Bloom filters in parallel and then probe the Main Bloom filter for the matched IP and Port pairs; the results of our simulations are shown in Table X. In addition to the Bloom Filter accesses, we have to take into account the accesses in HSH_TBL (that are equal to the collisions presented in Table VIII) and two memory accesses to acquire the final rule from RULES_TBL. The total numbers of average and maximum memory accesses triggered by the various subsystems of B2PC (excluding the BOS engines) are presented in Table X.

Finally, in order to get the overall memory accesses needed by the complete B2PC scheme we sum up the accesses of the BOS engines with those generated by the other B2PC subsystems. It should be noted that all four BOS engines can work in parallel; each of them may be configured in parallel or in sequential mode depending on the actual cost limitations (the parallel BOS mode needs four times the SRAM devices required by the sequential mode). In Table XI we present the total average number of memory accesses needed so as to process a complete packet. The worst-case numbers can be calculated if we add to the numbers of Table X the worst case number of accesses needed by the BOS engines which is 6 in the case of the parallel mode and 25 for the sequential mode.

TABLE X
NUMBER OF MEMORY ACCESSES IN B2PC DATA STRUCTURES (NOT INCLUDING BOS)

Set Name	Bloom Filters Accesses MAX	Bloom Filters Accesses AVG	Hash Table Accesses MAX	Hash Table Accesses AVG	Total Accesses MAX	Total Accesses AVG
ACL1	17	2,68	3	1,17	22	5,85
ACL2	29	4,03	3	1,19	34	7,22
ACL3	6	2,01	3	1,21	11	5,22
FW1	22	4,74	2	1,02	26	7,76
FW2	18	3,18	2	1,07	22	6,25
FW3	34	5,34	1	1	37	8,34
IPC1	16	2,16	2	1,10	20	5,26
IPC2	4	2,07	3	1,29	9	5,36

TABLE XI
TOTAL NUMBER OF AVERAGE MEMORY ACCESSES IN B2PC

Set Name	BOS Parallel	BOS Sequential	B2PC Accesses	B2PC with Seq. BOS	B2PC with Par. BOS
ACL1	2,20	9,20	5,85	15,05	8,05
ACL2	2,20	9,20	7,22	16,42	9,42
ACL3	2,20	9,20	5,22	14,42	7,42
FW1	2,20	9,20	7,76	16,96	9,96
FW2	2,20	9,20	6,25	15,45	8,45
FW3	2,20	9,20	8,34	17,54	10,54
IPC1	2,20	9,20	5,26	14,46	7,46
IPC2	2,20	9,20	5,36	14,56	7,56

TABLE XII
B2PC SILICON COST

Components	Area (mm^2)	Equivalent NAND Gates
Combinatorial	0,595	115K
Non-Combinatorial	0,250	48K
Memories	0,456	88K
Total	**1.301**	**251K**

V. HARDWARE DEVICE'S COST & PERFORMANCE

The proposed scheme has been Synthesized and Placed and Routed for a 0.18μm CMOS technology and it works at 400MHz, covering the area shown in Table XII. Using either one (in case of sequential accesses) or four (for parallel accesses) 400MHz inexpensive external SRAMs we have measured the performance of our hardware system.

Table XIII presents the network performance of B2PC counted both in Millions of Packets Per Second (Mpps) and in Gigabit Per Second (Gbps) assuming the device processes only minimum-size IP-Packets (40 bytes). Obviously, in case our classifier is employed in a real-world environment it will process IP packets with a mean size much greater than 100 bytes, as reported in [14] , and therefore it would easily be able to support network rates of 40 Gbps or higher.

In order to demonstrate the exploitation space of the proposed framework we have listed, in Table XIV, the performance of B2PC together with that of other similar classification schemes, in terms of supported rules, storage requirements, and traffic rate serviced. Our comparison contains the most efficient such schemes, that do not use TCAMs and have been implemented in hardware. The scheme in [10] as well as HiCuts and HyperCuts, although they have been designed so as to have very low memory requirements, they could not be included in this comparison for a number of reasons : (a) there is no specific hardware implementation proposed for any of them, (b) for [10] the efficient implementation sketched by the authors use very expensive and power hungry TCAMs, (c) especially HyperCuts which is more efficient that HiCuts, seems to be a very innovative approach but unfortunately no assumptions can be made regarding the performance of their sketched hardware implementation in terms of operation frequency and actual memory bandwidth needed, while the fact that they need at least 10 SRAMs working in parallel, is making it more expensive than B2PC.

Furthermore, in order to be able to better demonstrate the efficiency of the various classification schemes when the memory cost is also taken into account, we introduce the metric of Mpps per Mbyte. This metric has been calculated for all the schemes of Table XIV by considering that they all work at 400MHz and linearly extrapolating their throughput, which is an optimal (for all but our scheme) assumption. It should be noted that the ABV scheme has not been implemented in hardware and therefore some elements are missing in Table XIV. However, we ended up with the demonstrated Mpps/Mbyte number based on a number of assumptions: (a) since ABV is similar, yet more complicated than BV, we can assume that it would work at the same speed as the latter (a rather optimistic approach), (b) given the fact that ABV is about an order of magnitude faster than BV in the majority of

TABLE XIII
WORST CASE NETWORK PERFORMANCE OF B2PC

Set Name	Mpps		Gbps	
	B2PC with Seq. BOS	B2PC with Par. BOS	B2PC with Seq. BOS	B2PC with Par. BOS
ACL1	26,57	49,68	8,50	15,90
ACL2	24,36	42,46	7,80	13,59
ACL3	27,73	53,90	8,88	17,25
FW1	23,58	40,16	7,55	12,85
FW2	25,88	47,33	8,28	15,15
FW3	22,80	37,95	7,30	12,14
IPC1	27,66	53,61	8,85	17,16
IPC2	27,47	52,91	8,79	16,93

TABLE XIV
SUMMARY OF CLASSIFICATION SCHEMES

Scheme	Freq. (MHz)	# of Rules	Storage (# of mems)	Throughput (Mpps)	**Efficiency (Mpps / Mbyte)**
BV [15]	33	512	640KB (5)	1	18,9
RFC [4]	125	1700	976 KB (2) + 15,6 MB (2)	30	5,8
ABV [7]	-	700	35KB (-)	3,7	105,7
B2PC	**400**	**3300**	**540 KB (4)**	**42,5**	**78,7**

the cases, the efficiency metric can be calculated by linear extrapolation.

