JA-trie: Entropy-Based Packet Classification

Gianni Antichi*, Christian Callegari†, Andrew W. Moore*, Stefano Giordano†, Enrico Anastasi†

*Computer Lab, University of Cambridge, United Kingdom
Email:{name.surname}@cl.cam.ac.uk

†Department of Information Engineering, University of Pisa, Italy
Email:{name.surname}@iet.unipi.it

***Abstract*—Any improvement in packet classification performance is crucial to ensure Internet functions continue to track the ever-increasing link capacities. Packet classification is the foundation of many Internet functions: from fundamental packet-forwarding to advanced features such as Quality of Service enforcement, monitoring and security functions. This work proposes a novel trie-based classification algorithm, named Jump-Ahead Trie (JA-trie), utilizing an entropy-based pre-processing phase and a novel approach to wildcard matching. Through extensive experimental tests, we demonstrate that our proposed algorithm is able to outperform a range of state-of-the-art classification algorithms.**

I. Introduction

The Internet has been characterized by a relentless growth and diversity in network capacities, user and host-count, and an equally staggering diversity of applications. Additionally, the emergence of new network applications have introduced significant challenges in Quality of Service (QoS) support for high-speed networks. Packet classification holds a key role in modern communication networks, not only to forward packets in routers but in order to provide security and QoS. Although many solutions have been proposed over the years, each approach has limitations. This has permitted a continuous stream of approaches to be realised, each improving performance and attending past limitations. Each approach tipically seeks an optimization (e.g., memory foot-print, inherent latency of lookups, and algorithmic complexity) permitting a range of approaches to coexist - their precise implementation and behaviour dictating the domain of use.

In this work, we propose a novel trie-based classification algorithm, named Jump-Ahead Trie (JA-trie). Our approach builds two core ideas: an entropy based pre-processing step is applied to the classification rule-set and a novel mechanism is used to incorporate wildcard entries. The entropy is strictly related to the number of nodes created in a tree-based data structure indeed. Using such a property we show that it is possible to reduce the memory requirements of the lookup data structure. Section III details the proposed classification algorithms, focusing initially on the construction of the JA-trie data structure and then discussing the entropy-based pre-processing phase.

We present the results of experimental evaluations conducted across a range of classification rule-sets. These results illustrate the effectiveness of our proposed approach and service to highlight our performance for both tree-depth and memory footprint. Along with an outline of the experiments, Section IV presents results for our algorithm and a comparison with various state-of-the-art algorithms.

We first put our work into the context of the field by describing the related work.

II. Related Work

Packet classification has been extensively studied over the past decade. Many different kind of approaches have been applied to this problem in order to meet the ever-increasing demands of new networking devices. To provide context to our approach, we outline a representative set of current algorithms from each of the *CAM-based*, *trie-based* and *hash-based* classes of classification.

Hardware classifiers traditionally used Content Addressable Memory (CAM) based techniques to provide key-based table lookups. Given an input key, a CAM compares this against all table entries in parallel; hence, a lookup effectively requires only one clock cycle. While binary CAMs perform well for exact match operations, the widespread use of CIDR (Classless Inter-Domain Routing) requires storing and searching entries with arbitrary prefix lengths. Hence, ternary CAMs have become a common hardware approach. With the ability to store an additional don't care state this enables them to provide single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density and power consumption. Therefore, solutions such as [1], [2] suffer the same problem.

To overcome these issues, many trie-based, and hash-based, solutions have been proposed. A few solutions try to leverage longest-prefix matching trie-based algorithms (which were conceived for lookup applications) to bi-dimensional matching involving several fields. Such solutions [3] are commonly used when rules are specified only over destination and source IP addresses. Other solutions leverage a geometric formalization of the classification problem [4]: as each classification rule can be thought as a range in the multi-dimensional space, classifying a packet means finding out which ranges the corresponding point belongs to. To this end, well-known results from the field of computational geometry can be used. Cohen & Lund [5] proposed to optimize decision trees by introducing the optimization of common branches. These common branches are

