Efficient packet classification using order-independent decompositions
Project Milestone Report

Márton Zubor*

Contents

*<zuborm@gmail.com>

1 Abstract

Packet classification is concerned with categorizing packets into traffic aggregates according to some configurable set of rules and apply some collective action to the aggregates' packets. It is a fundamental ingredient in essentially any network functionality today, from the elementary, like Internet packet forwarding, to the complex, like firewalls, OpenFlow switches, or load-balancers. Packet classification must be done for all packets received by a device, preferably at line rate, and, what is more, over possibly hundreds of thousands of rules. This makes efficient hardware and software implementations extremely difficult, or may in fact inhibit any sorts of fast implementation whatsoever.

In this project, we have studied the question of how to boost software-based packet classification performance by exploiting data level parallelism built into every modern CPU. We have proposed a new classifier decomposition scheme, whereas a huge and complex classifier is disassembled into a set of smaller and simpler classifier modules, substantially curtailing the complexity of rule matching by reducing the lookup semantics at each module to quasi-exact matching and permitting to do lookups across multiple modules simultaneously.

The report contains the results of the first evaluation of the proposed scheme and an initial comparison to the Intel DPDK built-in `librte_acl` classifier library. Our initial results are promising: we provide empirical evidence that even very large classifiers can effectively be decomposed into only a couple of dozen modules (which, again, can be searched in parallel) and we present preliminary runtime performance analysis to show that this may yield improved classification performance when implemented in a real classifier.

2 Motivation

- **Goal:** exploit data level parallelism for fast and memory-efficient software packet classification
- main motivation is that today's CPUs come with advanced SIMD instruction sets off-the-shelf and packet classification, like most common network functions, lends itself readily to a data-parallel implementation:
  - execute same classification program simultaneously (SI: "Single Instruction...")
  - on multiple packets (MD: "Multiple Data")
  - SI + MD = SIMD !

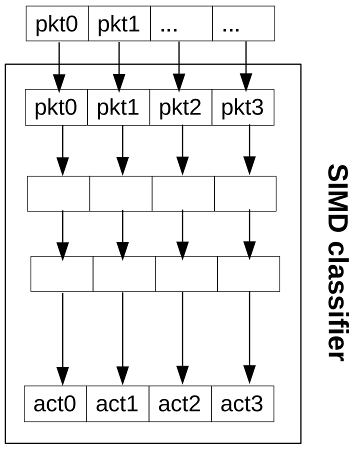

Figure 1: Packet classification using data level parallelism

2.1 General definition and notation

- a classifier K is a prioritized list of n *rules* (filter-action pairs), each defined on w bits:

$$K = \{R_i \to a_i : i \in [1, n]\},$$

  where $R_i \in \{0, 1, *\}^w$ are *filters* and $a_i \in [0, A-1]$, the *actions*, are identified by integers
- we use the terms "rule" and "filter" interchangeably
- each rule R_i is represented by a subset of $\{0, 1, *\}^w$ and some word r *matches* rule R_i, that is, $r \in R_i$, if and only if $\forall j \in \{1, 2, \ldots, w\} : R_{ij} \neq * \implies r_j = R_{ij}$
- we assume priorities are aligned with indices: $R_i \prec R_j \Leftrightarrow i < j$

3 Disjoint classifier decompositions

3.1 A Naive SIMD Implementation: Linear Search

- a SIMD register can contain up to 256 (in AVX2) or 512 (in AVX-512) bits
- a naive strategy to exploit this would be to
  1. load packet header into a SIMD register
  2. load next rule into another SIMD register
  3. do the matching on *all the bits of the header against the rule simultaneously* (using SIMD)
  4. do this iteratively until either a match is found or all rules have been processed this way
- a possible implementation using AVX2:

```
// cycle through all rules
for(unsigned int i = 0; i < n; i++){
        // load packet into SIMD register
        __m256i packet =_mm256_load_si256(packet);

        // load rule into SIMD register part 1: positions of don’t care bits
        __m256i rstar =_mm256_load_si256(rules[i].star);

        // load rule into SIMD register part 2: rest of the mask bits (0 or 1)
        __m256i rbit =_mm256_load_si256(rules[i].bit);

        // matching: mask with don’t care bits
        __m256i masked =_mm256_and_ps(packet, rstar);

        // matching: compare the rest
        __m256i comp =_mm256_cmpeq_epi32(masked, rbit);

        // evaluate results
        int result =_mm256_movemask_ps(comp);
        if(comp == 255){
                // rule i matches
                ...
                break;
        }
}
// no match found
```

