Comparative Evaluation of Packet Classification Algorithms for Implementation on Resource Constrained Systems

Comparative Evaluation of Packet Classification Algorithms for Implementation on Resource Constrained Systems

Gianluca Varenni*, Federico Stirano***, Elisa Alessio**, Mario Baldi*, Loris Degioanni*, Fulvio Risso*

* Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy
**Telecom Italia Labs - System On Chip, Torino, Italy
***Istituto Superiore Mario Boella, Torino, Italy

{gianluca.varenni,mario.baldi,loris.degioanni,fulvio.risso}@polito.it; stirano@ismb.it; elisa.alessio@tilab.com

Abstract – **This paper provides a comparative evaluation of a number of known classification algorithms that have been considered for both software and hardware implementation. Differently from other sources, the comparison has been carried out on implementations based on the same principles and design choices. Performance measurements are obtained by feeding the implemented classifiers with various traffic traces in the same test scenario. The comparison also takes into account implementation feasibility of the considered algorithms in resource constrained systems (e.g. embedded processors on special purpose network platforms). In particular, the comparison focuses on achieving a good compromise between performance, memory usage, flexibility and code portability to different target platforms.**

I. INTRODUCTION

A vast literature on classification algorithms and their performance does exist, but our work is necessary, hence relevant since existing evaluations do not allow a significant comparison based on real-life data. In fact, a comparison based on existing literature could be carried out only according to analytical worst-case bounds. Even though figures on the performance of classification algorithm implementations in real-life scenarios can be found, they are part of studies on a single algorithm: the measurement scenarios are different and the implementations are not uniform, consequently the results are not comparable.

This work studies known classification algorithms with respect to their suitability for being (*i*) deployed for common networking applications (i.e., not optimized for a specific one), and (*ii*) implemented in embedded systems, i.e., systems with strict requirements, limited resource availability, and no specific hardware support, such as content addressable memories.

A (packet) classifier is a collection of *rules* — usually called *ruleset* — that is used to partition network traffic into different groups, sometimes called *flows* or *buckets*. Every rule specifies a subset of the network traffic, for example "IP traffic", or "traffic sent from host 1.2.3.4", thus somehow characterizing packets grouped into that flow. When a packet satisfies a rule, the packet is said to *match* the given rule. A *classification algorithm* determines whether a packet matches at least one rule of a classifier.

Packet classifiers are widely used in IP networking where rules usually involve one or more packet header fields (e.g. IP source address, TCP destination port). Each rule *R* is composed of *i* components, so that each component *R[i]* applies to a specific header field. When more than one field is considered, the classifier is said to be *multifield*. As an example, Table 1 shows a small multifield ruleset that includes value/mask rules on the source and destination IP addresses.

Packet classifiers are widely used for various network applications, many of which related to quality of service (QoS) provision, and consequently in several types of network devices that might be implemented as or composed of embedded systems. Examples of QoS related applications of packet classifiers are:

- *Traffic conditioning and shaping appliances*; they use multifield classifiers, usually on session tuples, to separate traffic flows in order to be able to apply on them admission, marking and shaping policies. Traffic conditioning appliances or functionality are fundamental whenever in the deployment of both the IntServ [1] and DiffServ [2][3] approach.
- *IntServ routers*; they use multifield classifiers, usually on session tuples, to separate traffic flows in order to store packets in different queues on which scheduling algorithms suitable to provide the required QoS are applied.
- *DiffServ routers*; they use single field classifiers based with a limited ruleset concerning the value of the DS (Differentiated Services) field [3] to separate packets belonging to different traffic classes in order to handle them according to the corresponding per-hop behavior (PHB).

This work aims at identifying classification algorithms that can be effectively implemented on embedded systems and deployed in any of the above listed applications. Execution in embedded systems imposes strict limits on the characteristics of the algorithms, such as simple (static) memory management, limited code size, limited CPU usage requirements, limited data storage necessities,

TABLE 1. SAMPLE MULTIFIELD RULESET

	IP source		IP destination	
Rule 1	Value	= 130.192.1.0	Value	= 130.192.2.0
	Mask	= 255.255.255.0	Mask	= 255.255.255.0
Rule 2	Value	= 130.192.2.0	Value	= 130.192.1.0
	Mask	= 255.255.255.0	Mask	= 255.255.255.0
Rule 3	Value	= 130.192.0.0	Value	= 130.192.3.0
	Mask	= 255.255.0.0	Mask	= 255.255.255.0

adaptability to various hardware platforms and architectures.