978-1-4799-1633-7/14/$31.00 ©2014 IEEE

rules that, due to wildcards, get assigned to both children of a decision node; separate-handling reduces worst-case size. A speed-up is proposed by [6] by using a small cache using a set of evolving rules, which preserve classification semantics. Finally, [7] achieves improvements by partitioning the rules into sets, which are close to one another in the tuple space. Thereby leveraging information from single-field lookups to discard subsets and limit the search space. A further class of algorithms leverage decision trees: although, formally, the algorithm model is analogous to the trie-based approaches, it allows for larger flexibility. Instead of having all of the relevant fields inspected in a sequential manner, each node can perform an arbitrary check. In particular, Hicuts [8], performs a range check on a particular field, while [9] tests single bits. Hypercuts [10] further improves performance by checking multiple fields at each step. Finally, Efficuts [11] eliminates overlap among small and large rules and achieves fewer access per node.

As for hash-based approaches, the Tuple Space Search (TSS) algorithm [12] currently remains widely respected. The scheme is motivated by the observation that while filter databases contain many different prefixes or ranges, the number of distinct prefix lengths tends to be small. Thus, the number of distinct combinations of prefix lengths is also small.

It is clear that the opportunities for performance improvements are significant.

III. An Entropy–Based Packet Classification Scheme

The growth that is a hallmark of Internet networking applies a specific pressure to network-devices; devices must keep-pace with this demand. This has led to increasingly optimised and specialised algorithms, resulting in the packet classification problem being still an open and on-going issue. The common approach to packet classification is to create a tree-based data structure starting from a given rule-set. Every time a new packet arrives, the 5-tuple (*i.e.*, IP source/destination addresses, protocol, and layer 4 ports) is extracted and the lookup process on the data structure starts. Accelerating such a process can be done by either compressing the data structure (thus allowing to use smaller and faster memories) or by reducing the number of memory accesses.

To this end, we have developed a new classification structure: Jump-Ahead Trie (JA-trie) that, along with entropy-based optimisation (described in § III-B), is able to reduce the memory-size footprint while maintaining a very small tree-depth, leading to fast key lookup times.

The remainder of this section outlines the classification tree (§III-A), covering the algorithmic processes for forming the Trie structure (§III-A1) and the mechanism for performing lookups within the JA-Trie (§III-A2). The Entropy-based preprocessing phase is outlined in the final sub-section (§III-B).

A. JA-Trie: Jump-ahead Trie data structure

A prefix-tree (*i.e.*, trie) is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. All the descendants of a node share a common prefix of the string associated with that node, with the tree root associated with the empty string. Values are normally not associated to every node of the tree, but just to the leaves and to some of the inner nodes that correspond to keys of interest.

While unibit tries are traversed using a bit at a time, (an approach that offers excellent memory-usage performance but terrible memory-lookup overheads; proportional in lookups to the tree-depth), the multi-bit tries are traversed using b-bits (*i.e.*, a stride) at a time. This lowers the required number of memory accesses per lookup, but at the cost of an increase in memory footprint. For this reason we propose an improved multi-bit trie, based in part on the "classical" 8-bit multibit-trie [13][§11.5 *Multibit Tries*].

Our proposal is significantly different from the "classical" 8-bit multibit-trie in our incorporation of wildcard entries. Indeed, it is well-understood that wildcards have a significant performance impact on a tree-based data structure because they force the creation of a different node for each of the possible values of the wildcard. This, in-turn, significantly increases the memory requirements when incorporating wildcards into the "classical" 8-bit multibit-trie. Moreover, wildcards do not usefully discern different rules which leads to an increase in the required tree-depth leading to a slower lookup process. The drawbacks of incorporating wildcards into "classical" 8-bit multibit-trie provide the motivation for our approach. Jump-Ahead Trie, unlike the "classical" 8-bit multibit-trie, does not directly incorporate wildcard strides. Instead, lookups jump from one fixed stride to the next.

Let us analyse in detail how this has been achieved. Firstly, let us consider two different bitmaps that have been inserted to enable the "jump-ahead" feature:

- `transition bitmap`: every transition has a k-bit bitmap associated, where k is the number of strides composing the string to match. The j^{th}-bit of the bitmap is asserted if the transition represents the j^{th} stride of the string to match. It is important to note that a transition can represent more than one stride at the same time.
- `rule bitmap`: every rule stored in a node also has a k-bit bitmap, in which the j^{th}-bit is asserted if the rule is referring to the j^{th} chunk of the tuple.

