General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures

Pavel Chuprikov*†, Kirill Kogan†, Sergey Nikolenko*

*Steklov Institute of Mathematics at St. Petersburg †IMDEA Networks Institute, Madrid

***Abstract*—Ternary Content-Addressable Memory (TCAM) is a powerful tool to represent network services with line-rate lookup time. There are various software-based approaches to represent multi-field packet classifiers. Unfortunately, all of them either require exponential memory or apply additional constraints on field representations (e.g, prefixes or exact values) to have line-rate lookup time. In this work, we propose alternatives to TCAM and introduce a novel approach to represent packet classifiers based on ternary bit strings (without constraining field representation) on commodity longest-prefix-match (LPM) infrastructures. These representations are built on a novel property, *prefix reorderability*, that defines how to transform an ordered set of ternary bit strings to prefixes with LPM priorities in linear memory. Our results are supported by evaluations on large-scale packet classifiers with real parameters from ClassBench; moreover, we have developed a prototype in P4 to support these types of transformations.**

I. Introduction

Packet classification is a core functionality for representing packet processing programs on the data plane. There are two major program categories: traffic forwarding between certain points in a communication network and service policies that guarantee desired traffic properties or track network behavior during forwarding (e.g., quality-of-service, access-control, firewall). Both can be captured as tuple matching with action sets, but they have distinct behavior and may rely on different invariants; e.g., forwarding tables can be represented by prefixes with priorities based on longest-prefix-match (LPM) while policies can consider general multi-field classifiers; forwarding tables may change frequently, while policies representing economic models or specific traffic signatures are designed mostly *a priori*. In this work, we concentrate on policies in the second category based on multi-field packet classifiers.

It is easier and more efficient to represent prefixes with LPM priorities than to use multi-field packet classifiers in software-based approaches [1]. Complexity bounds derived from computational geometry imply that a software-based packet classifier with N rules and $k \geq 3$ fields uses either $O(N^k)$ space and $O(\log N)$ time or $O(N)$ space and $O(\log^{k-1} N)$ time [2] , which makes them either too slow or too memory-intensive even with few prefix-fields.

Software-based approaches become even worse if classification rules are represented as general ternary bit strings, which is extremely useful in many applications [3], [4], [5]. Ternary content-addressable memory (TCAM) was introduced to overcome performance limitations of software-based solutions to represent multi-field packet classifiers and add a new level of expressiveness [6]. Unfortunately, TCAMs are expensive and power hungry [7], so TCAMs of a sufficient size are the *de facto* standard for classifier implementations only in high-end network elements [8], [9]. Most network elements efficiently implement prefix classifiers with LPM priorities at line-rate.

In this work, we explore alternatives to TCAMs and other software-based approaches and show how to represent multi-field packet classifiers on commodity LPM infrastructures (transparently to them) with line-rate performance. Various approaches to represent multi-field packet classifiers on LPM infrastructures exist, but all of them impose additional constraints on how fields are represented (e.g., prefixes or exact values) and most do not achieve desired worst-case guaranteed lookup time [10], [2], [11], [12]. Unlike prior art, we do not apply additional constraints on field representations and assume that classifier rules are ternary bit strings with general priorities as in TCAMs. We do not propose a specific classifier implementation but rather define an *abstraction layer* that chooses a subset of bit indices to be used in the lookup process. A classifier based on these bit indices can be transparently represented by other schemes, both in hardware and software.

The paper is organized as follows. Section II introduces the model; Section III, a novel structural property, *prefix reorderability*, with an optimal algorithm that transforms a given classifier into a prefix LPM classifier without extra memory (if possible). In Sections IV and V we show how to represent non-prefix-reorderable classifiers on existing LPM infrastructure without and with extra memory. Since classification width supported by LPM infrastructures is usually limited to 32 or 128 bits, in Section VI we show how to represent much wider classifiers on LPM infrastructures and study a composition of prefix reorderability with another structural property, *rule disjointness* (order independence [13]). Section VII discusses dynamic updates. In Section VIII, we evaluate our approach on ClassBench classifiers with real parameters [14]. Section IX outlines implementation details on top of the P4 domain-specific language [15]; we have released the code under an open source license. Section X discusses related prior art, and Section XI concludes the paper.

II. Model Description

In this section, we provide formal definitions for further exposition, starting with the basic notions of a packet header and classifier. A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits: each bit $h_i \in H$ has a value of either zero or one, $h_i \in \{0, 1\}$, $1 \le i \le w$. For example, (1 0 0 0) is a 4-bit header. A *classifier* $\mathcal{K} = \{R_1, \ldots, R_N\}_\prec$ is an ordered (by $\prec$) set of rules, where each *rule* $R_i = (F_i, A_i)$ consists of a *filter* F_i and a pointer to the corresponding *action* A_i. A filter $F = (f_1, \ldots, f_w)$ is a sequence of, again, w values corresponding to bits in the headers, but this time the possible bit values are 0, 1, or $*$ ("don't care"). We illustrate classifiers by a table as in Example 1, where $R_i \prec R_{i+1}$ is implied.

Example 1: A classifier $\mathcal{K}$ on four bits (w = 4).

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	1	0	0	A_1
R_2	0	$*$	$*$	$*$	A_2
R_3	1	0	1	$*$	A_3
R_4	1	$*$	0	$*$	A_4

A classifier's main purpose is to find the action corresponding to the highest priority rule matching a given header. A header H *matches* a rule R if it matches R's filter, and it matches a filter F if for every bit of H the corresponding bit of F has either the same value or $*$. The set of rules has a non-cyclic priority ordering $\prec$; if a header matches both R and R' for $R \prec R'$, the action of rule R is applied. E.g., in Example 1 the header (0 1 0 0) matches both R_1 and R_2, but A_1 is applied. Filters F_1 and F_2 are *disjoint* if no single header matches both of them. Otherwise, F_1 and F_2 *intersect*, and rules have to be prioritized. Two rules *intersect* (are *disjoint*) if their filters intersect (are disjoint). In Example 1, R_1 and R_2 (as well as their filters) intersect (e.g., (0 1 0 0) matches both filters), while R_1, R_3, and R_4 are pairwise disjoint.

