Packet Classification using Rule Caching

Nitesh B. Guinde, Roberto Rojas-Cessa and Sotirios G. Ziavras
Electrical and Computer Engineering Department
New Jersey Institute of Technology
Newark, NJ 07102, USA
Email: {nbg2, rojas, ziavras}@njit.edu

Abstract—**Data rates on Internet links keep increasing with the deployment of optical technology. Packets coming into high-speed networks need to be classified quickly. Different packet classification schemes have been developed but they require a number of memory accesses as classification is complex and memory is slow. We follow the approach of providing support with fast memory, as cache, in computer systems, to support packet classification schemes. Here, we propose a scheme based on memory cache to support packet classification. The scheme not only makes use of faster and smaller memories but also reduces the number of memory accesses to perform packet classification. It can make the performance of the adopted classification scheme independent of the number of connection flows. We present various packet-classification caching schemes for performing classification and provide the cache hit ratio results for various traffic models generated with Classbench.**

Index Terms*—*Packet classification, caching.

I. INTRODUCTION

Packet classification is an important and complex problem to solve [10]. It is used most commonly in firewalls, edge routers, network translators, and networks that provide quality of service (QoS) guarantees. It is applied to the forwarding functions required for resource reservations, QOS routing, unicast routing and multicast routing provided by Internet routers. The forwarding database in the routers typically consists of a large number of rules arranged in a priority order so that the best matching rule is placed first. A rule is a combination of *d* header fields.

The matching of the fields requires three different matching mechanisms. *Exact match*: In this matching scheme, the header of the packet should match exactly the field in the rule. It is usually applied to match protocol numbers. *Prefix match*: In this case, the prefix mask and the value are provided in the rule field, and the header of the packet should match the prefix of the field. It is used for matching network addresses. *Range match*: A range is given in the field and, the port number of the header of a packet should be within a (rule) given range.

Each rule has an action associated with it, which corresponds to the operation that needs to be performed on the incoming and matched packet. This action could be either accepting or dropping of packet (in the case of a firewall) or forwarding the packet to a particular port (in the case of a router), providing differentiated service to it, or some other action.

The function of the classification engine is to find the best matching rule that suits the incoming packet based on the header fields. There is a plethora of algorithms developed to speed up the process of packet classification [3, 4, 5, 6, 7, 8, 9]. Still, as the line rates are increasing, the ability to perform classification at high speeds could be overwhelming for high-speed networks. Research work has also been carried out in the router architecture design area to improve the speed of this classification [11, 12, 13, 14]. TCAMs are the a popular choice in terms of architecture-based solutions since they offer high performance in terms of speed and provide fast lookups independent of the characteristics of the rule sets. However, they are not well suited for large classifiers because of low density, high power consumption, and relatively high cost.

Some approaches use caching to exploit the temporal locality of traffic. We can classify them as two types of carrying out caching, namely flow caching and rule caching. Caching has also been examined from the accuracy perspective of the design space using Bloom filters. However, misclassifications (i.e., false positives) are possible when involving such an approach.

Most of the solutions delve into flow-based caching since packets belonging to a flow are seen to follow a temporal locality pattern. Thus, it helps to cache the flow identification fields to a certain extent since all the packets belonging to the cached flow could be worked upon in a faster manner. However, the number of active flows at any given instance could be large, in the millions, while most flows are active for a small duration.

Also, as the link rates increase, the number of flows at classification nodes is also expected to increase, thus requiring the cache memory to scale up to support the rising speeds but at the same time, to ensure high hit rates.

As an alternative, rule caching seems to be a potential solution for scaling up classification speeds since the number of rules would be generally much smaller than the number of concurrent active flows at the node. A technique was proposed to improve the search performance using characteristics inherent to Internet traffic [1]. A similar approach was used in [2] by employing a smart cache for evolving rules; it preserves classification semantics and uses additional logic to match incoming packets to these rules. Our method studies the various effects of traffic changes and cache size changes on the cache hit ratios.