In the subsequent sections we will describe both the trie construction and lookup processes in order to better understand the JA-trie properties.

1) Trie construction: Algorithm 1 shows the pseudocode for the trie construction process. Firstly, we note that the construction-process requires strides to be composed of either fixed values or wildcards (rule expansion over strides is needed when such properties are not guaranteed in the actual rule-set).

During the trie construction, only the strides with fixed values produce a new node in the data structure, while strides that are wildcards do not create any new node because they are merged with existing nodes by means of transition bitmaps. The reduction in the number of transitions and inner-nodes,

Algorithm 1 Pseudo-code for trie construction. n is the current node, k is the number of strides, $S(j)$ is the j^{th} stride of a rule

```
1:  procedure CONSTRUCTION(rule)
2:      create ROOT
3:      read(rule)
4:      while rule do
5:          n = ROOT
6:          for j = 0 → k − 1 do
7:              q = S(j)
8:              if q ≠ ∗ then
9:                  if n.next(q) ≠ NULL then
10:                     n.TransitionBitmap[j] ← 1
11:                     n = n.next(q)
12:                 else
13:                     create NEWNODE
14:                     n.next(q) = NEWNODE
15:                     n.TransitionBitmap[j] ← 1
16:                 end if
17:                 n.RuleBitmap[j] = 1
18:             end if
19:         end for
20:         n.Store(rule, RuleBitmap)
21:     end while
22: end procedure
```

which results from wildcard-strides, leads to a reduction in the overall memory footprint of the data structure.

Rule	Stride 1	Stride 2	Stride 3	Stride 4
R1	∗	10	∗	7
R2	10	∗	2	6
R3	10	2	6	4
R4	10	∗	7	∗

TABLE I
RULE SET EXAMPLE

To illustrate such a concept, let us refer to the rule-set proposed in Table I, where wildcards are indicated by the symbol $*$, and to the resulting JA-trie shown in Figure 1. As can be clearly seen the root node has only one child node, corresponding to the transition that is obvious for rules R2, R3, and R4, while it does not have any transition for the rule R1, where we have a wildcard in the first stride. This implies that during the lookup process we move from the root node to the child node, when considering R2, R3, or R4, while we do not have any transition when considering R1.

Now, let us consider the transitions due to the second stride. In our case R1 would generate a transition that is already present in the data structure, R2 and R4 do not generate any transition (because they have a wildcard in the corresponding stride), and R3 generates a transition toward a child node, corresponding to the value 2. It is clear that in this way, the same node can refer to different strides of different rules. Considering the current example, the transition from the root node toward the only child node can refer either to the first stride of rules R2, R3, and R4, or to the second stride of R1. To solve such an ambiguity every child node has to maintain a bitmap, *transition bitmap* illustrated in Figure 1 alongside each transition in the figure. This is used in the lookup process to identify which strides must be considered.

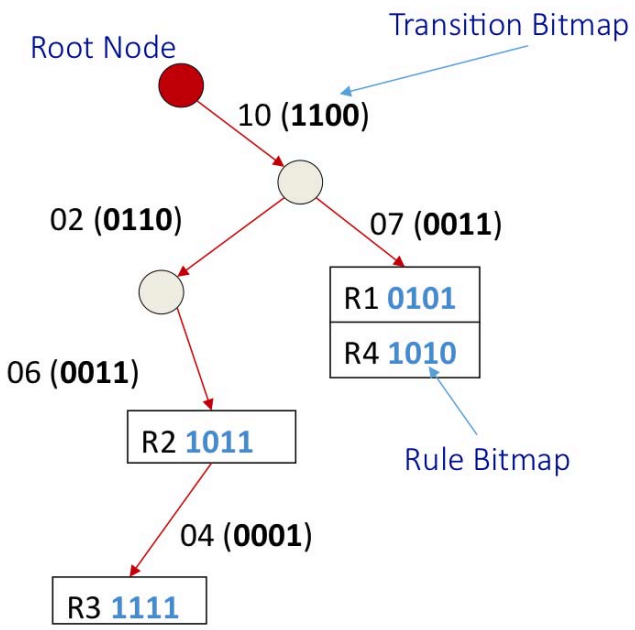

Fig. 1. JA-Trie example

In our example, the transition bitmap associated to the "first" child node will have the first and second bit set, to indicate that the transition can be related either to the first or second stride.

