A Multi Gigabit FPGA-based 5-tuple classification system

Antonis Nikitakis
Technical University of Crete,
Department of Electronic and Computer Engineering
Kounoupidiana, Chania, Crete, GR73100, Greece
anikita@mhl.tuc.gr

Ioannis Papaefstathiou
Technical University of Crete,
Department of Electronic and Computer Engineering
Kounoupidiana, Chania, Crete, GR73100, Greece
ygg@mhl.tuc.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 Dual Stage Bloom Filter Classification Engine (2sBFCE). 2sBFCE comprises of an innovative 5-field search scheme that decomposes multi-field classification rules into internal single-field rules which are combined using multi-level Bloom filters. The design of 2sBFCE is optimized for the common case based on analysis of real world classification databases. The hardware implementation of this scheme handles 4K rules while supporting network streams at a rate of 2Gbps even in the worst case, and more than 6Gbps in the average case when implemented in an off-the-shelf FPGA.

Keywords: Packet classification, QoS, Hardware Scheme

I. INTRODUCTION

It is well known 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. 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 TCMs, 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 the packet. Filters (or rules as they are frequently called) map a flow or a set of flows to a specific action. Each 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 (either 2 or 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 triggers 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.

In this paper we propose a classification engine, called 2sBFCE, 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]. The 2sBFCE 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. The 2sBFCE main advantages are: (i) pipelined organization which results in high processing rates (up to 4Gbps) using low cost FPGAs, (ii) steady performance which means: (a) performance is not affected by the number of rules which are examined, (b) worst case performance is close to average case performance in real data scenarios. (iii) scalability, since it can easily support a much larger number of rules without changing the architecture or compromising the performance.

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.

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.

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. [7].

The main advantage of the tuple space search algorithm 0 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 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.

Our 2sBFCE 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. The proposed scheme targets the same application scenarios as [11] and follows a similar implementation approach, but it is more efficient than [11] as the performance section demonstrates.

III. 2sBFCE DESIGN

The 2sBFCE design 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: I. Current filter sets size 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.

II. The protocol field is restricted to a small set of values; TCP, UDP and wildcards are most commonly (i.e. more than 95%) used.

III. 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’.

IV. The number of unique address-prefix rules matching a given source or destination address is typically five or less.

V. The number of single field filters matching a given packet is typically five or less.

VI. Different multi-dimensional rules very often share a number of single-field values.

VII. The number of single field values is significantly less than the number of overall filters.

A. Single Field Operations

Given that 2sBFCE 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 a bloom filter scheme described in the following section. Our single-field lookup mechanism not only report the longest prefix match but, instead, all the prefixes that match, and for each match the associated match length, as described in [19]. Moreover, and since 2SB2PC 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].

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. The number of supported 5-tuple rules could be easily increased without affecting either the architecture or the overall performance of the design. A general overview of the 2SB2PC scheme is presented in Figure 1 where all the discrete components are shown. The idea of 5-tuple classification is based on single field classification. Therefore, for each field we perform all prefix match lookup and then using a permutation process we combine these results.

Figure 1 : Overall Architecture of 2sBFCE

B. Internally Represented Filters

The internal representation of prefix variations in each field is based on the combination of each IP field with the corresponding prefix length. No other internal Id is assigned. This scheme produces a representation of 32+6 bits for the SourceIP and DestIP prefixes, 16+5 bits for the Source-Port and Dest-Port prefixes and last 2 bits for the Protocol (no prefix length used). Each of these representations is applied to an individual Bloom Filter-based classification engine (sub-engine). Since Protocol comprises of just two bits, in our implementation it is treated in a different way as it will be explained later.

During the store procedure each single field prefix is stored at the corresponding sub-engine and the matching Bloom Filter (BF) element is set. In parallel, all these single field representations are combined to a single 120-bit “rule vector” using the Permutation Engine. This vector is applied to the second BF Stage in order to store the information of this specific combination of each single field which is needed for the later query procedure. This 120-bit “rule vector” is also combined with a unique 12-bit flow ID and kept in the 4K entry RulesTable. The RulesTable is indexed by a hash function using the 120-bit vector information as input. This hash indexing scheme introduces a collision probability during store procedure which could lead to an unwanted rule overwrite. This problem is described in section H. Table 1. shows an example of a real-like filter set.