II. OUR METHOD

We use the concept of caching of rules using the least-recently used (LRU) policy to replace information in the cache. Our purpose is to study the effects of different kinds of traffic, including randomly generated traffic patterns, on the cache hit ratio with our proposed scheme. For example, let us consider the classifier Table I shows. We first map the rules using a simple one-bit trie. Since, with our method, the packet classification scheme is only relevant in retrieving the rule from the main memory, irrespective of how many memory accesses it takes to access the rule in the cache, any packet classification scheme would suffice.

We create two tries, one for the source IP and the other one for the destination IP, as shown in Figures 1 and 2, respectively.

TABLE I: Example rules

Rule number	Source IP	Desti-nation IP	Source Port	Desti-nation Port	Protocol	Flags
R1	1100*	111*	0:5	16:20	TCP	SYN
R2	1100*	111*	0:65535	16:20	TCP	SYN
R3	0110*	0110*	1200:1200	3111:3111	TCP	ACK
R4	1011*	1*	0:65535	500:1500	TCP	*
R5	0*	0*	0:6	0:65535	TCP	*
R6	*	*	*	*	*	*

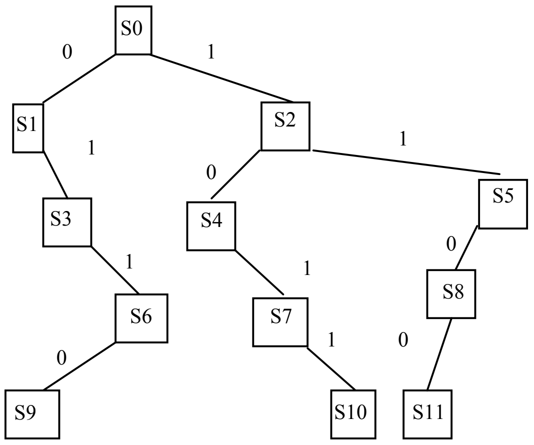

Fig.1 Source IP trie

For our caching scheme, we would like to move the whole rule into the cache. Along with it, we also need to move some additional data structures that will help us in performing the decision making process on packet matching; i.e., to determine whether the rule needs to be classified using the packet classification scheme or whether the packet could be classified using the rules in the cache. Thus, we first allot priorities to the rules according to their precedence. If a rule has its priority bit set to 1, it means that there is no rule of higher priority in the classifier that could be matched, if a rule match is observed at the cache.

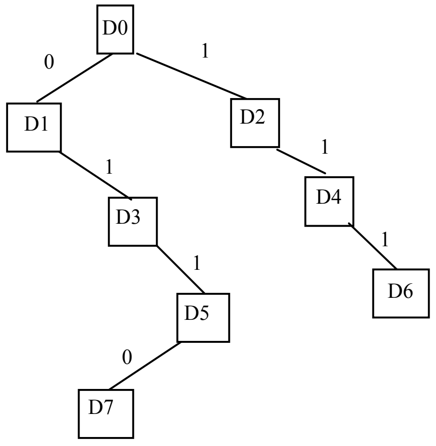

Fig.2 Destination IP trie

Thus, if a rule is in the cache with its priority bit set to 1 and an incoming packet matches the rule, then we can be rest assured that there is no better rule in the main memory; hence, we can make a decision without going to the main memory. The priority bit for a rule r is set to 1 using any of the following conventions: (1) there does not exist any rule which could be matched if r is matched; or (2) there exists no rule of higher precedence which could be matched if rule r is matched.

For example, if we look at Table I, we can see that R1 is much more precise compared to R2; at the same time, it is given a higher priority than R2. Therefore, the priority bit for rule R1 is set to 1 and that for rule R2 it is set to 0. However, rules R3 and R5 will have their priority bit set to 1 because both are mutually exclusive rules.