This approach introduces some false positives; more than one stride can be represented by the same node, thus ambiguity occurs when we reach a tree leaf corresponding to one or more rules. We cannot know which transitions we have actually done.

For this reason, each rule has an associated bitmap. The *rule bitmap* associated with each rule is illustrated in Figure 1 as a table in each transition. The bitmap highlights which strides have been used to obtain such a result; the lookup process will consider the rule bitmap to see if it matches the followed transitions, thus selecting the correct rule.

In our example the leaf node with transition 7 has two rule bitmaps, corresponding to R1 and R4.

2) Lookup: Algorithm 2 shows the pseudocode for the lookup process. Jumping from one node to another is allowed only if there is a valid transition for a given stride $S(i)$. Since more than one stride can be represented by the same node it is not obvious to have a valid transition for a matching stride $S(i)$, beacuse it could be referred to another one ($S(j)$). The transition bitmaps are used just to avoid possible wrong paths.

During the lookup process a temporary bitmap (*i.e.*, BM_{class}) must be updated in order to keep track of the considered strides.

Every time a new node is reached, all possible strides (aside from the ones already traversed) must be taken into account to find a valid transition. Once a transition is found, the temporary bitmap must be updated by asserting the bit referred to the stride just used. If no valid transitions are present, the

Algorithm 2 Pseudo-code for the lookup in a JA-Trie. n is the current node, k is the number of strides and $S(i)$ is the ith stride of the input string

```
1:  procedure LOOKUP(S)
2:      n = ROOT
3:      for i = 0 → k − 1 do
4:          BM_class[i] ← 0
5:      end for
6:      for i = 0 → k − 1 do
7:          q ← S(i)
8:          if n.next(q) ≠ NULL then n1 = n.next(q)
9:              if n1.TransitionBitmap[i] = 1 then
10:                 BM_class[i] ← 1
11:                 n ← n.next(q)
12:             end if
13:         end if
14:     end for
15:     if BM_class == RuleBitmap then
16:         select(rule)
17:     end if
18: end procedure
```

process ends. Upon reaching a node that stores one or more rules; if it has just one rule, the lookup result is obvious, otherwise, if more rules are stored, the process must compare the temporary bitmap with the rule bitmaps in order to find the correct match.

B. Entropy-Based JA-Trie

In information theory, entropy is a measure of the uncertainty in a random variable [14]. The term usually refers to the Shannon entropy, which quantifies the expected value of information contained in a message. Formally, given a random variable X, which can assume n possible distinct values $\{x_1, x_2, \ldots, x_n\}$, it is defined as:

$$H(X) = \sum_{i=1}^{n} P(x_i) \times log_2 \frac{1}{P(x_i)} \qquad (1)$$

where $P(x_i)$ is the probability mass function of outcome x_i.

In a nutshell, this means that an event with high probability has low entropy and vice-versa. An analogous consideration can also be applied to the strides; low value of the entropy of a stride means that many strides from different rules share the same value. Therefore, the entropy is strictly related to the number of nodes created in a tree-based data structure. Exploiting such a property makes possible to reduce the memory requirements of the data structure. JA-Trie can benefit from this observation, by employing two additional phases, namely a pre-processing phase and an entropy-based byte organisation phase.

The pre-processing phase takes care of dividing the rules into 8-bit strides and calculates the entropy value over each of them. We note that such a process needs strides composed of either fixed values or wildcards. Since wildcard strides do not produce any child node in the trie construction process, they have zero entropy value.

Once the entropy values have been computed, the rules are reorganised. In practice the different strides of the rules are re-ordered based on increasing value of the entropy; the rules are written so as that they begin with the stride with the lowest entropy values, followed by the stride with the second lowest entropy value and so on.