C. Combining Results

Given the 5 fields of a packet, the 2sBFCE 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. 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, resulting in at most 2 matches. For that reason we don’t dedicate a separate sub-engine for the protocol filed, we just deal with it at the last stage of the procedure. 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). The above task is performed during query procedure in the Permutation Engine module. The Permutation Engine combines all the matches of each single field into a number of possible matching 120-bit “rule vectors”. Each of these vectors is examined in the second stage Bloom Filter scheme as explained in Section E. The total number of possible permutations is equal to the overall product of the number of matches in every field:

Total\textsubscript{perm} = \#Src IP * \#Dest IP * \#Src Port * \#Dest Prt

Protocol is not included because we don’t deal with it at this moment, leaving this check in the end of the process. We are doing this to reduce in half the number of permutations as it is meaningless a 4-field matching combination to have two different entries varying only in protocol field. Each such combination should have either a specific value or a wildcard in protocol field.

Although the possible number of permutations could be large, in real-world databases, as it was described in Subsection III.IV, 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.

D. Set Membership Queries with Bloom Filters

As mentioned before the combined results from every single-field engine should be examined so as to find out which of these are actually valid. One of the most important challenges of 2sBFCE, 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 rule set and support quick set membership queries, is needed. Hash tables and B-Trees are widely used for this type of queries but we use Bloom Filters. 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 sometimes reports that a certain item is part of the set, even though it does not actually belong to this set (i.e. false-positive error). This happens due to the hash-based structure they use. As a result, for every combination which is a positive match in the second BF stage, we consider this combination as a possible multi-field matching rule. We still say “possible” because there is a false positive probability as mentioned before. We deal with this false positive scenario in the last stage of RulesTable. A similar scenario could also happen during the single field query procedure. These false positive results don’t affect the final result because they are automatically resolved during the second BF stage. However, they do affect the overall performance of the lookup as described in subsection F.

E. Bloom Filter Tuning

In the 2sBFCE, in order to efficient support classification databases with up to 4K rules, we employ a suitable Bloom Filter. 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 [19] we ended-up with a bit vector which is 3*2^12 bits wide (we used 3 identical BF Elements -BFE). Based on this same analysis the optimal number of hashing functions that set the bits on this vector is 2. Given those parameters, the produced BF has a theoretical false positive probability of 2.36 %.

The bit-vector of the BF is relatively large to be kept in registers/flip-flops, and therefore it is stored in a memory array. Having 2 hash functions means that we have to set 2 bit positions in the bit vector and 2 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 two memory accesses to locate each of the 2 bits. Thus, in order to avoid sequential accesses, and since the array is quite small and can easily be kept on-chip. For that reason we use the dual-port Memory modules of the Xilinx Virtex4 FPGA. These modules have appropriate configured to have 1bit of width and to each port we assign a different hash function. 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 to the bit vector.

After careful analysis of the classification databases and the Bloom Filter properties, we have defined a hash function based on H\textsubscript{3} Class of Universal Hash Functions [20]. As it is supported, this class of hash functions, is suitable for hardware applications because of its computational simplicity and its high level of parallelism. The following formula describes the function we used:

\[ h(x) = (x1 \cdot q(1) \oplus q(2) \oplus \ldots \oplus x(i) \cdot q(i)) \] where \( \cdot \) denotes the bitwise AND operation and \( \oplus \) the bitwise XOR operation. The \( q(i) \) derives from a randomized vector. Figure 3, shows an example of producing a 2-bit hash value for a 3-bit Key[21].