- the linear search strategy is of course sensitive to the number of rules n: needs $O(n)$ time (the even more naive full scalar implementation would need $O(nw)$ steps, so we spared a w factor with SIMD)

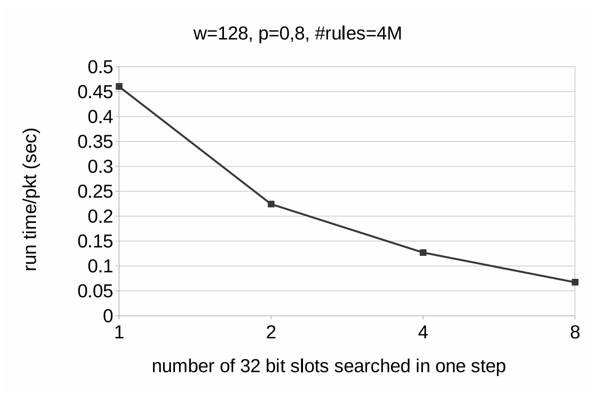

Figure 2: Linear Search: The more bits we consider the faster the linear search, but still, iteratively matching a single packet against 4M rules takes 50 milliseconds

3.2 Improving the naive implementation

- the problem with the naive implementation is that it looks at too many bits, that is, rules/headers are too wide for us to squeeze multiple rules and headers into a single SIMD register and parallelize across rules/packets
- our idea (to be fair, the idea belongs to Kirill Kogan, our collaborator from IMDEA Networks, Spain) is to look at only a smaller subset of rule/header bits at a time, say, only 8 or 16 bits
- either we can establish that there is no match on these bits for the current rule, in which case we can carry on with matching the rest of the rules, or we have a candidate match
- then, do a false positive check on candidate matches to verify that they indeed match (not just on the subset of bits considered but on *all* bits) and then find the maximum priority candidate and return it as a match
- now, since we consider only a smaller number of bits in each step we can fit multiple rules/headers into a single SIMD register and do matching simultaneously

3.3 Disjoint decompositions

- our decompositions are special since we do not want modules to return multiple candidate matches (which will then all need to be false-positive-checked), just a single one
- it is easy to see that the following property, called disjointness (or order-independence), is the simplest criterion that warrants this
- **Disjointness:** rules R_1 and R_2 are (filter-)disjoint, if there is no packet/header $p \in \{0,1\}^w$ so that $p \in R_1$ and $p \in R_2$
- a classifier K is disjoint of all rules in K are pairwise disjoint

3.4 Problem formulation

- of course we want to look at the smallest number of relevant bits (to maximize SIMD parallelism) and we want to get the fewest possible modules (each module incurs additional runtime cost), which gives rise to a nice optimization problem