The priority bit gives us some information regarding the rule in the cache and, to some extent, helps in making a decision if a rule match occurs. However, in the case of a rule match in the cache with the respective priority bit set to 0, additional rules need to be retrieved from main (and slower) memory. We can decrease the number of steps in trie traversing by directly checking the rules that could be matched and those that have a priority higher than the rules in the cache. To achieve this, we need to store two address pointers, the source IP trie address pointer and the destination IP trie address pointer, only for the rules with the priority bit set to 0.

The rule data structure contains the following members: {source IP, destination IP; source port range (high, low);

destination port range (high, low); protocol; flags; priority bit; source IP trie address pointer; destination IP trie address pointer}. Let us look at an example where we assume that rule R2 is in the cache and a TCP ack packet arrives with source IP address 1100…. , destination IP address 111….., source port 6, and destination port 17. A match is found in the cache but the matching rule holds the priority bit set to 0; hence, retrieval of more rules from the main memory, using the source address trie pointer and the destination IP trie address pointer, is needed. These pointers will take us directly to nodes S11 and D6, which contain rule R1.

Since a match will not be obtained, we move to the next rule, which is rule R2. Table II shows the priority bits and the address pointers of the respective rules.

TABLE II: Priority bits and address pointers

Rule number	Priority Bit	Address Pointer
R1	‘1’	S11, D6
R2	‘0’	S11, D6
R3	‘1’	S9, D7
R4	‘1’	S10, D2
R5	‘1’	S1, D1
R6	‘0’	S0, D0

In the Classbench filters [17], we notice that some sets of filters (especially fw1, …, fw5) have a lot of rules with all wildcard characters or predominantly wildcard characters in either the source IP field or the destination IP field. This causes some problems in differentiating the rules, with a lot of rules having a priority bit set to 0 and thereby increasing the misclassification ratio. For example, consider the following set of rules from a sample fw5 file sorted in a best matching order; Table III shows some similar rules.

TABLE III: Example rules

R11	0.0.0.0/1	81.170.248.180/32	750 : 750	113 : 113	*	*
R12	0.0.0.0/1	81.255.109.64/32	123 : 123	50 : 50	*	*
R13	0.0.0.0/0	76.140.76.255/32	53 : 53	53 : 53	*	*
R14	0.0.0.0/0	235.152.138.85/32	67 : 67	22 : 22	*	*
R15	0.0.0.0/0	163.22.84.133/32	123 : 123	37 : 37	*	*
R16	69.63.137.234/32	0.0.0.0/1	0 : 65535	0 : 65535	*	*
R17	166.1.17.247/32	0.0.0.0/0	123 : 123	22 : 22	*	*
R18	165.69.41.196/32	0.0.0.0/0	53 : 53	53 :53	*	*
R19	160.166.194.42/32	0.0.0.0/0	67: 67	0 : 65535	*	*
R20	183.56.132.227/32	0.0.0.0/0	750 : 750	53 : 53	*	*

In the rules of Table III, if a packet with a source IP: 69.63.137.234, destination IP: 81.170.248.180/32, source port: 750, destination port: 113 arrives, then R11 is the best matching rule for the packet although other rules would also match. However, this is just one case while for many other packets which have the same source IP as the above but different destination IPs, the best matched rule will be R16.

The probability that a packet with a source IP: 69.63.137.234 will match rule R11 is one out of 2^{64} chances; in all the remaining cases, it will match R16. For such cases, the addition of a priority bit alone does not help and results in a higher rate of misclassifications. This is because even if we move rule R16 to the cache and all the incoming subsequent packets match R16, using the above method we still have to go to the main memory to find more information about the matching. This will make the classification scheme very inefficient as a miss would make the classification process incur into a large (time) penalty by accessing main memory. To avoid such a scenario, we provide an additional data structure,

More specifically, we create a data structure that is 256 bits long and has information about the dependent rules. Let us choose the same rule R16; the dependent rules with higher priority are R11, R12, and R13. We keep a 256-bit vector for every byte in the field. The vectors contain all zeroes. We change the position of the bit pointed to by the value in the byte to 1. Finally, we keep the bit vector that contains the smallest number of 1’s along with the marker signifying the byte number of the bit vector. The rest of other bit vectors are deleted.