A classifier $\mathcal{K}$ is called a *prefix* classifier if in every filter $F \in \mathcal{K}$ all 0s and 1s precede all $*$ bits, i.e., they match prefixes of a header. A prefix classifier is called a *longest prefix match* (LPM) classifier if for every two intersecting rules R_1 and R_2, the one with the longer prefix in its filter takes precedence. LPM does not necessarily mean that a longer prefix is higher in the ordering: the constraint only concerns intersecting filters, so in Example 1 the first three rules comprise an LPM classifier.

Two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$ are *equivalent* if they choose the same action for every packet. Formally, for every header $H \in \{0, 1\}^w$ H matches a rule in $\mathcal{K}_1$ iff it matches a rule in $\mathcal{K}_2$, and if H does match $R_1 = (F_1, A_1) \in \mathcal{K}_1$ and $R_2 = (F_2, A_2) \in \mathcal{K}_2$ then $A_1 = A_2$.

III. Prefix-reorderable classifiers

Our main objective is to find representations of classifiers with general priorities on commodity LPM infrastructures transparently to their implementations. To achieve this, we proceed in two steps: (1) reorder bit indices of ternary bit strings to transform them into prefixes (still with original priorities, not necessary LPM) and (2) convert the resulting prefix classifier (with general priorities) to an equivalent LPM classifier. Here we assume that the classification width w is at most w_{LPM} used in LPM infrastructures (e.g., 32 or 128 bits). We will return to the case $w > w_{\text{LPM}}$ in Section VI.

A. Ternary bit strings to prefixes

In this subsection, we identify precise conditions when a given classifier can be transformed by bit reordering into an equivalent prefix classifier. Moreover, we present an algorithm that constructs a required order.

Let $B = (b_1, b_2, \ldots, b_k)$, $k \le w$, be a sequence of distinct bit indices, $1 \le b_i \le w$, that represents a desired bit order. For a header $H = (h_1, h_2, \ldots, h_w)$, we denote by H^B the (sub)header $(h_{b_1}, h_{b_2}, \ldots, h_{b_k})$. E.g., for $H = (0\ 1\ 0\ 0)$ and $B = (3, 2)$ $H^B = (0\ 1)$. Similarly, for a filter F we have $F^B = (f_{b_1}, f_{b_2}, \ldots, f_{b_k})$, extending this notation to a rule $R = (F, A)$: $R^B = (F^B, A)$. Finally, for a classifier $\mathcal{K} = \{R_1, \ldots, R_N\}$ the *B-reordering* of $\mathcal{K}$, denoted $\mathcal{K}^B$, is obtained from $\mathcal{K}$ as follows: (i) replace each rule $R \in \mathcal{K}$ by R^B; (ii) for incoming packets, replace H by H^B prior to matching. E.g., in Example 1 for $B = (3, 1, 2)$ we have

$\mathcal{K}^B$	#3	#1	#2	*Action*
R_1^B	0	0	1	A_1
R_2^B	$*$	0	$*$	A_2
R_3^B	1	1	0	A_3
R_4^B	0	1	$*$	A_4

To match $H = (0\ 1\ 0\ 0)$ in this classifier, H is firstly transformed to $H^B = (0\ 0\ 1)$, which matches R_1^B.

Note that for $H' = (0\ 1\ 0\ 1)$ in the above classifier the matching rule is also R_1^B, while in the original classifier it would match R_2 with a different action. Thus, $\mathcal{K}$ and $\mathcal{K}^B$ are not equivalent in general. Equivalence is obviously preserved, however, when no bits are omitted.

Observation 1: For every classifier $\mathcal{K}$ on w bits, if B is a permutation of $(1, \ldots, w)$ then $\mathcal{K}$ is equivalent to $\mathcal{K}^B$.

We call a classifier $\mathcal{K}$ on w bits *prefix-reorderable* if there exists a *permutation* B of $(1, \ldots, w)$ such that $\mathcal{K}^B$ is a prefix classifier (here $\mathcal{K}^B$ is always equivalent to $\mathcal{K}$).

Example 2: Classifier from Example 1 is not prefix but prefix-reorderable: for $B' = (1, 3, 2, 4)$ we have

$\mathcal{K}^{B'}$	#1	#3	#2	#4	*Action*
$R_1^{B'}$	0	0	1	0	A_1
$R_2^{B'}$	0	$*$	$*$	$*$	A_2
$R_3^{B'}$	1	1	0	$*$	A_3
$R_4^{B'}$	1	0	$*$	$*$	A_4

Unfortunately, not every classifier is prefix-reorderable.

Example 3: Consider the following classifier:

$\mathcal{K}$	#1	#2	*Action*
R_1	0	$*$	A_1
R_2	$*$	0	A_2

The key to prefix-reorderability criteria lies in the reason why Example 3 fails. Consider a filter $F = (f_1, \ldots, f_w)$; denote by $\text{exact}(F)$ the set of bit indices i such that $f_i \neq *$; we extend exact to rules as $\text{exact}((F, A)) = \text{exact}(F)$. In Example 3 we have $\text{exact}(R_1) = \{1\}$, and $\text{exact}(R_2) = \{2\}$, but in a prefix classifier $*$ must follow 0s and 1s in F^B.

Observation 2: For any filter F on w bits and any permutation B of $(1,\dots,w)$, if F^B is a part of a *prefix* classifier, then indices from $\mathrm{exact}(F)$ must precede in B all other indices.

Algorithm 1 `perm`$(\mathcal{K})$

```
1: $E'_1,\dots,E'_{|\mathrm{exact}(\mathcal{K})|} \leftarrow$ sort_by_size$(\mathrm{exact}(\mathcal{K}))$
2: $B \leftarrow E'_1$
3: for $i \in \{1,\dots,|\mathrm{exact}(\mathcal{K})|-1\}$ do
4:     if $E'_i \not\subseteq E'_{i+1}$ then
5:         return NO
6:     $B \leftarrow B, E'_{i+1} \setminus E'_i$
7: $B \leftarrow B, \{1,\dots,w\} \setminus E'_{|\mathrm{exact}(\mathcal{K})|}$
8: return YES(B)
```

In Example 3, we cannot satisfy Observation 2 for both filters at once. This generalizes to any two filters with $\mathrm{exact}(F_1) \neq E_1$ and $\mathrm{exact}(F_2) = E_2$ such that there exists $x \in E_1 \setminus E_2$ and $y \in E_2 \setminus E_1$, i.e., $E_1 \not\subseteq E_2$ and $E_2 \not\subseteq E_1$. The next theorem states that to require either $E_1 \not\subseteq E_2$ or $E_2 \not\subseteq E_1$ is actually sufficient. The criterion is naturally formulated in terms of the structure of $\mathrm{exact}(\mathcal{K}) = \{\mathrm{exact}(R) : R \in \mathcal{K}\}$.