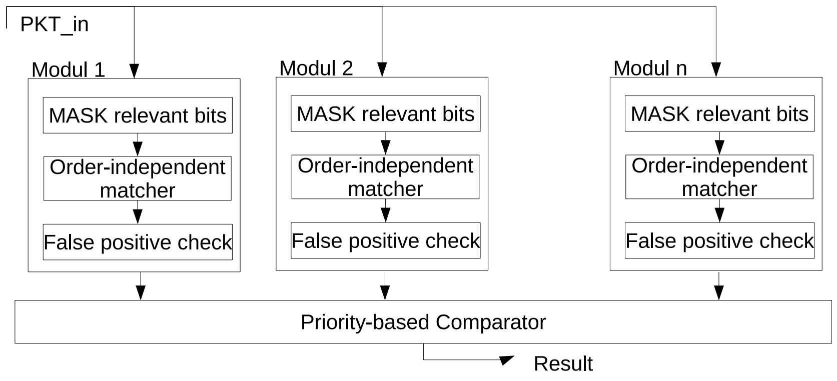

Figure 3: Disjoint Classifier Decomposition

- **Classifier reduction:** given classifier $K(n, w)$ and bit positions $B \subseteq [1, w]^b$ with $b \leq w$, the reduction K^B of K to bit positions B is defined as $K^B = \{[R_{ij}] \rightarrow a_i : i \in [1, n], j \in B\}$
- **Classifier partitioning:** given classifier $K(n, w) = \{R_i \rightarrow a_i : i \in [1, n]\}$ and some partition of the index set $\mathcal{C} = \{C_1, \ldots, C_k\}$ where $\bigcup C_i = [1, n]$ and $C_i \cap C_j = \emptyset$ whenever $i \neq j$, the classifier partition $K_{\mathcal{C}}$ is given by exactly k classifier groups on $\mathcal{C}$: $\mathcal{K}_{\mathcal{C}} = \{\mathcal{K}_{C_j} = \{R_i \rightarrow a_i : i \in C_j\}, j \in [1, k]\}$
- **Minimal Disjoint Classifier Decomposition:** given a classifier $K(n, w)$, integers b and k, and bit positions $\mathcal{B} = \{B_j \subseteq [1, w]^b, j \in [1, k]\}$, decide whether a classifier partition $\mathcal{K}_{\mathcal{C}}^{\mathcal{B}}$ exists so that each $K_{C_j}^{B_j}$ is disjoint
- we consider only the version where an oracle fixes the bit positions for each module B_j; of course the better we choose bit positions the smaller decomposition we might get; however, the full-fledged version of the problem (that includes picking B_j bit positions) is even more difficult
- the role or order-independence in classifier decompositions is first explored in [1], minimal disjoint decompositions are also introduced there
- we have already tackled the problem to some degree in [2], namely, we explored minimal decompositions for classifiers that contain prefix rules only; herein we generalize those findings to general classifiers and give some new analyses

4 Algorithms

- since the problem is very difficult, we start with a simple greedy heuristics which we shall later endow with useful enhancements to improve efficiency

4.1 A fast heuristics

- **Greedy partition:** given $K(n, w)$ and b, iteratively
  1. draw some random bit positions $B \subseteq [1, w]^b$
  2. move all rules that are disjoint on B to a new module/subclassifier
  3. repeat until all rules are removed from K
- complexity is $O(n^2 b)$
- **Observation:** on termination, no rules can be placed into modules found before the rule's own module (weak optimality)

4.2 Extensions

- **Bit-selection strategies:** how to select the bit positions for each module
  - choose the same fixed bit positions for each module: $\forall j : B_j = \{0, 1, ..., b-1\}$ (name: uniform)
  - choose bit positions randomly (name: random)
  - choose consecutive bit positions (with wrap-around if $kb > w$): $B_1 = \{1, ..., b\}$, $B_2 = \{b+1, ..., 2b\}$, $\ldots$, $B_i = \{w - b + 1 ... w\}$, $B_{i+1} = \{1, ..., b\}$, $\ldots$ (name: next-bits)
  - choose a random number $0 \le k < \frac{w}{b}$ then the selected bits are $kb+1, kb+2, ..., (k+1) * b$ (name: semi-random)
  - a version of the above when bit positions are aligned at byte-boundaries, which will make masking simpler in the implementation (name:semi-random2)
- **Preprocessing:** given B for a module, order the rules in increasing order of the number of * bits in the positions defined by B (will take rules with fewer *s first, improving utilization of the module) (name: order)
- **Greedy partition with star splitting:** after doing the greedy partitioning algorithm, try to move rules from rearward modules to front ones by taking a rule in the last group, split a random *, try to add the resultant 2 rules to some former group, and do until no rules can be placed to former groups
- **Quasi-independence:** generalize the notion of rule disjointness, namely, in each group allow at most k rules to match for each possible packet header
- of course, that will induce multiple false positive checks but, hopefully, less modules
- so traditional disjointness is \$1\$-independence, \$2\$-independence is when each packet matches at most 2 rules in a group, etc
- **k-independent greedy partitioning algorithm:** take a "normal" greedy partition and then merge some k buckets defined on the same bit positions (result trivially k-independent)

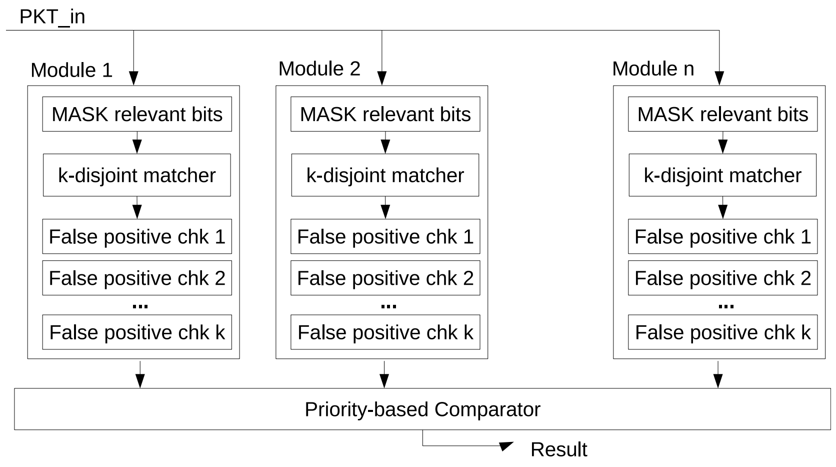

Figure 4: Quasi-disjoint Classifier Decomposition

4.3 Results

- we implemented the algorithms to see how the default heuristics and the above extensions fare
- input is always a random classifier (usually containing 10K rules); this allows to fine-tune model parameters (mostly p) but distorts the results significantly; all results will be re-checked on ClassBench classifiers