For instance, in the above example the first byte bit vector will have two 1’s, one in position 81 and the other one in position 76, as shown in Fig. 2. Similarly, the second byte bit vector will have three 1’s in bit positions 140, 170, and 255. The third byte bit vector will have three 1’s in positions 76, 109, and 248, whereas the fourth byte bit vector will have three 1’s in positions 64, 180, and 255. Thus, the first byte will be chosen since the probability of hitting a 1 in the case of the first byte is the lowest and, hence, will result in the lowest probability of having a misclassification.

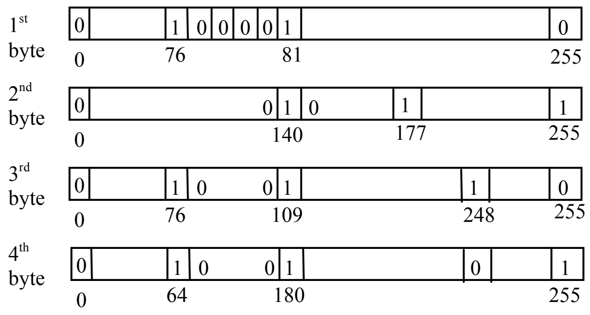

Fig. 2: 256-bit vectors

We tested the use of the data structures on the files which had a high misclassification ratio. We observed high drops in the misclassification ratio and a corresponding rises in the cache hit ratio.

Using the above data structures, we implemented the following caching policies and compared the results.

TABLE IV: Results

			Old Method		CASE II	
File	Corr of traffic	Number of packets	Cache hit ratio	Mis-Classific. Ratio	Cache hit ratio	Mis-Classific. Ratio
fw4	1.0	1M	0.468	0.441	0.73	0.17
fw5	0.5	1M	0.22	0.631	0.70	0.15
fw5	0.5	2M	0.142	0.781	0.751	0.172
fw5	1.0	2M	0.301	0.614	0.703	0.212

Method I : This method uses a simple LRU scheme where the rule is moved to the head of the cache when there is a hit while it is moved downwards towards the tail with every new addition of a rule in the cache. Thus, when the cache is full the rule, which is at the bottom is moved out of the cache and a new rule is inserted at the top. Also, we only use a priority bit without the 256-bit vector.

Method II : This method is the same as method I with the difference that we use the 256-bit vector along with the priority bit.

Method III : We use the frequency-based replacement method for packet caching. It is very similar to the one used in [15] for managing caches with disk blocks in a file system. The method yields very good results in terms of the cache hit ratio, the number of cache accesses and, finally, the number of main memory accesses.

According to Method III, the cache is divided into three sections: namely the NEW, MID and OLD sections. As soon as a rule is hit in the main memory, it is moved into the head of the NEW section in the cache and its count is set to 1. As new rules get added into the cache, the old rules move downwards into the MID section. If there are no further hits, then the rule moves into the OLD section from where it moves eventually out of the cache.

While in the cache, if the rule is hit then it is moved to the head of the NEW section irrespective of its position in the cache. Every rule in the cache has a reference count associated with it. The reference count of the rule is not incremented in the NEW and OLD sections. However, if the rule hit is in the MID section, then its count is incremented and the rule is moved to the head of the NEW section. If the cache is full, then we remove from the OLD section the rule that has the least count. We use the LRU policy in the case multiple counts are equal; i.e.; the rule at a lower location in the OLD section is removed.

Method IV : Under this method, we study the effect of delaying the caching of rules as soon as there is hit. In other words, we maintain a separate count for the rule in the main memory and cache. If a rule is hit in the main memory, we increment the counter pertaining to the rule. We do not cache the rule until the count for the rule crosses a threshold. We maintain a sampling window of T time slots (i.e., incoming packet counts). At every t=n*T, for n= 1, 2, 3, …, we transfer into the cache the rules that have counts bigger than the chosen threshold. For the rest of the time slots, no transfer takes place; only the count values of the rules hit in the main memory and the cache are incremented. Once a rule is moved into the cache, the count of this rule is preset to 1. We then increment the count of the rule in the cache if it is a hit. When the cache is full, then we replace all the rules in the cache that have count values smaller than the chosen threshold with the ones from the main memory which have accumulated counts greater than the threshold. In case there are no rules with count values greater than this threshold in the main memory, then no transfer takes place.