Our work, and this paper describing it, was organized as follows. The various algorithms proposed in the literature (Section B) as well as the metrics commonly deployed to evaluate them (Section A) are first surveyed. The implementation objectives and the guidelines followed to develop software for embedded systems are then shown in Section III. Based on this, selection criteria (Section A) are formulated and are used to identify a limited set of algorithms on which to perform a more detailed and targeted comparative evaluation. Section IV provides the results of the comparative evaluation conducted with real-life traffic traces and final conclusive remarks are provided in Section V.

II. THEORETICAL ANALYSIS OF CLASSIFICATION ALGORITHMS

Among the others [5], the comparative survey of classification algorithms by Gupta and McKeown [4] provides a detailed comparison of the most important known algorithms for multifield classification. Even though this work represents a complete and interesting tutorial on classification algorithms, it does not present any performance comparison based on real life network traffic. Our work leverages off some of the criteria and results presented by Gupta and McKeown to select a reduced set of classification algorithms that best fit to be implemented in embedded systems. Another contribution of our work lies in the detailed and homogeneous evaluation of such selected algorithms that have been implemented with common criteria and evaluated in a common test bed using real traffic captures.

A. Evaluation metrics and parameters

The metrics adopted are the ones commonly used by various authors [6][7][8][9][11][12] in literature, including Gupta and McKeown in [4]: search time, memory consumption, and update time.

Search time (T), i.e. the amount of time needed to classify a packet, is the most obvious metric; in order to devise a measurement (at least partially) independent from the particular test bed, the search time is measured in terms of CPU clock cycles.

Memory consumption (M) is the amount of memory needed to store the ruleset in some specific data structure in memory, computed either at instantiation or run time. Memory consumption is an excellent indicator of the compression capability of the algorithm measured as the ratio between the ruleset size (i.e. number of rules and number of fields) and its footprint in memory.

The *update time* (U) is the amount of time necessary to insert, delete, or modify a rule in the running ruleset.

An interesting metric is represented by the number of memory accesses performed by the algorithm, but it is not widely used because getting this data is far from being trivial.

The three metrics previously described generally depend on the following parameters:

- The number of rules *N* in the ruleset
- The number of fields *d* globally used within the *R[i]* components of each rule
- The length of each field, in bits, called *Wi*. In order to simplify the evaluation of the algorithms, we will use a new fictitious parameter *W*, defined as *W=max(Wi)*

Section A will provide some insight in the implications of such simplification on the comparative evaluation presented later.

B. Theoretical complexity of some well-known algorithms

In order to have a first general comparison of the classification algorithms and select which to adopt for a more thorough analysis, the theoretical worst-case bounds for the metrics identified in Section A were taken into consideration. Table 2 shows the formulas expressing the bound for each of the metrics. Such formulas were either taken directly from the literature, when available, or inferred from a paper describing the corresponding algorithm.

TABLE 2. WORST CASE BOUNDS FOR THE CONSIDERED METRICS

Algorithm	Search time (T)	Memory usage (M)	Update time (U)
Linear search	N	N	1
TRIES			
Hierarchical tries [4]	W^d	NdW	D^2W
Set pruning tries [11]	dW	N^d	N^d
Heap-on-Trie [6]	W^d	NW^d	W^dlogN
Binary search-on-Trie [6]	W^dlogN	NW^d	W^{d-1}logN
GEOMETRIC TECHNIQUES			
Cross producting [7]	dW	N^d	*N/A*
HEURISTICS			
Hierarchical Cuttings [9]	D	N^d	*N/A*
Tuple Space Search [8]	N	N	N
Recursive Flow Classification [12]	D	N^d	*N/A*

Hardware based [14] and ad-hoc algorithms [10] were not included in this evaluation since either the selected metrics cannot be applied to them, or a comparison based on them is meaningless due to the particular nature of such algorithms. Instead, the linear algorithm was included because it is widely used by software based firewalls (e.g. Linux netfilter/iptables [13]) and it is an excellent baseline against which other algorithms can be compared to, especially in the implementation and testing part of this work.

The bound on the update time is not shown for some of the algorithms since they do not explicitly support dynamic updates to the running ruleset. This stems from the fact that these algorithms preprocess the ruleset into a specific custom data structure that does not support insertion or removal of rules. Instead, in order to cope with ruleset changes the whole ruleset must be re-processed thus yielding a new data structure. Such an approach is usually inefficient, since the preprocessing time is typically quite high.