4.3.1 Greedy classifier partitioning: the effects of heuristics

- we chose $b = 8$ and $b = 16$, which both allow byte-aligned masking and disjoint rule matchers to be implemented with simple arrays of reasonable size

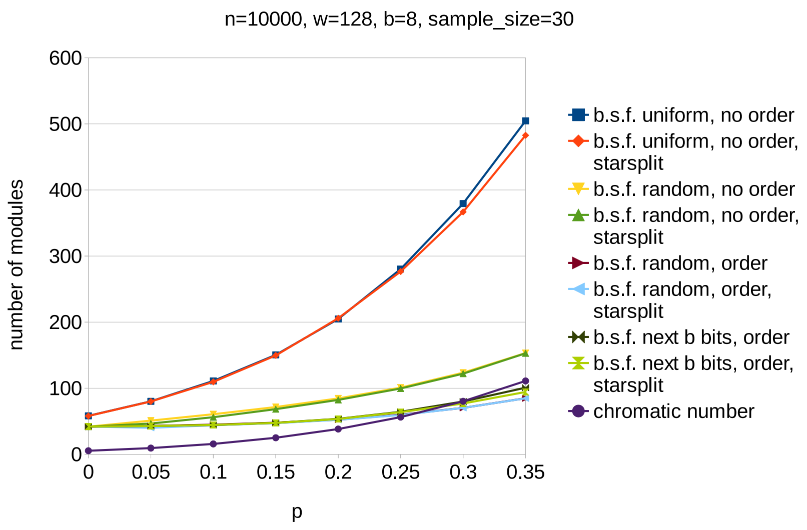

Figure 5: Disjoint Classifier Decomposition with heuristics when $b = 8$: $n = 10000$, $w = 128$, x axis: $p \in [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35]$, y axis: number of modules with the plain greedy algorithm plus with different extensions

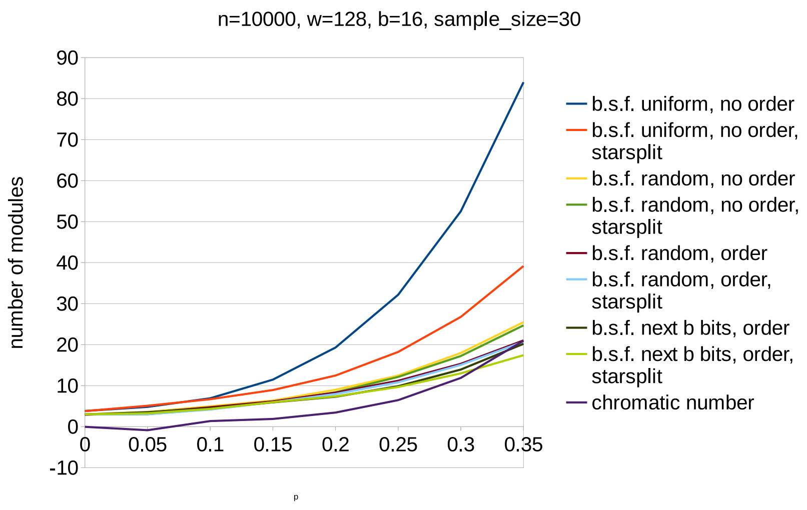

Figure 6: Disjoint Classifier Decomposition with heuristics when $b = 16$: $n = 10000$, $w = 128$, x axis: $p \in [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35]$, y axis: number of modules with the plain greedy algorithm plus with different extensions, note that the negative values are due to the error of the estimation for small p

- messages: See Fig. 5 and Fig. 6
  - we can decompose a random classifier into roughly 100 modules when $b = 8$ and only about 20 for $b = 16$
  - the greedy algorithm seems highly effective even with absolutely blind rule selection!!
  - the heuristics improve significantly, except star-splitting