Theorem 1 (chain criterion): A classifier $\mathcal{K}$ is prefix-reorderable iff for every $E_1, E_2 \in \mathrm{exact}(\mathcal{K})$ either $E_1 \subseteq E_2$ or $E_2 \subseteq E_1$ holds, i.e., $\mathrm{exact}(\mathcal{K})$ can be reordered to form a "chain": $E_{i_1} \subseteq E_{i_2} \subseteq \ldots \subseteq E_{i_{|\mathcal{K}|}}$. The permutation of bit indices can be found in $O(|\mathcal{K}| \cdot w)$ time (if one exists).

Proof: ($\Rightarrow$) Necessity follows from the observation above: if we have $x \in E_i \setminus E_j$ and $y \in E_j \setminus E_i$ for some i, j, and a B-reordering of $\mathcal{K}$ is a prefix classifier, then both y must precede x in B and x must precede y, a contradiction.

($\Leftarrow$) Sufficiency. We need to present a permutation of bits that leads to prefix representations of all rules. We claim that Algorithm 1 constructs such a permutation or returns `NO` if the theorem's assumption does not hold. First, by assumption $\mathrm{exact}(\mathcal{K})$ ordered by the size of $|E_i|$ is a chain, i.e., $E'_i \subseteq E'_{i+1}$ for all i, so the check on line 4 never succeeds. Next, since $E'_1 \subseteq \ldots \subseteq E'_{|\mathcal{K}|}$, after m iterations of the loop on line 3 B represents a permutation of E'_m. Finally, for every i in the resulting permutation, all bits from E'_i precede all bits from $\{1,\dots,w\} \setminus E'_i$ since they get mapped later. Thus, B turns the classifier $\mathcal{K}$ into a prefix one. Algorithm 1 needs $O(|\mathcal{K}| \cdot w)$ operations to construct the set $\mathrm{exact}(\mathcal{K})$; sorting can be done in time $O(w + |\mathcal{K}|)$ with radix sort, any set-theoretic operation (including sorting) requires $O(w)$ time, and the loop has $|\mathcal{K}| - 1$ iterations of time $O(w)$ each. ■

Note that in Example 3 the set $\mathrm{exact}(\mathcal{K}) = \{\{1\},\{2\}\}$ is not a chain. Algorithm `perm` exploits the criterion (Theorem 1) to construct a permutation of bit indices that results in a prefix-reorderable classifier, if one exists. To illustrate the behavior of `perm`, consider again the classifier $\mathcal{K}$ from Example 1. Here, if sorted by sizes, we have $\mathrm{exact}(\mathcal{K}) = \{\{1\},\{1,3\},\{1,2,3\},\{1,2,3,4\}\}$. Line 2 assigns $B = (1)$. On the first iteration, we have $E'_i = \{1\}$ and $E'_{i+1} = \{1,3\}$, thus $B = (1,3)$. On the second iteration, $E'_i = \{1,3\}$, $E'_{i+1} = \{1,2,3\}$, so now $B = (1,3,2)$. Line 7 adds the last bit index 4, and the algorithm returns $B = (1,3,2,4)$, leading to the same prefix classifier as in Example 2. Algorithm `perm` can now be used to find the equivalent prefix classifier if it exists; we let `general_to_prefix`$(\mathcal{K}) =$ `NO` if `perm`$(\mathcal{K}) =$ `NO` and `general_to_prefix`$(\mathcal{K}) = \mathcal{K}^B$ if `perm`$(\mathcal{K}) =$ `YES`(B).

Algorithm 2 `prefix_to_lpm`$(\mathcal{K})$

```
1: while $\mathcal{K}$ is not LPM do
2:     $R_1, R_2 \leftarrow$ rules violating LPM
3:     if $|\mathrm{exact}(R_1)| < |\mathrm{exact}(R_2)|$ then        ▷ $R_1$ has a shorter prefix
4:         $\mathcal{K} \leftarrow \mathcal{K} \setminus R_2$
5:     else
6:         $\mathcal{K} \leftarrow \mathcal{K} \setminus R_1$
7: return $\mathcal{K}$
```

B. Prefix to LPM *classifiers*

Once a classifier is transformed to an equivalent prefix classifier $\mathcal{K}'$, the rule priorities of $\mathcal{K}'$ do not necessary conform to LPM priorities. In this part, we introduce a transformation of a prefix classifier (with general priorities) to an LPM classifier that does not add new rules and works in time $O(|\mathcal{K}| \cdot w)$.

Example 4: Consider a prefix classifier $\mathcal{K}_0$:

$\mathcal{K}_0$	#1	#2	#3	#4	*Action*
R_1	0	∗	∗	∗	A_1
R_2	1	0	∗	∗	A_2
R_3	1	0	1	∗	A_3

Observe that the last two rules in $\mathcal{K}_0$ do not conform to LPM since R_2 intersects with R_3, R_2 has a shorter prefix, but $R_2 \prec R_3$. In this case, the last rule is just redundant and can be removed since all packets matched by R_3 will always be matched by R_2 with higher priority. As a result, we get an equivalent prefix classifier that already has LPM priorities:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	0	∗	∗	∗	A_1
R_2	1	0	∗	∗	A_2

It turns out that it suffices to remove redundant rules of this type to transform a prefix classifier into an LPM classifier, and Algorithm 2 does this without increasing the number of rules.

Theorem 2: `prefix_to_lpm` transforms a prefix classifier $\mathcal{K}$ into an equivalent LPM classifier $\mathcal{K}'$ with $|\mathcal{K}'| \leq |\mathcal{K}|$. Moreover, it can be implemented in time $O(w \cdot |\mathcal{K}|)$.

Proof: Since each iteration reduces the number of pairs of rules that violate LPM order, and there are finitely many such pairs, `prefix_to_lpm` terminates. To prove that it preserves equivalence, consider two rules: R_i with prefix x and R_{i+1} with prefix y: w.l.o.g. $|x| < |y|$, so $y = uz$, where $|u| = |x|$. There are two cases: (1) $x = u$ and (2) $x \neq u$. In the former case, all packets matched by y would be already matched by x; thus, rule R_{i+1} is redundant and can be removed. In the latter case, the rules do not intersect and hence conform to LPM priorities.