F. False Positives and Filter Tuning

Although the false positive matches in the first level of the single filed classifiers are easily resolved by the second bloom filter level, they produce a large number of redundant permutations. This can result in decreased performance. For that reason we have appropriate adjusted the parameters of the BF in order to reduce the false positive rate. For the IP fields (source , destination) we chose to use some dedicated filters. Each of those classifier module uses 1 dedicated bloom filter for the most common prefix lengths (16,24,32). A statistic analysis was performed in real-like data Rules Tables generated by ClassBench tool[14]. ClassBench involves a filter set generator that uses seeds from real filter sets to provide synthetic filter sets that accurately model real filters.

G. FlowID Resolving and bloom filter collisions

To locate the FlowID we use the 4K-entry RulesTable that give us the matched FlowID. As mentioned in subsection D, once we have a valid permutation we use a hash indexing scheme to locate the appropriate RulesTable entry. After this we should determine whether it is a false positive match and in case it is not, we have to return the corresponding FlowID. In order to accomplish this task efficiently, during the store procedure, we keep the whole information of the rule in the RulesTable in combination with its FlowID. In that way, for each valid permutation, we read the whole rule from the appropriate entry of the Rules Table and then we compare each field with the corresponding field of each permutation. If we have a full match then we send the 12-bit
FlowID to the output pins; in the opposite case a false positive match has occurred. Then, we continue with the testing of the rest of the permutations. At this specific stage of the procedure we deal with the protocol field, by comparing as described before, the appropriate fields. More specific we compare the information carried in the packet with this stored in each valid rule (including wildcard). If this field is a match (as well the as the other fields) then we have a valid rule match.

In a single query it is possible to have more than one matching rules. To find out which of these matching rules is the best-matching we have to set some priorities among the 5-tuples first. These priorities may change from time to time so we leave this additional check to an upper layer of control. For that reason our classifier could reply with more than one matching result, until a Query_Done signal is set to 1.

H. Indexing the Rules_Table and Incremental Updates

Indexing the Rules Table requires a hash function and obviously this function may produce collisions. Resolving these collisions can be performed by using variable size blocks (such as in [11]) that hold the colliding FlowIDs. In the proposed implementation and for simplicity we have just one memory block where we put all the colliding FlowID and in case there is a collision we search all of them sequentially. In the next version of the classifier we plan to adopt a scheme similar to [11].

A property of Bloom filters is that it is “very hard to delete a message stored in the filter” [13]. This happens due to the hash based structure of BF which may causes collisions to some specific hashed bits in the bit vector. Deleting a particular entry requires that the corresponding hashed bits in the bit vector to be set to zero. This could disturb other messages programmed into the filter which hash to any of these bits. Therefore, in the current scheme we don’t support incremental updates. In the next version and in order to cope with this problem we plan to maintain a vector of counters. Whenever a message is added to or deleted from the filter, the counters corresponding to the k hashed values are incremented or decremented, respectively. We delete a particular entry only when the corresponding counter reaches zero.

IV. SIMULATION RESULTS

In order to measure the efficiency of our scheme we employed a large number of 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. 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, as it is widely supported, 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.

A. Improving Memory Access

As described in the last section, the lookup performance of the proposed scheme mainly depends on the set-membership queries in the Bloom filters. This is because the set-membership derives from the overall product of the single filed results. In contrast the performance of the single field sub-engines remains practically constant as it needs 30-33 on-chip memory accesses. About 1 cycle for each prefix number which is examined and all the queries run in parallel.

In the implemented scheme we used a rather different scheme: instead of 1 we dedicate 2 bloom filter sets for each of the SourceIP and Dest.IP sub-engines. This scheme improves the performance of these two sub-engines by 16 clock cycles as it splits the 32 possible prefix lengths in two smaller sets. Each set is consisted of 16 possible prefix lengths and is applied to a separate Bloom filter. Thus the whole single-field query process finishes in about 17 cycles. The other two sub-engines (source & destination port) always produce their results in at most 17 clock cycles. As a result, the whole single-field query procedure finishes in about 17 clock cycles.

The second stage needs one on-chip memory access per each permutation produced, as we dedicate one Bloom Filter for the entire set of permutations. Therefore the whole number of permutations depends on the data applied to the system. Simulation of real-world filter sets shows that the second stages needs 10-15 clock cycles in average.