4.3.2 Greedy classifier partitioning: distribution of number of rules in each module

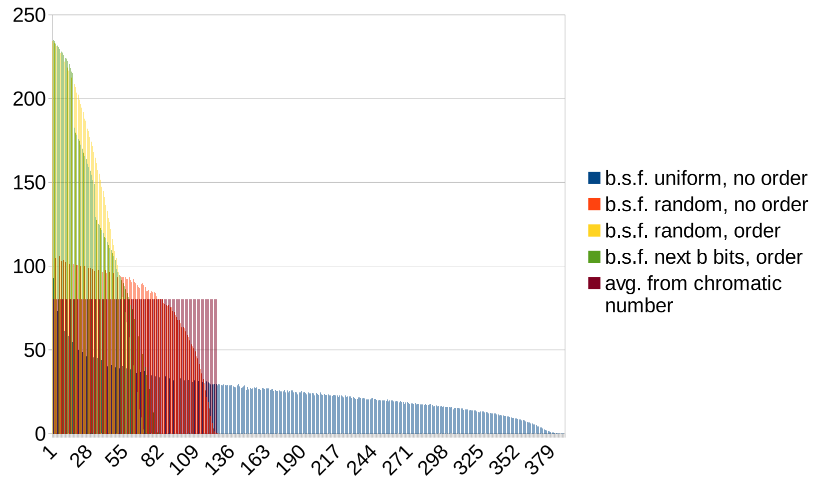

Figure 7: Disjoint Classifier Decomposition and the distribution of rules per module: $n = 10000$, $w = 128$, $b = 8$, $p = 0.3$, x axis: modules one by one, y axis: number of rules in the module (with and without different extensions)

- messages: See Fig. 7
  - distribution is very biased, first modules are much mode loaded than subsequent ones, we should look into this more deeply

4.3.3 Greedy classifier partitioning: k-independence

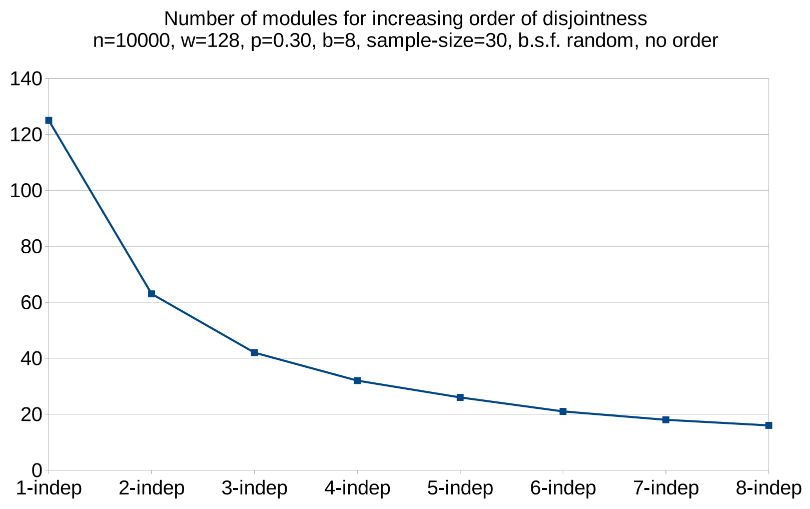

Figure 8: k-independent Classifier Decompositions for increasing k: $n = 10000$, $w = 128$, $b = 8$, $p = 0.3$, plain greedy partitioning

- messages: See Fig. 8
  - quasi-disjointness is highly effective: already setting $k = 2$ halves the number of modules and all subsequent increases seem to do the same
  - of course, we pay runtime cost for that; we still need to decide whether this decrease in module count is worth the increase in false-positive-check load (see later)

5 Implementation and numerical evaluations

- we created an initial runtime to test the viability of the approach
- the implementation works with random data, but it is deliberately designed to do everything that a real implementation will need to do for classification
- next step will be to integrate the greedy decomposer and the runtime into DPDK

5.1 SIMD strategies

- only the number of SIMD register loads count, the running time of the SIMD instructions themselves is negligible
- factor in that the number of SIMD registers is rather limited (8 for SSE)
- so we need to minimize the number of register loads: SIMD strategies

5.1.1 Per-packet strategy: 8P1M

- match the first rule on a batch of k packets, then load the next rule, etc.
- needs $n + k$ register loads to match n rules on k packets (initially load k packets then load a new rule n times)
- for $k = 8$ we get the 8P1M strategy