A naive implementation of `prefix_to_lpm`, shown in Algorithm 2, works in $O(w \cdot |\mathcal{K}|^2)$ time. A faster approach can be based on bit tries: start with an empty bit trie T. Sort rules in non-decreasing order of $\mathrm{exact}(R_i)$ in $O(|\mathcal{K}| + w)$ time using radix sort. Then, for every rule R_i in this order with 0-1 prefix p_i, check for every rule R_j such that its prefix p_j lies on the path from T's root to p_i that $R_i \prec R_j$; if the check does not

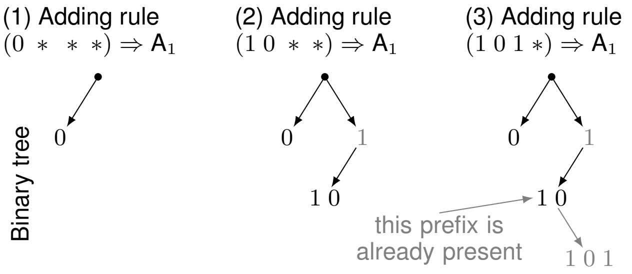

(1) Adding rule
$(0 * * *) \Rightarrow A_1$
(2) Adding rule
$(1 0 * *) \Rightarrow A_1$
(3) Adding rule
$(1 0 1 *) \Rightarrow A_1$
Binary tree
0
0
1
1 0
0
1
1 0
1 0 1
this prefix is already present

Fig. 1. Adding prefixes of $\mathcal{K}_0$ to a binary trie.

fail, add p_i to T; if for some R_j $p_i = p_j$, leave in T the rule with higher priority. Figure 1 shows an example of adding the prefixes of $\mathcal{K}_0$ to this trie: the third rule's prefix (shown in gray) is not included since an earlier rule already has a shorter prefix. The resulting classifier $\mathcal{K}$ consists of all rules whose prefixes are left in T, together with the corresponding actions; the theorem follows since we always keep only top priority rules among intersecting ones. ■

In this section, we have shown how to transform prefix-reorderable classifiers to LPM classifiers. The proposed transformations, `perm` and `prefix_to_lpm`, are transparent to internal implementations of LPM infrastructures. Next we will see how to represent non-prefix-reorderable classifiers on commodity LPM infrastructures.

IV. Non-prefix-reorderable classifiers without extra memory

We introduce two approaches for dealing with non-prefix-reorderable classifiers. The first, which we consider in this section, requires prefix reorderability only on a subset of rules, exploiting the capability of a target architecture to perform multiple LPM lookups at line-rate and leading to multiple prefix-reorderable classifiers that are together equivalent to the original classifier.

A. Groupwise reorderability

In most cases, target architectures are able to perform multiple lookups to the LPM infrastructure at line-rate. Hence, we can partition rules into multiple prefix-reorderable classifiers and look up a header in each group, combining the outcomes to choose the highest priority rule as the final result. Since each rule appears only in one group, no extra memory is required.

Example 5: Consider the following (non-prefix-reorderable) classifier $\mathcal{K}$ that will be our running example in this section:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_3	$*$	1	0	0	A_3
R_4	0	0	$*$	$*$	A_4
R_5	$*$	0	1	$*$	A_5
R_6	$*$	1	0	$*$	A_6
R_7	$*$	0	$*$	$*$	A_7

Algorithm 3 `mg`(H) for $\mathcal{K}$ and $\{\mathcal{K}_1, \ldots, \mathcal{K}_\beta\}$

```
1: for $i \in 1, \ldots \beta$ do
2:     $R_i^* \leftarrow$ find_match$(\mathcal{K}_i, H)$
3: $(F, A) \leftarrow \max_{1 \le i \le \beta} R_i^*$, w.r.t $\prec_{\mathcal{K}}$
4: return $A$
```

Algorithm 4 `min_group_partition`$(\mathcal{K})$

```
1: $\mathcal{E}_1, \ldots, \mathcal{E}_\beta \leftarrow$ min_chain_partition$(\mathrm{exact}(\mathcal{K}), \subseteq)$
2: $P \leftarrow \emptyset$
3: for $i \in \{1, 2, \ldots, \beta\}$ do
4:     $\mathcal{K}_i \leftarrow \{R \in \mathcal{K} : \mathrm{exact}(R) \in \mathcal{E}_i\}$
5:     $P \leftarrow P \cup \{\mathcal{K}_i\}$
6: return $P$
```

$\mathcal{K}$ can be split into prefix-reorderable classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_4	0	0	$*$	$*$	A_4
R_7	$*$	0	$*$	$*$	A_7

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
R_3	$*$	1	0	0	A_3
R_5	$*$	0	1	$*$	A_5
R_6	$*$	1	0	$*$	A_6

If $\mathcal{K}_1$ and $\mathcal{K}_2$ both receive a header $H = (0\ 0\ 1\ 0)$, the former finds a matching rule R_2 and the latter finds R_5. The multigroup implementation compares priorities of the matched rules and returns the one with higher priority, namely R_2.

Algorithm 3 defines the lookup procedure to a multigroup representation of a given classifier. Since all rules of the original classifier $\mathcal{K}$ participate in the lookup process, and the rule with highest priority is returned, the multigroup representation of a given classifier is equivalent to $\mathcal{K}$. Different objectives can be optimized in the construction of multigroup representations. We begin with a natural one that minimizes the number of prefix-reorderable groups.

B. Minimizing the number of groups

*Problem 1 (*MINGR*):* Find a partition of the rules of a given classifier $\mathcal{K}$ into a minimal number of disjoint prefix-reorderable groups.

Intuitively, the resulting number of groups should depend on the order in which rules are assigned to groups, and at first glance it looks like a hard problem. However, according to the chain criterion in Theorem 1, $\mathcal{K}'$ is a prefix-reorderable subset of $\mathcal{K}$ if and only if $\mathrm{exact}(\mathcal{K}')$ is a chain. Thus, instead of partitioning $\mathcal{K}$ into prefix-reorderable groups we can partition $\mathrm{exact}(\mathcal{K})$ into chains and then distribute rules to groups according to the chain partition. This transformation changes the problem in two important ways: first, the size of $\mathrm{exact}(\mathcal{K})$ is usually much smaller than the number of rules in $\mathcal{K}$; second, most importantly, the minimal chain partition problem can be solved in polynomial time with the help of order theory [16].