C. Practical issues with the theoretical complexity

The worst cases in Table 2 show quite clearly that the *linear* search algorithm outperforms the other algorithms in terms of memory consumption and update time. Its

search time performance is comparable to the other algorithms when the number of rules is not large; for example, when classifying UDP flows or TCP connections (d=5 and W=32) the break point is one or two hundreds rules. In fact, the search time of the other algorithms depends on the total number of bits dW of the various fields in each rule because the classification algorithm processes the classification fields bit by bit– in particular, this is the approach used by all the algorithms based on tries. Consequently, the linear algorithm might be particularly interesting in cases, IPv6 addresses, in which the total number of bits dW is high.

As a matter of fact, the theoretical analysis previously conducted is limited by several factors:

- The performance of many classification algorithms when used with real traffic might be very different from the theoretical results shown in Table 2; this is particularly true for heuristics, that are engineered to achieve good performances in the average case, and not in the worst case.
- The theoretical complexities shown in Table 2 have been devised assuming that all fields used for the classification have the same length, equal to the length of the largest one; this simplification can bring to unrealistic theoretical results (e.g. in the case of IPv6 session identifiers, we consider the length of a TCP/UDP port to be 128 bit, and this is completely misleading). A solution to this problem could be to re-formulate each metric taken into consideration using the various fields' lengths Wi, but this out of the scope of this paper.

III. Implementation

An objective of this work is to identify and evaluate the packet classification algorithms that are more suitable for an implementation on resource constrained systems. When writing software for an embedded system, specific constraints have to be taken into account in order to grant good performances and flexibility in terms of code portability to different target platforms: hence, several aspects have been considered while implementing the above mentioned algorithms.

First of all, the main goal of our work was to write a code portable to different target platforms, independent from the processor and the operating system used. To accomplish this objective, we developed a software library made up of pure ANSI C, trying to avoid any use of OS/compiler support functions that could not be available on special purpose processors. The crucial point in generating portable code is to separate the coding of functional modules from the one related to the specific target environment. This can be achieved by defining some sort of API, which avoids the use of platform dependent functions directly inside the code. A second consideration is that the code should use static memory allocation, since a dynamic allocation infrastructure is not granted to be present on all the target platforms.

Another requirement is that the code should avoid the use of explicit pointers in the raw data structures containing the ruleset; in fact, sometimes the code creating and initializing the data structure and the code that classifies packets using this structure run either on different processors (e.g. network processors using multiple processing units) or within different address spaces (e.g. code running partially at kernel level and partially at user level on a general purpose PC). A commonly used solution to the problem is to make use of indirect addressing, using only displacement pointers in the data structure, and the base pointer outside it.

In a network embedded system we can distinguish among data-plane functions (related to packet processing functionalities, with high performance requirements) that usually run on specific processor engines and control-plane functions (for data structure initialization and configuration, usually with high memory requirements) that may run on a general purpose processor. Thus, one general issue is to modularize the code as deeply as possible, trying to separate the main algorithm functionalities, which may have high performance requirements, from the control and configuration functions that may run on a different processor.

A. Selecting the algorithms to be implemented

Given previous considerations and taking into account the practical issues enlightened in Section 2, we decided which algorithms to implement to meet our objectives.

1. We excluded *Cross-Producting* and *Set-Pruning Tries*, because their memory consumption grows as N^d, which is extremely critical even with rather low values of N and d (e.g. with N=100 rules and d=4 fields memory consumption is about 10^8). While *RFC* and *HiCuts* have the same worst case memory consumption, they are heuristic algorithms, therefore this value is not enough to get rid of them.
2. We excluded *Heap on Tries* and *Binary trees on Tries*, because their memory consumption and search time is proportional to W^d which is too large (e.g. this value is larger that 10^{10}, when the maximum field size W is 128 bits and the number of fields d is 5); moreover the paper presenting these algorithms does not give any hint about any working implementation of them. Although the *Hierarchical Tries* algorithm has the same search time as the two previous ones, it has not been excluded because of its excellent characteristics referred to memory consumption.
3. We excluded *HiCuts*, because this algorithm is patent pending.
4. *Tuple Space Search* was excluded essentially because it was decided that the comparative study would include a single heuristic algorithm and from the information we gathered in the literature the implementation details of RFC seemed clearer.

In summary, we decided to implement the *Linear* algorithm, to be used as a baseline for the comparison, the *Hierarchical Tries* algorithm (the only remaining non-heuristic algorithm after the screening described above), and the *Recursive Flow Classification* algorithm.