In contrast to a trie constructed without this pre-processing phase, this phase reduces the number of nodes in the first levels and enlarge the number of leaf nodes, leading to a smaller memory footprint. The memory-size of the structure can be sufficiently small to fit in cache, particularly for new-generation of large-cache CPUs now entering the market [15], thus potentially improving the overall lookup speed.

R1	64.91.107.0/32
R2	95.105.142.0/32
R3	96.105.142.0/32
R4	96.10.142.0/32

TABLE II
RULE SET EXAMPLE

Rule	Stride 1	Stride 2	Stride 3	Stride 4
R1	64	91	107	0
R2	95	105	142	0
R3	96	105	142	0
R4	96	10	142	0
Entropy	1.5	1.5	0.81	0

TABLE III
RULE SET EXAMPLE, WITH STRIDES ENTROPY VALUES

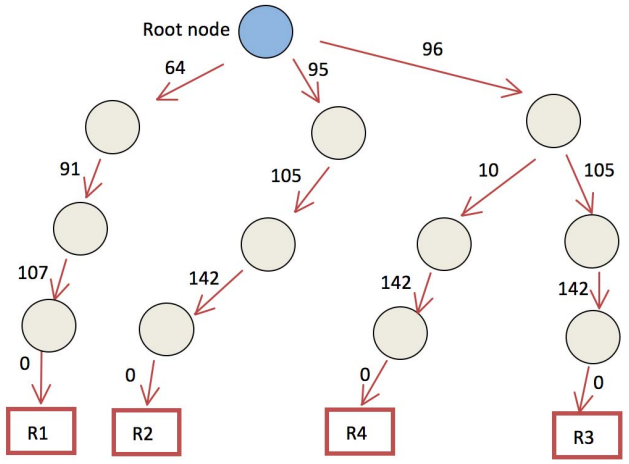

Fig. 2. Example of standard JA-Trie

To better quantify the effectiveness of the preliminary phase, let us take into account the rule set shown in Table II. In this example there are four simple rules each based on a single IP address. Splitting the rules into different 8-bit long strides, we get Table III, containing the values of the entropy associated to each of the single strides in the last row. The corresponding JA-trie is shown in Figure 2.

Alternatively, applying the pre-processing phase, the processed rule set is re-ordered on the basis of the strides entropy value. This leads to the rule-set shown in Table IV and to the related Entropy-Based-JA-Trie, shown in Figure 3.

Rule	Stride 1	Stride 2	Stride 3	Stride 4
R1	0	107	64	91
R2	0	142	95	105
R3	0	142	96	105
R4	0	142	96	10

TABLE IV
ORDERED RULE SET EXAMPLE

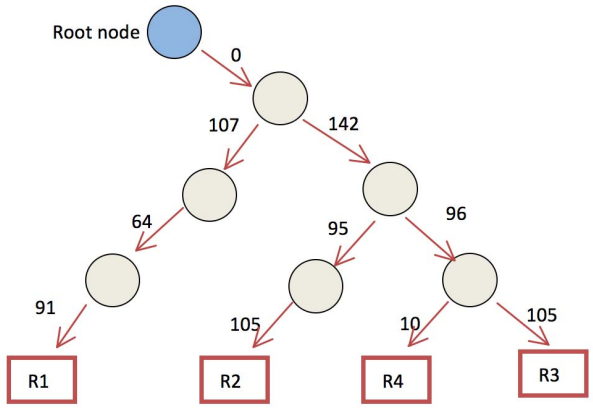

Fig. 3. Example of Entroby-Based-JA-Trie

Even in this very simple case the Entropy-Based-JA-Trie is more compact than the "standard" JA-Trie; presenting only 10 nodes instead of 15. It is worth highlighting that, given the construction procedure the number of nodes of the Entropy-Based-JA-Trie is always less than or equal-to the number of nodes of the corresponding JA-Trie.

IV. SIMULATION RESULTS

In this section we present the results of an experimental evaluation of the effectiveness of the proposed algorithm (note that here and in the following, with the expression our algorithm, if not differently specified, we always refer to the Entropy-Based-JA-Trie).