Also, in the case where there are no rules in the cache with count values less than the threshold, then we compare the counts of the rules in the cache against the counts of the rules in the main memory (we focus on the rules that have counts greater than the chosen threshold); we replace a rule in the cache if the count of a rule in the main memory exceeds the count of the rule in the cache. In case where there are two such rules in the cache and only one rule in the main memory, then we use the LRU scheme according to which we replace the rule in the lower location of the cache. The count values of all the rules in the cache are preset to 1 at time slots t= n*T. We show results for n=1 and a threshold value of 3 in Table V.

Method V : This method is very similar to the last method except that we remove the MID section in the cache. We also increment counts in the OLD section. Here, the OLD section is very wide containing 768 locations.

The results in Tables V through VII are for the traffic model from file ACL1 and high, medium and low traffic locality, respectively. We notice that Method IV yields the best results in terms of the cache hit ratio for all of the Classbench files. Fig. 4 shows a graph with the Y-axis representing the cache hit ratio for various files shown on the X-axis.

TABLE V: Results for the traffic model from file ACL1

Traffic Type (Locality)	Method	Cache Hit Ratio	Mis-classification Ratio	Average Number of main memory accesses without cache	Average Number of main memory accesses with cache	Average number of cache accesses
High	I	0.828	0.116	88.32	8.47	61.55
		0.81	0.118	86.71	9.27	72.19
	II	0.831	0.113	88.32	8.39	61.55
		0.82	0.114	86.71	9.19	72.19
	III	0.831	0.113	88.32	8.39	61.55
		0.82	0.114	86.71	9.19	72.19
	IV (n=1)	0.841	0.105	88.32	8.1	68.73
		0.838	0.105	86.71	8.53	71.16
	V	0.83	0.113	88.32	8.39	82.44
		0.82	0.114	86.71	9.18	98.79

III. Conclusion

We presented various caching policies for packet classification and studied the effects of each one of them on the cache hit ratio using Classbench. Our basic method for packet classification can make the classification scheme independent of the number of connection flows. We also obtain a speedup on the packet classification process by supporting a classification scheme with the proposed caching method.

TABLE VI

TRAFFIC TYPE (LOCALITY)	METHOD	Cache Hit Ratio	Mis-classification Ratio	Average Number of main memory accesses without cache	Average Number of main memory accesses with cache	Average number of cache accesses
Medium	I	0.796	0.103	88.00	12.2	111.08
		0.77	0.095	88.6	14.23	145.45
	II	0.80	0.098	88.32	12.13	111.08
		0.77	0.095	86.71	14.16	145.45
	III	0.8	0.1	88.32	12.13	111.09
		0.82	0.114	86.71	9.19	72.19
	IV (n=1)	0.84	0.105	88.32	10.11	110.34
		0.825	0.098	86.71	9.37	108.46
	V	0.83	0.113	88.32	8.39	82.44
		0.82	0.114	86.71	9.18	98.79

TABLE VII

TRAFFIC TYPE (LOCALITY)	METHOD	Cache Hit Ratio	Mis-classification Ratio	Average Number of main memory accesses without cache	Average Number of main memory accesses with cache	Average number of cache accesses
Low	I	0.53	0.084	87.43	34.50	422.24
		0.696	0.045	90.43	23.24	287.53
	II	0.538	0.081	87.43	34.41	422.24
		0.699	0.042	90.43	23.16	287.53
	III	0.538	0.081	87.43	34.41	422.24
		0.699	0.042	90.43	23.16	287.53
	IV(n=1)	0.612	0.077	87.43	27.33	350.08
		0.703	0.039	90.43	21.23	233.25
	V	0.538	0.081	88.32	34.39	445.32
		0.699	0.042	86.71	23.15	295.51