IV. Performance Evaluation

Although our implementation is targeted to both general and special purpose platforms, so far it has been validated through extensive tests only on a standard personal computer. We did not consider tests on special purpose platforms in the context of this work since it specifically aims at giving a homogeneous comparison between the implementation of various algorithms by measuring their performance in real-life working conditions. Moreover, the obtained experimental results are compared against

the theoretical worst-case results. However, tests on special purpose platforms will be carried out as a future work in an effort to evaluate the performance disparities on different platforms.

A. Testbed

The tests were conducted using a network trace taken from our university link to the Italian trans-university backbone. This trace has the following characteristics:

- duration: 6 hours
- total packets: 24 milions
- total bytes: 13 GBytes
- average traffic: 5 MBps, 1100 pps.

The implemented algorithms have been compiled with the Microsoft Visual C++ 6.0 SP 5 compiler. We used an Intel Pentium IV 2GHz workstation with 1GB RAM, running Microsoft Windows XP. The measurements were taken with the x86 assembler instruction RDTSC (Read TimeStamp Counter), which gives the number of CPU clock ticks from the machine bootstrap.

We used the ruleset running on the router connected to the same link on which we captured the network trace (the packets were captured immediately before the router classifier); this ruleset is formed of 349 rules, each rule working on these fields:

- source / destination IPv4 address
- Layer 4 protocol (TCP/UDP/ICMP/any)
- source / destination TCP/UDP port.

In order to evaluate the algorithms with rulesets of different size, we extrapolated some fictitious ruleset from the original one. These are the new rulesets we defined:

- 2 rulesets formed of 50 rules (rules 1-50 and 51-100 of the original ruleset)
- 2 rulesets formed of 100 rules (rules 1-100 and 101-200 of the original ruleset)
- 1 ruleset formed of 200 rules (rules 1-200 of the original ruleset).

B. Search time test results

This test aims at measuring the average packet classification time for the various rulesets; the results are shown in Table 3.

The results of this test show that the mean search time grows linearly with the number of rules in the case of the *linear* algorithm; in the case of the *Hierarchical Tries* algorithm, the search time seems to grow linearly, too, but the trend is much lower than the linear one. The *RFC* algorithm, instead, shows a mean search time that is independent on the number of rules in the ruleset.

By comparing the results in Table 3 and the worst cases in Table 3, we can note that:

- the *linear* algorithm performs worse than the other two algorithms in our tests, compared to the theoretical results;
- the *Hierarchical Tries* algorithm seems to be loosely dependent on the number of rules *N*, while its worst case is independent from this parameter. This behavior could be due to the fact that the number of recursive visits of the tries grows with the number of rules *N*.

TABLE 3. AVERAGE SEARCH TIME (IN CLOCK TICKS) FOR THE GIVEN RULESET

	Number of rules	Linear	HiTrie	RFC
Ruleset 1-50	50	2603	981	419
Ruleset 51-100	50	2170	560	422
Ruleset 1-100	100	4572	1014	416
Ruleset 101-200	100	4408	1141	420
Ruleset 1-200	200	8949	1276	428
Ruleset 1-349	349	17552	2032	437

C. Memory consumption test results

We measured the amount of memory needed to store the raw data structure containing the ruleset for each algorithm. The results of this test are shown in Table 4.

TABLE 4. MEMORY CONSUMPTION (IN BYTES) FOR THE GIVEN RULESETS

	Number of rules	Linear	HiTrie	RFC
Ruleset 1-50	50	2192	32708	1838596
Ruleset 51-100	50	2192	34028	1836964
Ruleset 1-100	100	4192	64588	1841668
Ruleset 101-200	100	4192	59428	1847796
Ruleset 1-200	200	8192	115068	1860148
Ruleset 1-349	349	14112	155048	6074748

The results of this test illustrate that:

- The *linear* algorithm consumes memory linearly with the number of rules, and the average memory occupation per rule is about 40 bytes.
- The *hierarchical tries* algorithm has a memory consumption that grows linearly with the number of rules, as the *linear* algorithm, but the consumption per rule is much higher, about 500 bytes per rule.
- The *RFC* algorithm shows an unusual behavior: memory occupation is roughly constant when there are up to 200 rules in the classifier – about 1.8 Mbytes; with the complete ruleset, made up of 349 rules, memory consumption reaches 6 Mbytes. This "explosion" can be due to the fact that the last rules contain a large number of "any" as IP source and destination values.

D. Preprocessing time test results

The last test attempts to measure the amount of time needed to process the various rulesets and create the internal data structures used by each classification algorithm. The results of this test are shown in Table 5.