Table XIV clearly shows that, despite the fact that RFC has the best throughput, its performance is based on greedy memory consumption and supports at most 1700 rules. On the other hand ABV does provide the highest number of Mpps/Mbytes, which is about 35% higher than that of B2PC, but it supports such efficiency with 80% less rules than B2PC. Therefore, we claim that our scheme provides the optimal bandwidth-to-memory approach, for any device that supports a few thousand rules. Obviously, if performance is the only issue RFC would be more appropriate, or for embedded, low-memory, devices scheme of [10] would probably be preferable. Moreover, for devices supporting relatively small filter sets ABV seems the natural case.

In general, the efficiency of the proposed scheme comes from the fact that it takes advantage of all the specific features of the current real-world filter databases, while it has been designed from the beginning for efficient hardware implementation. The algorithm proposed may be less efficient from the other algorithms found in the bibliography, in worst-case scenarios, but the hardware implementation of this scheme is the most efficient one demonstrated so far when memory requirements, number of rules and bandwidth are all taken into account. Moreover, its hardware cost (in terms of silicon covered) is minimal making it an even more promising approach for low cost classification engines.

VI. CONCLUSIONS

This paper presents a 5-dimensional classification scheme optimized for state-of-the-art networking applications/services which incorporate up to 4K classification rules. The proposed mechanism decomposes multi-field classification rules into internal single-field rules, which are then combined using multi-level Bloom filters. Its main advantages come from the fact that it employs only 530KB of inexpensive external SRAMs, while it can support network rates higher than 40Gbps. This scheme has been designed so as to be efficiently implemented in hardware and it covers 1.3mm^2 of silicon in a 0.18μm CMOS technology. As our performance results demonstrate, given a certain memory budget, number of supported rules and silicon cost, the B2PC provides the highest performance when compared to all the similar systems implemented in hardware.

REFERENCES

[1] A. Feldmann and S. Muthukrishnan, *"Tradeoffs for Packet Classification"*, in IEEE Infocom'00, March 2000.

[2] J. van Lunteren and T. Engbersen, *"Fast and scalable packet classification"*, IEEE Journal on Selected Areas in Communications, vol. 21, pp. 560–571, May 2003.

[3] T. Y. C. Woo, "*A Modular Approach to Packet Classification: Algorithms and Results*", in Infocom'00, March 2000.

[4] P. Gupta and N. McKeown, *"Packet Classification using Hierarchical Intelligent Cuttings"*, in *IEEE Micro*, vol. 20:1, Jan/Feb 2000, pp 34-41

[5] S. Singh, F. Baboescu, G. Varghese, and J. Wang, *"Packet Classification Using Multidimensional Cutting"*, in ACM SIGCOMM'03, August 2003.

[6] V. Srinivasan, S. Suri, G. Varghese, and M.Waldvogel, *"Fast and Scalable Layer Four Switching"*, in ACM SIGCOMM'98, June 1998.

[7] F. Baboescu and G. Varghese, *"Scalable Packet Classification"*, in ACM SIGCOMM'01, August 2001.

[8] P. Gupta and N. McKeown, *"Algorithms for Packet Classification"*, in *IEEE Network Special Issue*, March/April 2001, v15, n2, pp 24-32.

[9] Deepa Srinivasan, Wu-chang Feng *"Performance Analysis of Multi-Dimensional Packet Classification on Programmable Network Processors"*,in IEEE LCN 2004, November 2004.

[10] Xuehong Sun, S.K. Sahni, Y.Q. Zhao, "*Packet classification consuming small amount of memory"*, in IEEE/ACM Transactions on Networking, Volume: 13, Issue: 5, pp 1135- 1145, Oct. 2005

[11] I. Papaefstathiou, V. Papaefstathiou, *"An innovative low-cost Classification Scheme for combined multi-Gigabit IP and Ethernet Networks",* in IEEE ICC'06, June 2006

[12] F. Chang, K. Li, Wu-chang Feng, *"Approximate Caches for Packet Classification",* in IEEE Infocom'04, March 2004

[13] S. Dharmapurikar, P. Krishnamurthy, D.E. Taylor, *"Longest Prefix Matching Using Bloom Filters"*, in ACM SIGCOMM'03, August 2003

[14] David Taylor and Jonathan Turner, *"ClassBench: A Packet Classification Benchmark"*, in IEEE Infocom'05, March 2005.

[15] T. V. Lakshman and D. Stiliadis, *"High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching"*, in ACM SIGCOMM'98, September 1998.

[16] V. Srinivasan, S. Suri and G. Varghese, *"Packet Classification Using Tuple Space Search,"* in ACM SIGCOM'99, September 1999.

[17] V. Papaefstathiou, "*Design and Implementation of Network Packet Classification Engines",* MSc Thesis, Computer Science Department, University of Crete, 2005, Heraklion, Crete, Greece.

[18] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar and A. Campbell. *"Directions in Packet Classification for Network Processors"*. 9th HPCA, February 2003.