The `min_group_partition` algorithm for MINGR, shown in Algorithm 4, exploits this idea. It calls the `min_chain_partition` function that returns a minimal size chain partition of a given ordered set based on

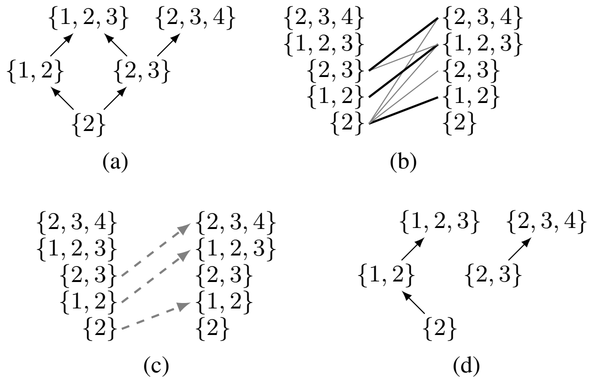


Fig. 2. Dilworth's decomposition theorem: (a) original ordered set; (b) the bipartite graph and its maximal matching; (c) chain decomposition in the bipartite graph; (d) chain decomposition in the original graph.

Dilworth's decomposition theorem [17]. The implementation of `min_chain_partition` shown in Algorithm 5 is based on Fulkerson's proof of Dilworth's theorem [18].

Example 6: Fig. 2 illustrates Algorithm 4 with $\mathcal{K}$ from Example 5. Starting from $\text{exact}(\mathcal{K}) = \{\{2\}, \{1,2\}, \{2,3\}, \{1,2,3\}, \{2,3,4\}\}$, we order it by inclusion (Fig. 2a), construct the corresponding bipartite graph (Fig. 2b), find a maximal matching, in this case $\{(\{2\}, \{1,2\}), (\{1,2\}, \{1,2,3\}), (\{2,3\}, \{2,3,4\})\}$ (shown on Fig. 2b and c), and construct the chains by following this matching (Fig. 2c and d).

Algorithm 5 `min_chain_partition`$(X, \leq)$

```
1: $G \leftarrow$ bipartite graph, s.t. $V(G) = E(G) = \emptyset$
2: for $x \in X$ do
3:     $G$.add_left_vertex($x_L$)
4:     $G$.add_right_vertex($x_R$)
5: for $x, x' \in X$ do
6:     if $x < x'$ then
7:         $G$.add_edge($x_L, x'_R$)
8: $M \leftarrow$ max_matching($G$)
9:
10: $H \leftarrow$ graph, s.t. $V(H) = X$ and $E(H) = \emptyset$
11: for $(x_L, x'_R) \in M$ do
12:     $H$.add_edge($x, x'$)
13: return connected_components($H$)
```

Theorem 3: The algorithm `min_chain_partition` finds an optimal solution to the MINGR problem in time $O\left(|\,\text{exact}(\mathcal{K})|^{5/2} + |\mathcal{K}|^2 w\right)$.

Proof: For any prefix-reorderable partition of rules, by Theorem 1 the set of $\text{exact}(\mathcal{K}_i)$ in each group $\mathcal{K}_i$ constitutes a chain, leading to a chain partition. Vice versa, if we have found a chain partition it suffices to group rules with respect to this partition to get a prefix-reorderable partition of the same size. Execution time of `min_pmgr` is dominated by the call to `min_chain_partition`, whose execution time is dominated by `max_matching`, so the Hopcroft–Karp algorithm [19] yields running time $O\left(|K|^{5/2}\right)$. It takes time $O\left(|\mathcal{K}|^2 w\right)$ to construct the ordered set $(\text{exact}(\mathcal{K}), \subseteq)$: we have to compute $\subseteq$ for every pair of $|\mathcal{K}|$ elements in this set. ■

C. Mixed representations

The number of parallel and serial lookups to the LPM infrastructure at line rate is a property of the underlying target architecture. In the worst case, one can construct artificial instances of classifiers whose optimal representations require an exponential (in width w) number of disjoint groups.

Theorem 4: There exists a classifier $\mathcal{K}$ with $|\mathcal{K}| = O\left(\frac{1}{\sqrt{w}} 2^{\frac{w}{2}}\right)$ such that an optimal solution of the MINGR problem requires exactly $|\mathcal{K}|$ groups.

Proof: The idea of this worst-case example is to construct many filters with different sets of $*$ fields that are not subsets of each other (and hence cannot be reordered). Consider the set of ternary bit strings of length $\frac{w}{2}$ that contain exactly $\frac{w}{4}$ 1s and $\frac{w}{4}$ $*$. There are $O\left(2^{\frac{w}{2}}/\sqrt{w}\right)$ such strings. If we concatenate each of them with a different string from $\{0,1\}^{w/2}$, we get $O\left(2^{\frac{w}{2}}/\sqrt{w}\right)$ non-intersecting filters. Moreover, no two of these filters can belong to the same prefix-reorderable classifier, because their $\text{exact}(F_i)$ sets are unequal due to different "don't care" positions in the first halfs of the filters. Assigning to each filter a different action, we get a classifier without redundant rules that cannot be prefix-reordered. ■

E.g., for the following classifier $\mathcal{K}$ with $w = 8$ we get six different combinations of two $*$ and two 1 bits in the left half:

$\mathcal{K}$	#1	#2	#3	#4	#5	#6	#7	#8	*Action*
R_1	$*$	$*$	1	1	0	0	0	0	A_1
R_2	$*$	1	$*$	1	0	0	0	1	A_2
R_3	$*$	1	1	$*$	0	0	1	0	A_3
R_4	1	$*$	$*$	1	0	0	1	1	A_4
R_5	1	$*$	1	$*$	0	1	0	0	A_5
R_6	1	1	$*$	$*$	0	1	0	1	A_6

While instances such as in Theorem 4 are rare, it often happens in practice that a small number of "bad" rules form a hard example and result in a large number of representing groups. To allow our methods to cover non-prefix-reorderable instances, in this section we introduce "mixed" representations that implement a part of the original classifier in a traditional way (e.g., in TCAM). The assumption is, again, that we can look up both multigroup and traditional parts of a given classifier and return the matched rule with highest priority, so the mixed representation is again equivalent to $\mathcal{K}$. We assume that traditional representations are more expensive than multigroup ones. Therefore, our objective is to maximize the number of rules from a given classifier used in the multigroup part, while still limiting the total number of groups.