Using Classbench [16] we have run several experimental tests, varying the number of rules in the dataset (i.e., ranging from 500 to 10,000 rules). For each dimension of the rule-set we have generated 1,000 different rule-sets. Each plot shows the mean value over all the rule-set tests for a particular parameter. Moreover, to correctly evaluate the performance of our method we have compared it with a number of the most popular classification algorithms: WOO, TSS, HiCuts, and HyperCuts [9], [12], [8], [10]. We chose the best possible configuration for each algorithm, for example, both the best bin^{th} and space factor in the case of HiCuts.

In common with the past-practice of other papers using toolkit [8], [10] we do not provide lookup speed results as this depends on the hardware used for the implementation (e.g., sizing and use of fast and slow memories). Furthermore, the majority of the comparative algorithms do not provide information about sample implementations, rendering a direct comparison to be error-prone at best while, at-worst, liable

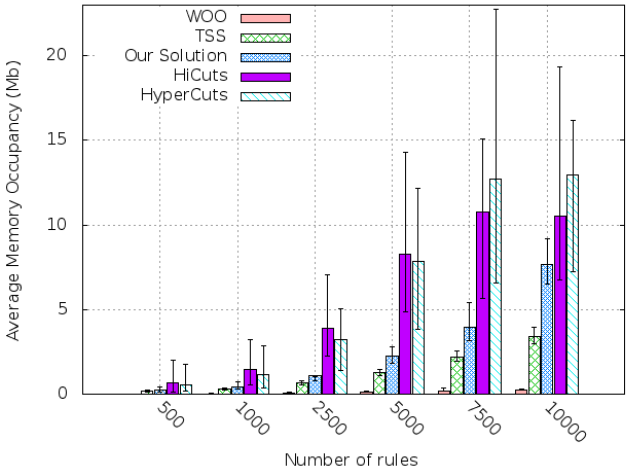

Fig. 4. Comparison of Memory Occupancy between chosen algorithms for different datasets

to misrepresent the work of others. Instead of direct lookup-speed results, we use the tree-depth as a proxy representation of lookup-speed. This is because, in general, the lookup speed is closely proportional to tree-depth. This consideration allows us to conclude that our algorithm has the potential to offer good lookup-speed performance.

Figure 4 shows the memory occupancy of each algorithm. In this case, the proposed algorithm clearly outperforms both HiCuts and HyperCuts, but behaves slightly worse than WOO and TSS.

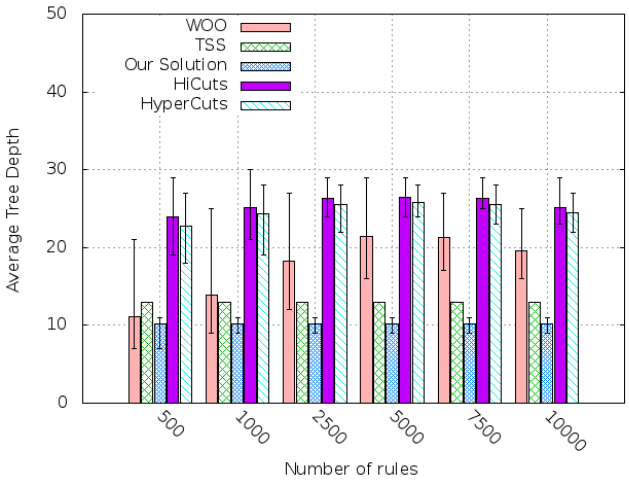

Fig. 5. Comparison of Tree Depth between chosen algorithms for different datasets

The tree depth is illustrated in Figure 5. It can be clearly seen that the JA-trie method presents a near-constant behaviour with the best performance over the comparison-set of algorithms. Results are almost comparable to those offered by either WOO or TSS for "small" rule-sets, but our algorithm behaves considerably better when increasing the rule-set size is increased.

While there are distinct algorithms capable of either the best memory-footprint, or the best performance from the tree with the least depth, aside from JA-trie, no algorithm can provide a balance. The JA-trie algorithm is able to offer good performance while considering both performance criteria and thus this results in a good choice for fast packet classification.