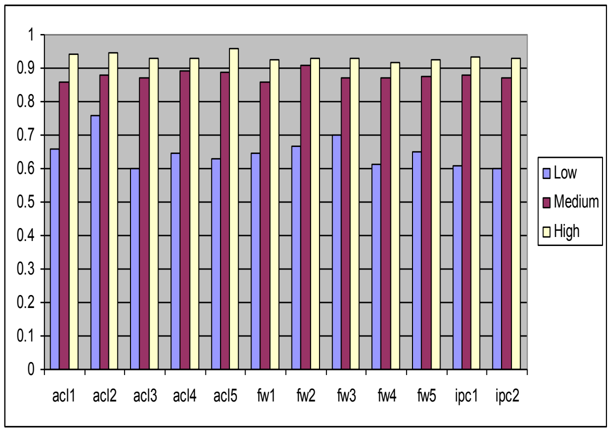

Fig. 4: Cache hit ratio using Method IV for various Classbench files

REFERENCES

[1] J. Philip, M. Taneja and R. Rojas-Cessa, "Rule Caching for Packet Classification Support", in Proceedings of IEEE Sarnoff Symposium, Princeton, NJ, April 28-30, 2008.

[2] Q. Dong, S. Banerjee, J. Wang and D. Agrawal, "Wire Speed Packet Classification without TCAMs: A Few More Registers (And a Bit of Logic) Are Enough"

[3] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in Proceedings of ACM SIGCOMM, pp. 147-160, August 1999.

[4] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in Proceedings of Hot Interconnects VII, August 1999.

[5] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multidimensional Range Matching," in Proceedings of ACM SIGCOMM, pp. 203-214, September 1998.

[6] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Fast and Scalable Layer Four Switching," in Proceedings of ACM SIGCOMM, pp. 191-202, June 1998.

[7] F. Baboescu and G. Varghese, "Scalable Packet Classification," in Proceedings of ACM SIGCOMM, pages 191-202, August 2001.

[8] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in Proceedings of IEEE INFOCOM, pp. 1193-1202, March 2000.

[9] T.Y. C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," in Proceedings of IEEE INFOCOM, Vol. 3, pp. 1213-1222, March 2000.

[10] M.H. Overmars and A.F. Van der Stappen, "Range searching and point location among fat objects," Journal of Algorithms, 21(3), pp. 629–656, November 1996.

[10] G. Gibson, F. Shafai, and J. Podaima, "Content Addressable Memory Storage Device," United States Patent 6,044,005, SiberCore Technologies, Inc, March 2000.

[11] R.A. Kempke and A.J. McAuley, "Ternary CAM Memory Architecture and Methodology," United States Patent 5,841,874. Motorola, Inc, November 1998.

[12] A. J. McAulay and P. Francis, "Fast Routing Table Lookup Using CAMs," in Proceedings of IEEE INFOCOM, Vol. 3, pp. 1382-1391, 1993.

[13] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," in Proceedings of IEEE International Conference on Network Protocols (ICNP), pp. 120-131, 2003.

[14] K. Lakshminarayanan, A. Rangarajan, S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs." ACM SIGCOMM Computer Communication Review, Volume 35, Issue 4, pp.193–204, October 2005.

[15] J.T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement" in ACM SIGMETRICS Performance Evaluation Review 1990, pages 134-142.

[16] N. Guinde, S.G. Ziavras and R. Rojas-Cessa, "Efficient Packet Classification on FPGAs also Targeting at Manageable Memory Consumption," in Proceedings of the 4th International Conference on Signal Processing and Communication Systems, Gold Coast, Australia, 2010, December 13-15, 2010.

[17] D. Taylor and J. Turner, "ClassBench: APacket Classification Benchmark," Proc. IEEE Infocom 2005, vol. 1, pp. 2068-2079, 13-17 March, 2005.