*Problem 2 (*MAXCOV*):* Given a classifier $\mathcal{K}$ and a constant $\beta > 0$, assign the largest possible subset of $\mathcal{K}$'s rules into at most β prefix-reorderable disjoint groups.

Note that both MINGR and MAXCOV problems do not change the rules themselves and do not require additional memory. The mixed setting allows us to exclude "bad" rules from consideration when constructing a chain partition. To exclude an element E from $\text{exact}(\mathcal{K})$ we must move every rule R with $\text{exact}(R) = E$ to a traditional representation. Denoting a set of all such rules as $\mathcal{K}[E]$, we can associate with every removal of E a cost $|\mathcal{K}[E]|$; the goal now is to minimize the total cost. Algorithm 6 (`max_coverage_partition`) essen-

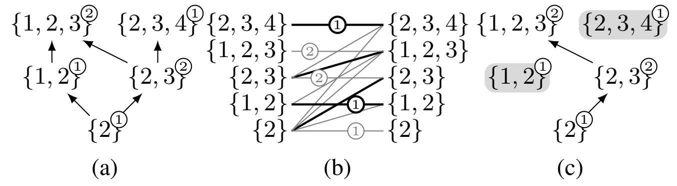


Fig. 3. `max_coverage_partition` example with $\beta = 1$: (a) ordered set $\mathrm{exact}(\mathcal{K})$; (b) minimal weighted matching with additional edges; (c) resulting chains; shaded regions show excluded rule subsets.

tially follows the same logic as `min_group_partition`, but `max_coverage_partition` instead of a maximal matching uses a minimal weight matching of a fixed cardinality and introduces additional weighted edges on line 5. The cardinality $|M|$ of the matching determines the number of subgroups ($\beta = |\mathrm{exact}(\mathcal{K})| - |M|$). This lets MINGR minimize the number of groups but MAXCOV simply bounds it while another objective is being optimized.

Algorithm 6 `max_coverage`$(\mathcal{K}, \beta)$

```
 1: $G \leftarrow$ weighted bipartite graph, $V(G) = \emptyset$, $E(G) = \emptyset$
 2: for $E \in \mathrm{exact}(\mathcal{K})$ do
 3:     $G$.add_left_vertex($E_L$)
 4:     $G$.add_right_vertex($E_R$)
 5:     $G$.add_edge($E_L, E_R$, weight $= |\mathcal{K}[E]|$)
 6: for $E, E' \in \mathrm{exact}(\mathcal{K})$ do
 7:     if $E \subset E'$ then
 8:         $G$.add_edge($E_L, E'_R$, weight $= 0$)
 9: $M \leftarrow$ min_weight_matching($G, |\mathrm{exact}(\mathcal{K})| - \beta$)
10: $\mathcal{K}' \leftarrow \{R \in \mathcal{K} : (\mathrm{exact}(R)_L, \mathrm{exact}(R)_R) \in M\}$
11: return $\mathcal{K}'$
```

Example 7: Fig. 3 shows `max_coverage_partition` on a classifier $\mathcal{K}$ from Section IV-A with $\beta = 1$; numbers of rules with each $\mathrm{exact}(R)$ are shown in top right corners (Fig. 3a). In the bipartite graph on Fig. 3b, `max_coverage_partition` adds weighted edges connecting each E with itself in the other part, with weight equal to the number of edges. Weighted edges in the minimal partition correspond to removed rule subsets, and the rest form the resulting chain (Fig. 3c).

Theorem 5: Algorithm `max_coverage_partition` produces an optimal solution for the MAXCOV problem in time $O(|\mathrm{exact}(\mathcal{K})|^2|\mathcal{K}| + |\mathcal{K}|^2 w)$.

Proof: Correctness of `max_coverage_partition` follows from a one-to-one correspondence between solutions $\mathcal{E}$ for the MAXCOV problem and weighted matchings $M_{\mathcal{E}}$ of size $|\mathrm{exact}(\mathcal{K})| - \beta$ in the bipartite graph with additional edges, and that $\sum_{E \in \mathrm{exact}(\mathcal{K}) \setminus \mathcal{E}} |\mathcal{K}[E]| = w(M_{\mathcal{E}})$. This fact is clear by construction:

- every matching of size $|\mathrm{exact}(\mathcal{K})| - \beta$ corresponds to β chains since in every chain, the number of vertices equals the number of edges plus one (in particular, a chain of size one corresponds to an unmatched vertex);
- by minimizing the weight of the matching, we minimize the total number of excluded rules.

Time complexity is similar to `min_group_partition`, but now `max_matching` is replaced with `min_weight_matching`; applying the augmenting paths approach, we get $|\mathrm{exact}(\mathcal{K})|^2|\mathcal{K}|$ complexity since the maximal flow is bounded by $|\mathcal{K}|$, and there are $|\mathrm{exact}(\mathcal{K})|$ vertices. ■

Note also that an optimal algorithm for MAXCOV can be used to find an optimal solution for the MINGR problem, with an extra factor of $\log(|\mathcal{K}|)$ for a binary search on β.

V. Non-prefix-reorderable classifiers with extra memory

So far, all proposed representations did not change the number of rules or rules themselves. However, in many cases we can improve prefix reorderability by expanding some $*$ bits into all possible combinations of 0s and 1s.

Example 8: Consider again the non-prefix-reorderable classifier from Example 3. We can cover only one rule with a single prefix-reorderable group, but by expanding the $*$ bit in R_2 we get the following prefix-reorderable classifier:

$\mathcal{K}$	#1	#2	*Action*
R_1	0	$*$	A_1
R_2	1	1	A_2
R_3	0	1	A_3

Now R_3 is covered by R_1 and can be removed, so the end result is a single group covering all rules.

This example shows that sometimes it is worthwhile to trade a modest increase in memory footprint to cover more rules by prefix-reorderable groups. Naturally, to avoid exponential memory blowup one should control the expansion process, so we incorporate into the MAXCOV problem a constraint m on the number of expanded bits per rule. In the following definition, $\mathcal{K}^*$ denotes an expanded version of $\mathcal{K}'$.