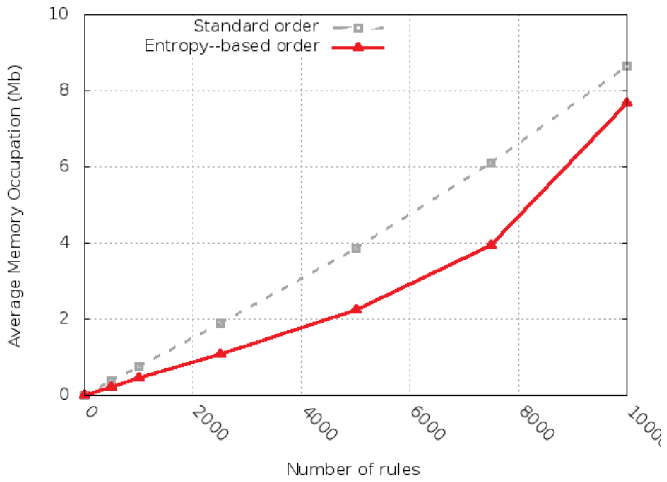

Fig. 6. JA vs Entropy-based JA

Finally, Figure 6 shows the effectiveness of the entropy-based pre-filtering phase on the memory occupancy. It is clear that entropy can have a significant effect and its plausible that such pre-filtering might improve the depth of a number of other classification algorithms if applied appropriately.

V. Conclusions

In this work we introduced a novel classification algorithm: the Entropy-based JA-trie. This approach utilizes an entropy-based pre-processing phase along with a novel mechanism for incorporating wildcards. An experimental evaluation illustrates the effectiveness of the proposed solution for different data-sets. Indeed, the algorithm is able to offer good performance both in terms of memory occupation and tree depth. The extensive comparison with state-of-the-art algorithms has demonstrated that our proposal is able to overcome a significant limitation of past algorithms able only to optimise a single performance metric at a time.

Acknowledgement

This work was jointly supported by the EPSRC INTERNET Project EP/H040536/1, by the National Science Foundation under Grant No. CNS-0855268, and by the MIUR project GreenNet (FIRB 2010).

References

[1] V. Ravikumar and R. Mahapatra, "Tcam architecture for ip lookup using prefix properties.," in *Micro*, IEEE, 2004.

[2] B. Vamanan and T. Vijaykumar, "Treecam: Decoupling updates and lookups in packet classification," in *Conference on Emerging Networking EXperiments and Technologies (CONEXT)*, ACM, 2011.

[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: A software architecture for next generation routers," in *Transactions on Networking*, pp. 229–240, ACM/IEEE, 1998.

[4] H. Adisheshu, "Services for next-generation routers," in *Ph.D. dissertation*, 1998.

[5] E. Cohen and C. Lund, "Packet classification in large isps: design and evaluation of decision tree classifiers.," in *SIGMETRICS*, ACM, 2005.

[6] 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.," in *SIGMETRICS*, ACM, 2007.

[7] H. Song, J. S. Turner, and S. Dharmapurikar, "Packet classification using coarse-grained tuple spaces," in *Symposium on Architectures for Networking and Communications Systems*, ACM/IEEE, 2006.

[8] P. Gupta, , P. Gupta, and N. Mckeown, "Packet classification using hierarchical intelligent cuttings," in *in Hot Interconnects VII*, pp. 34–41, 1999.

[9] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *In IEEE Infocom*, pp. 1213–1222, 2000.

[10] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *In Proceedings of ACM SIGCOMM*, pp. 213–224, 2003.

[11] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput.," in *SIGCOMM*, ACM, 2010.

[12] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *In Proc. of SIGCOMM*, pp. 135–146, 1999.

[13] G. Varghese, "Network algorithms: An interdisciplinary approach to designing fast networked devices," *Morgan Kaufmann Series in Networking*, p. 245, 2004.

[14] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[15] S. Chennupaty, P. Hammarlund, and S. Jourdan, "Intel 4th generation core processor (haswell)," in *Hot Chips: A Symposium on High Performance Chips*, Aug. 2013.

[16] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE INFOCOM*, 2004.