TABLE 5. PROCESSING TIME FOR THE GIVEN RULESET

	Number of rules	Linear	HiTrie	RFC
Ruleset 1-50	50	10.6 μs	0.84 ms	455 ms
Ruleset 51-100	50	15.5 μs	0.87 ms	448 ms
Ruleset 1-100	100	16.7 μs	1.08 ms	857 ms
Ruleset 101-200	100	16.1 μs	1.61 ms	966 ms
Ruleset 1-200	200	25.5 μs	3.19 ms	2.91 s
Ruleset 1-349	349	43.5 μs	5.43 ms	1289 s

The outcome of this test shows that the trend is roughly linear in the number of rules for the *linear* and *Hierarchi-*

cal Tries algorithm; moreover the latter is about 100 times slower than the former one, but the overall time to process the original ruleset containing 349 rules seems to be acceptable (less than 10 ms on the test platform). The *RFC* algorithm shows instead a rather interesting behavior: the trend is roughly linear on the number of rules up to 200 rules, with a cost that is about three orders of magnitude more expensive than the *Hierarchical Tries* algorithm; when we compute the data structure with the entire ruleset of 349 rules, the preprocessing time literally explodes to about 20 minutes. This explosion is generally due to two main factors:

1. It is a heuristic algorithm, so each metric normally depends on the particular ruleset used for the test.
2. Some experiments on this algorithm have shown that this behavior is largely due to rules containing a large number of "any" values in their components.

V. Conclusions

A continuously growing number of network appliances are deploying packet classifiers to implement Quality of Service, security, traffic engineering functionalities. As a consequence, in the last years several authors have proposed novel algorithms to achieve better results in terms of classification time and memory consumption. Many works provided case studies of such algorithms applied to a large number of real-life rulesets and network traffic traces. However, a fair comparison with common criteria and test cases has not yet been provided. Our main contribution in this work is filling this gap, by providing a homogeneous evaluation of three classification algorithms that have been implemented following the same criteria.

Our tests have shown that the Recursive Flow Classification algorithm outperforms, as expected, the other two algorithms in terms of search time. In fact, its heuristics is able to effectively exploit the characteristics of the real-life rulesets considered. However, it is known that this algorithm does not support dynamic updates, and our tests have shown that its preprocessing time is unpredictable.

The Hierarchical Tries algorithm shows acceptable performance in terms of classification time, being less than one order of magnitude worse that RFC. Instead it features low memory consumption, outperforming RFC for more than one order of magnitude. In practice, we have shown that the Hierarchical Tries algorithm is preferable over RFC when memory consumption and preprocessing time are more critical than classification time alone.

Finally, our tests confirm that the linear algorithm, despite the worst classification time with large rulesets, is the one that assures the lowest memory consumption, the fastest preprocessing phase, and the most flexible support for dynamic updates.

VI. References

[1] S. Shenker R. Braden, and D. Clark, "Integrated Service in the Internet Architecture: An Overview," RFC 1633, Standards Track, July 1994.

[2] DiffServ Working Group, "Differentiated Services (diffserv)", http://www.ietf.org/html.charters/diffserv-charter.html

[3] K. Nichols, S. Blake, F. Baker, D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, Standards Track, Dec. 1998.

[4] P. Gupta, and N. McKeown, "Algorithms for Packet Classification", IEEE Network Special Issue, March/April 2001, vol. 15, no. 2, pp 24-32.

[5] C. Macian, and R. Finthammer, "An Evaluation of the Key Design Criteria to Achieve High Update Rates in Packet Classifiers", IEEE Network, November-December 2001.

[6] P. Gupta, and N. McKeown, "Dynamic Algorithms with Worst-Case Performance for Packet Classification", Proceedings of NETWORKING 2000, France, May 2000.

[7] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching", Proceedings of ACM SIGCOMM '98, September 1998.

[8] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification using Tuple Space Search", Proceedings of ACM SIGCOMM '99, September 1999.

[9] P. Gupta, and N. McKeown, "Classifying Packets with Hierarchical Intelligent Cuttings", IEEE Micro, January-February 2000.

[10] M. WaldVogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups", Proceedings of ACM SIGCOMM '97, September 1997.

[11] P. Tsuchiya, "A Search Algorithm for Table Entries with Non-Contiguous Wildcarding", unpublished paper.

[12] Pankaj Gupta, and Nick McKeown, "Packet Classification on Multiple Fields", Proceedings of ACM SIGCOMM '97, August 1997.

[13] Linux NetFilter/IpTables framework, available at http://www.netfilter.org.

[14] Sibercore Technologies, available at http://www.sibercore.com.