*Problem 3 (*MAXCOV-*m):* Given a classifier $\mathcal{K}$, a constant $\beta > 0$, and a maximal number of rules M, find the largest subset $\mathcal{K}' \subseteq \mathcal{K}$ and a multigroup classifier $\mathcal{K}^*$ equivalent to $\mathcal{K}'$ with $|\mathcal{K}^*| \leq M$ such that $\mathcal{K}^*$ can be split into at most β prefix-reorderable groups.

In what follows we present a heuristic for MAXCOV-m. The general idea is to start with a solution for the MAXCOV problem (for some β) and then incorporate some of the extra rules into the prefix-reorderable groups by applying *bit expansion*. Specifically, we take a rule R (e.g., $((0 * 1 *), A)$) and a set of bit indices (e.g., $\{2\}$), and replicate R, replacing $*$ bits at given indices with all possible combinations of 0s and 1s: $R' = ((0\ 0\ 1\ *), A)$, $R'' = ((0\ 1\ 1\ *), A)$. We denote this procedure by `expand`$(\mathcal{K}, B)$, which applies bit expansion to all bits with indices from B in every rule in $\mathcal{K}$.

To understand how bit expansion can be useful for solving the MAXCOV-m problem, consider $\beta = 1$. Assume that `max_coverage_partition`$(\mathcal{K}, 1)$ produced a maximum prefix-reorderable group $\mathcal{K}' \subseteq \mathcal{K}$, so by Theorem 1 $\mathrm{exact}(\mathcal{K}') = E_1 \subseteq E_2 \ldots \subseteq E_k$. We denote the "extra" rules represented in a traditional way as $\mathcal{K}_t = \mathcal{K} \setminus \mathcal{K}'$. Prefix-reorderability is determined solely by the structure of $\mathrm{exact}(\mathcal{K}')$, so we can limit our consideration to $\mathcal{K}_t[E] = \{R \in \mathcal{K}_t : \mathrm{exact}(R) = E\}$. To add some $\mathcal{K}_t[E]$ to the chain $\mathrm{exact}(\mathcal{K}')$, we need to expand some bits in $\mathcal{K}_t[E]$ and/or possibly in $\mathcal{K}[E_i]$ so that the result is still prefix reorderable.

Algorithm 7 `expand_and_fit`$(\mathcal{K}', \mathcal{K}_t, \delta)$

```
1: $\mathcal{E} \leftarrow$ sort $\mathrm{exact}(\mathcal{K}_t)$ by $1/|\mathcal{K}_t[E]|$
2: for $E \in \mathcal{E}$ do
3:     $(E_1 \subseteq \ldots \subseteq E_m) \leftarrow$ min_chain_partition$(\mathcal{K}')$
4:     $i^* \leftarrow \max\{i : E \not\subseteq E_i\}$
5:     $E_\cup \leftarrow E \cup E_{i^*}$
6:     if $|E_\cup \setminus E| > \delta$ or $\#\mathrm{ex}(E_{i^*}) + |E_\cup \setminus E_{i^*}| > \delta$ then
7:         continue
8:     $\#\mathrm{ex}(E_\cup) \leftarrow \max(|E_\cup \setminus E|, \#\mathrm{ex}(E_{i^*}) + |E_\cup \setminus E_{i^*}|, \#\mathrm{ex}(E_\cup))$
9:     $\mathcal{K}' \leftarrow (\mathcal{K}' \setminus \mathcal{K}'[E_{i^*}]) \cup$ expand$(\mathcal{K}'[E_{i^*}], E_\cup) \cup$ expand$(\mathcal{K}_t[E], E_\cup)$
10:    $\mathcal{K}_t \leftarrow \mathcal{K}_t \setminus \mathcal{K}_t[E]$
11: return $(\mathcal{K}', \mathcal{K}_t)$
```

For example, if $E \not\subseteq E_k$ (the maximal element of $\mathrm{exact}(\mathcal{K}')$), we merge $\mathcal{K}_t[E]$ and $\mathcal{K}'[E_k]$ into $\mathcal{K}'[E_k \cup E]$ by replacing them with `expand`$(\mathcal{K}_t[E], E_k \cup E)$ and `expand`$(\mathcal{K}'[E_k], E_k \cup E)$.

The algorithm `expand_and_fit` (Algorithm 7) greedily selects $\mathcal{K}_t[E]$ to merge into a $\mathcal{K}'$ so that $|\mathcal{K}_t[E]|$ is maximized, since the less rules are left to a traditional representation (the subset $\mathcal{K}_t$) the better. To guarantee that bit expansion will not increase memory above M, we limit the number of bits that may be expanded in each rule by $\delta = \log_2(M/|\mathcal{K}|)$; the algorithm keeps track of already expanded bits from previous iterations in $\mathcal{K}'$ using $\#\mathrm{ex}$. The complexity of `expand_and_fit` without calls to `expand` and `min_chain_partition` is $O(w \cdot |\mathrm{exact}(\mathcal{K}_t)| \cdot \log|\mathrm{exact}(\mathcal{K}')|)$. An actual implementation of the algorithm can postpone `expand` until the end by operating on pairs $(E, |\mathcal{K}'[E]|)$ instead of actual rule sets; moreover, the chain partition is usually known from from the preceding call to `max_coverage_partition`.

Example 9: Consider the classifier $\mathcal{K}$ from Example 5 and let $\beta = 1$ and $\delta = 1$. The maximum prefix-reorderable classifier $\mathcal{K}'$ is $\mathcal{K}_1$ and $\mathcal{K}_t$ is $\mathcal{K}_2$. We have $\mathrm{exact}(\mathcal{K}') = (E_1 = \{2\}, E_2 = \{1,2\}, E_3 = \{1,2,3\})$; and $\mathrm{exact}(\mathcal{K}_t) = (\{2,3\}, \{2,3,4\})$. First, `expand_and_fit` considers $E = \{2,3\}$; the largest E_i not containing E is $E_2 = \{1,2\}$, so $i^* = 2$. At this point, the check at line 6 succeeds, and $\mathcal{K}'$ gets augmented with `expand`$(\{R_5, R_6\}, \{1,2,3\})$. At line 6, $\mathrm{exact}(\mathcal{K}')$ does not change but $\#ex(\{1,2,3\})$ is set to 1. Next, algorithm considers $E = \{2,3,4\}$ and selects $i^* = 3$. Here $|E_3 \setminus E_\cup| = 1$, but now $\#\mathrm{ex}(E_3) = 1$, and the check at line 6 fails. The end result is the following:

$\mathcal{K}'$	#1	#2	#3	#4	*Action*
R_1	0	0	0	$*$	A_1
R_2	0	0	1	$*$	A_2
R_4	0	0	$*$	$*$	A_4
R_{50}	0	0	1	$*$	A_5
R_{51}	1	0	1	$*$	A_5
R_{60}	0	1	0	$*$	A_6
R_{61}	1	1	0	$*$	A_6
R_7	$*$	0	$*$	$*$	A_7

Note the redundancy in R_{50} that resulted from bit expansion.