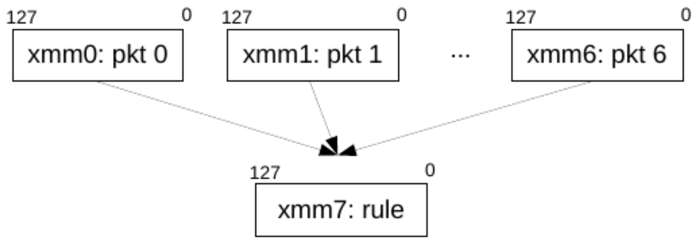

Figure 9: Per packet strategy

5.1.2 Per-rule strategy: 1P8M

- match the same packet on the first k rules, then load the next packet and the next k rules, etc.
- needs $2n$ register loads to match n rules on k packets (in each round load k rules plus k packets, and there are $\frac{n}{k}$ rounds)
- for $k = 8$ we get the 1P8M strategy

5.2 Implementation considerations

- we assume x86-64/AVX2 (16x256 bit registers)
- we need the following functionality for our runtime
  - **masking:** to select relevant bits (wrt to each module) from the header
  - **rule matcher:** to match (k-)disjoint rules in each module
  - **false-positive check:** to validate whether matching rule (wrt to the relevant bit positions) indeed match

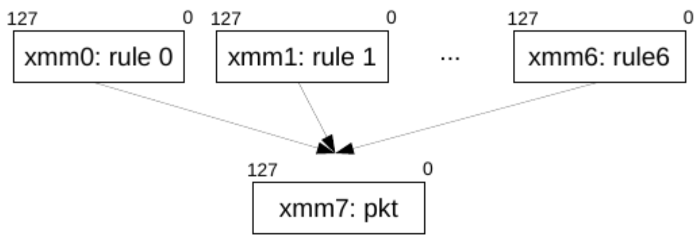

Figure 10: Per rule strategy

  - **priority comparator:** to select the highest priority rule across modules that matched
- the rule-matcher in each module is a simple array of 2^b values
  - for each 2^b different value that can appear (after masking) in the packet header, the array specifies the corresponding pair of rule id and priority values
  - this way, rule matching boils down to simple array lookup (can use 32-bit gather)
  - bit positions are some consecutive $b = 8$ or $b = 16$ bits for each module
- SIMD implementations lack many useful operations for small bit widths (no load/mask/-gather for `_epi8` and `_epi16`)
  - we chose 32 bits (`_epi32`) as the basic unit of parallelization
  - 8 or 16 bit headers/rules are expanded to 32 bits
  - this immediately wastes 16 bits per SIMD lane
  - (rule id, priority) pairs are also encoded on 32 bits, so lookup result will again yield 32 bit values
- the per-packet strategy needs fewer register loads, we went with this strategy: we implemented the 8P1M strategy

5.3 Pseudocode

- the implementation uses the below pseudocode; note that currently the code does multiple loads for each packet, code rewrite is underway to avoid that
- parameters are set up beforehand even at classifier-construction time and/or runtime, at receiving the packet batch:
  - `pack`: base address for packet batch
  - `masks`: base address for array containing mask for each module
  - `module`: lookup array for each module
  - `rstar` and `rbit`: "don't care" mask and effective bits for each rule

```
for(k=0;k<p;k+=8){                    // for each batch of 8 packets
  for(j=0;j<m;j++){                   // for each module

    // MASK relevant bits of the packets with preloaded masks
    __m256i mask = _mm256_load_si256(masks + j*8);
    __m256i hmask = _mm256_i32gather_epi32(pack + k*b, mask, 1);
    hmask =_mm256_and_ps(hmask, X);

    // MATCH masked bits in the module
    __m256i match = _mm256_i32gather_epi32(module[j]), hmask, 4);
    uint32_t *mp = (uint32_t *)&match;

    // FALSE POSITIVE CHECK for each packet
    for(i=0;i<8;i++){
      __m256i A =_mm256_load_si256(pack + (k+i)*b));
      __m256i B =_mm256_load_si256(rstar + (mp[i]/exp_12)*b));
      __m256i C =_mm256_load_si256(rbit +(mp[i]/exp_12)*b));
      __m256i D =_mm256_and_ps(A, B);
      __m256i E =_mm256_cmpeq_epi32(C,D);
      int fpc =_mm256_movemask_ps(E);

      // PRIORITY COMPARISON
      if(fpc == 255 && mp[i] > result_per_packet[k+i])
        result_per_packet[k+i]=mp[i];
    }
  }
}
```