B. Storage Requirements

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 2sBFCE for all the generated filter sets. To calculate the total storage for 2sBFCE we measure the storage requirements both of the single field Sub-engines and the second stage Bloom Filter. Our scheme employs only 178 KB of on-chip SRAM, and as Table 3 demonstrates, it is the most memory efficient when comparing with other classification schemes.

V. HARDWARE DEVICE’S COST & PERFORMANCE

The proposed scheme has been Synthesized and Placed and Routed in FPGA technology and it works at 153MHz (6.56ns clock period). The target device was a Xilinx Virtex 4, 4vfx40f672.

Our primary design performs each query in 42 clock cycles (in average), thus it needs 42 x 6.56ns = 275.5 ns per packet. That gives 3.63 MegaPackets Per Second (Mpps) at 153MHz. Our improved design minimizes the cycles per query by 16 cycles, thus it needs in the average case 26 clocks cycles per packet look-up. This results in a rate of 5.86 Mpps.

Table 2 presents the network performance of 2sBFCE 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 5 Gbps or higher.

Moving to the comparisons with the existing schemes and according to Table 4, it is clear that, despite the fact that RFC has the best throughput, its performance is based on large memory utilization, while it supports at most 1700 rules. On the other hand ABV does provide the highest number of Mpps/Mbytes, but it supports 80% less rules than our scheme. Considering that 4K rule set size or more is the upcoming standard, our scheme is one of the best proposed schemes along with B2PC[19].
Table 2: Worst Case Network performance of 2sBFCE

<table>
<thead>
<tr>
<th>Filter Set Name</th>
<th>Mpps</th>
<th>Gbps (Worst Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2sBFCE</td>
<td>3.63</td>
<td>5.86</td>
</tr>
<tr>
<td>2sBFCE optimized</td>
<td></td>
<td>1.16</td>
</tr>
<tr>
<td>Combined</td>
<td>1.87</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Summary of Classification Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Working Frequency (MHz)</th>
<th>Number of Rules</th>
<th>Storage Requirements (Number of memories)</th>
<th>Throughput (Mpps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2sBFCE</td>
<td>153</td>
<td>4k</td>
<td>178Kbytes</td>
<td>5.86</td>
</tr>
<tr>
<td>BV[15]</td>
<td>33</td>
<td>512</td>
<td>640KB (5)</td>
<td>1</td>
</tr>
<tr>
<td>RFC[4]</td>
<td>125</td>
<td>1700</td>
<td>976 KB (2) + 15,6 MB (2)</td>
<td>30</td>
</tr>
<tr>
<td>B2PC[19]</td>
<td>400</td>
<td>3300</td>
<td>540 KB (4)</td>
<td>42.5</td>
</tr>
</tbody>
</table>

Table 4: Schemes efficiency in Mpps per Mbyte

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33,8</td>
<td>18,9</td>
<td>106,2</td>
<td>5,8</td>
<td>78,7</td>
<td></td>
</tr>
</tbody>
</table>

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, very low-memory, devices the low-memory scheme of [10] would probably be preferable. Moreover, for devices supporting relatively small filter sets ABV seems to be 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 than 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 very low 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 178KB of inexpensive external SRAMs, while it can support network rates higher than 4Gbps in an FPGA implementation. This scheme has been designed so as to be efficiently implemented in hardware. As our performance results demonstrate, given a certain memory budget, number of supported rules and silicon cost, 2sBFCE provides the highest performance when compared to all the similar systems implemented in hardware.

ACKNOWLEDGEMENT

Part of the work presented in this paper has been carried out within the ICT AWISSENET project (Grant agreement no 211998), financed by the European Community. The authors would like to thank all the partners of the AWISSENET consortium for their contributions.

REFERENCES

[18] Ioannis Papaefstathiou, Vassilis Papaefstathiou “Memory-Efficient 5D Packet Classification at 40Gbps”. Infocom 2007