5.4 Evaluation

- used random data for classifiers, results are fit from the simulations

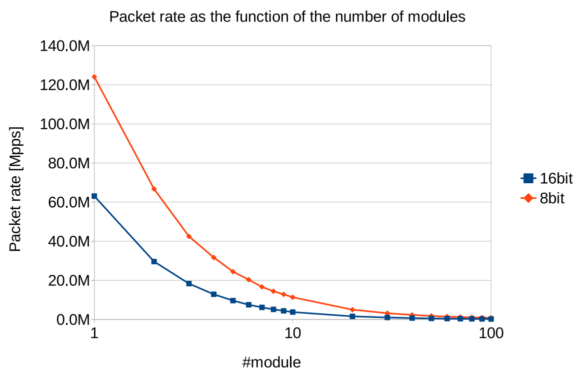

Figure 11: (Expected) packet rate as the function of the number of modules for 8-bit ($b = 8$) and 16-bit ($b = 16$) modules

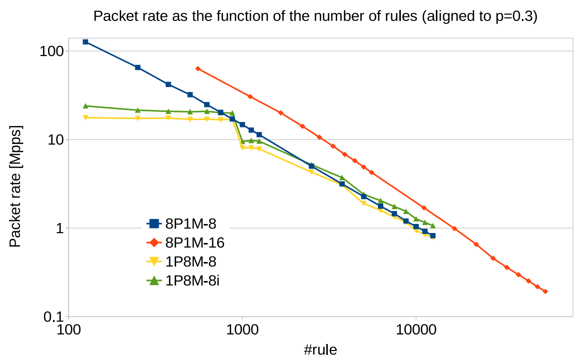

Figure 12: (Expected) packet rate as the function of the number of rules for 8-bit ($b = 8$) and 16-bit ($b = 16$) modules

- messages: see Fig. 11 and Fig. 12
  - 8-bit implementation is twice as fast as 16-bit one, when only looking at the number of modules
  - reason is deteriorating cache performance: looking at the numbers with `perf`, we see that the 16-bit implementation ($b = 16$) causes 55 times more cache misses than the 8-bit implementation ($b = 8$)
  - but since a 16-bit module can hold more rules, the 16-bit implementation is still faster if we look at the overall number of rules
  - we can expect line-rate performance (with 64 byte packets) even for several thousand rules
  - of course, real performance will be worse than this due the the unavoidable overhead of operational code

6 The DPDK built-in classifier `librte_acl`

- the DPDK already contains a SIMD classifier implementation `librte_acl`

6.1 Features

- parallel lookup on multiple rule sets (OpenFlow, ACL, etc.), lookup for each category in a single run
- arbitrary rule priorities
- flexible header field definitions (can match on arbitrary protocol)
- range and mask searches in rules
- batch lookup a'la DPDK
- hard resource limits on the classifier data structure
- multiple SIMD lookup backends (scalar, SSE, SSE4, AVX2)

6.2 The `librte_acl` API

- `rte_acl_create()`: create an `rte_acl_ctx` context
- `struct rte_acl_field_def` and `struct rte_acl_rule`: header field defs and the actual rules
- `rte_acl_add_rules()`: fill up the classifier
- `rte_acl_build()`: construct the classifier data structure from the given rules
- `rte_acl_classify()`: do classification on packet batch, with a separate SIMD backend for each CPU arch., best backend selected automatically
- `rte_acl_classify_alg()`: enforce specific SIMD backend

6.3 The classifier data structure

- classification is performed on (possibly multiple) multibit **prefix trees** (tries), stride=8
- overlapping rules are statically disambiguated by priority when building/updating the tries
- categories share trie(s), so a single trie may contain the results for multiple independent classifiers
- supports dynamic rule addition, but no rule modification/delete
- trie traversed in 4-byte chunks (packet read in 4 byte units) except the first pass that is done out of the main loop and reads only a single byte
- the trie may still take too much space due to cross-product effects between different rules and the related combinatorial explosion
- to conserve some space, `librte_acl` can split rule-set into non-intersecting subsets and construct a separate trie for each, at the cost of increased runtime classification time

6.4 Results

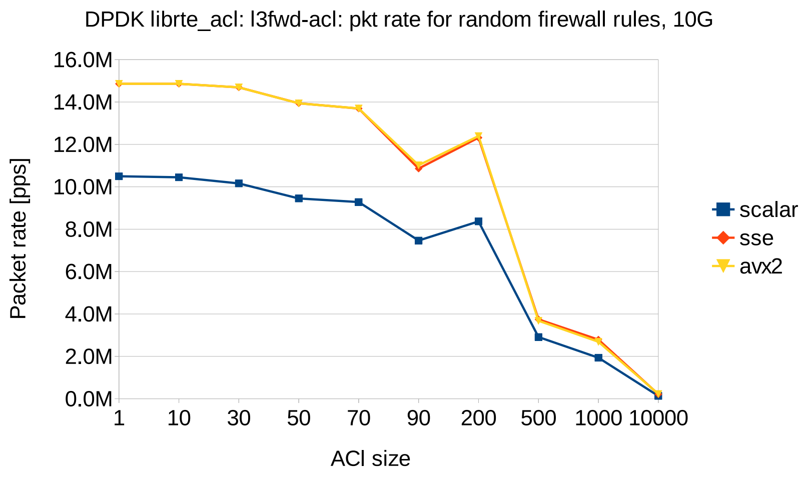

Figure 13: `librte_acl` packet rate for increasingly larger random IPv4 5-tuple classifiers

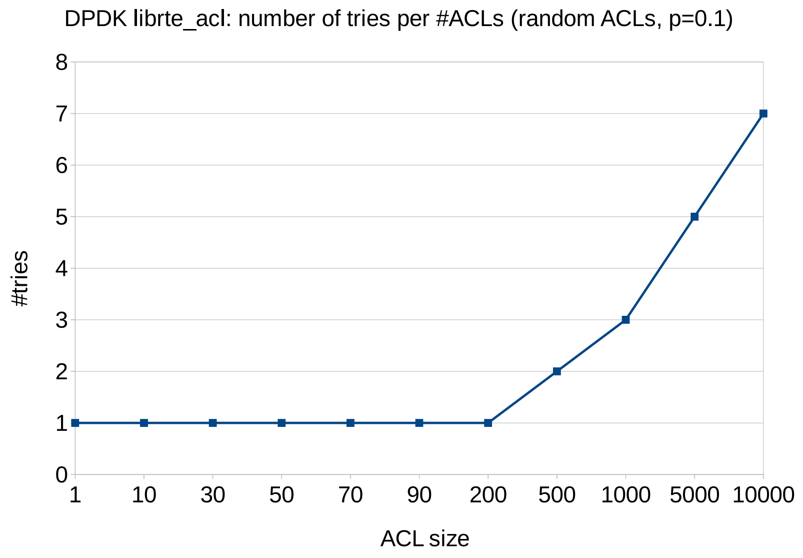

Figure 14: `librte_acl` number of lookup tries for increasingly larger random IPv4 5-tuple classifiers

- messages: See Fig. 13, Fig. 14, and Fig. 15
  - `librte_acl` achieves 10G line rate with 64 byte packets on small classifiers
  - performance quickly drops as classifier size grows
  - this is due to the increase in the number of tries created (which all have to be searched for each packet) and the memory size explosion and corresponding CPU cache-miss rate (0.5 Gbyte for 10K rules)

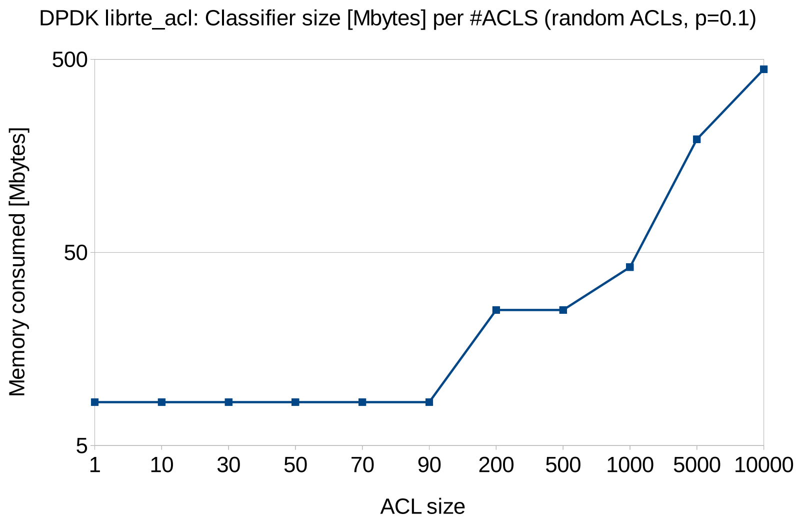

Figure 15: `librte_acl` memory footprint for increasingly larger random IPv4 5-tuple classifiers

  – SSE and AVX2 backends do not seem to differ, classifier is not CPU-bounded (will re-check this at 40Gbps)

References

[1] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, “SAX-PAC (Scalable And eXpressive PAcket Classification),” in *ACM SIGCOMM*, pp. 15–26, 2014.

[2] S. Nikolenko, K. Kogan, G. Rétvári, E. Bérczi-Kovács, and A. Shalimov, “How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes,” in *Proc. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): GI 2016: 9th IEEE Global Internet Symposium*, 2016.