VI. How to cut down classification width

We have already introduced several ways to represent ternary bit strings with general priorities on commodity LPM infrastructures. So far we assumed that classification width w of the classifiers is at most the classification width w_{LPM} of an implementing LPM infrastructure. In reality, w can be larger than supported by commodity LPM infrastructures (32 or 128 bits), so it is important to reduce classification width. The work [13] already introduced a structural property of classifiers called *rule disjointness* to tackle exactly this problem. A classifier is *rule-disjoint* iff all its rules are pairwise disjoint. If a given classifier $\mathcal{K}$ is still rule-disjoint on a subset of bit indices $B \subset \{1, \ldots, w\}$, the underlying lookup can be based only on these B bits, and then only a false-positive check is required on the remaining $\{1, \ldots, w\} \setminus B$ bits. Due to disjointness, only a single rule can be matched, so the false-positive check is not a lookup but just a bitwise comparison of bits from $\{1, \ldots, w\} \setminus B$ in the header and rule matched in the $\mathcal{K}^B$ classifier.

Example 10: Consider the following rule-disjoint classifier:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	1	$*$	0	A_1
R_2	1	$*$	0	$*$	A_2

Note that only a single bit is enough to keep an equivalent classifier $\mathcal{K}^B$ rule-disjoint, i.e., for $B = (1)$ we have

$\mathcal{K}^B$	#1	*Action*
R_1'	0	`false_positive_check`(R_1)
R_2'	1	`false_positive_check`(R_2)

Similarly to prefix reorderability, requiring rule disjointness on the entire classifier can be too restrictive, so multigroup representations are natural for both characteristics. In particular we want to have the width-bounded versions of the problems that we have defined earlier (e.g., MINGR or MAXCOV). We will not present here definitions for all of them (since they have similar changes) and limit ourselves to the following:

*Problem 4 (*MAXCOV-*w):* Given a classifier $\mathcal{K}$ and two positive numbers w_{LPM} and β, find a maximal subset of $\mathcal{K}$'s rules that can be assigned to at most β groups, where each group is prefix-reorderable and rule-disjoint on at most w_{LPM} bits.

At this point we need to understand the effects rule disjointness and prefix reorderability have on each other. The following two observations show that there is no interference.

*Observation 3 (*LPM-RD*):* If a classifier $\mathcal{K}^B$ is prefix-reorderable, $\mathcal{K}^{B'}$ is also prefix-reorderable for any $B' \subseteq B$.

*Observation 4 (*RD-LPM*):* Reordering of bit indices and expansion of $*$ bits preserve rule disjointness.

Thus, it is safe to combine transformations reducing classification width (RD-step) with those that aim for prefix reorderability (LPM-step) in any order. This is very important because it allows us to decouple them from each other. The only remaining question is in which order should we combine these step: RD-LPM or LPM-RD? It turns out that there is no definite answer to this question.

Example 11: First, consider the hard classifier instance from Theorem 4 for $w = 4$:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	$*$	1	0	0	A_1
R_2	1	$*$	0	1	A_2

Assume that $w_{\text{LPM}} = 2$, and we intend to minimize the number of groups as in the PMGR problem. The RD-LPM approach will produce one group because it will first switch to $\mathcal{K}^{(3,4)}$, which is prefix-reorderable. But the LPM-RD approach will first split the rules into two groups, getting two groups instead of one.

On the other hand, consider the following example:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	1	0	$*$	A_1
R_2	$*$	0	1	0	A_2
R_3	1	0	$*$	$*$	A_3
R_4	$*$	1	0	$*$	A_4

Here the LPM-RD approach produces exactly two groups: $\{R_1, R_3\}$ and $\{R_2, R_4\}$, which are both order-independent on the first and second bit respectively. Although RD-LPM may use the same solution as LPM-RD, it is not required to do so, and it may select the following two groups instead: $\{R_1, R_2\}$ and $\{R_3, R_4\}$. Each of those groups will require two subgroups to be represented on an LPM infrastructure producing four groups in the final representation. Thus, for the MINGR-w problem different approaches can work better for different instances of classifiers.

In the MAXCOV-w problem the objective is different: complete coverage is not required, but the number of rules is limited. The approach we suggest here is to greedily take the largest possible subset which is both rule disjoint and prefix-reorderable. It can be found in two steps: first, we find a rule disjoint group using a greedy algorithm from [20]; second, we run MAXCOV with $\beta = 1$ to find a maximal prefix-reorderable subset. By Observation 4, the resulting group possesses both properties. We call this algorithm RD-MC. Another version of this algorithm, RD-MC-EXP, runs an additional `expand_and_fit` step after MAXCOV.

Another curious interaction between LPM and RD happens when we vary w_{LPM} (infrastructure limits) in the RD-LPM approach. On one hand, if RD reduces the number of bits, it is easier for LPM to find a prefix-reorderable decomposition of rules (e.g., set $w_{\text{LPM}} = w/2$ in Theorem 4 and get a $|\mathcal{K}|$-fold decrease in the number of groups). On the other hand, RD itself may require a large number of groups, and the lower we set w_{LPM}, the more groups we will have. At the extreme, the following classifier is prefix-reorderable, but RD yields 4 groups even if $w_{\text{LPM}} = 4$:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	0	0	0	0	A_1
R_2	0	0	0	$*$	A_2
R_3	0	0	$*$	$*$	A_3
R_4	0	$*$	$*$	$*$	A_4

VII. Dynamic updates

Another recurring theme in classifier representations is the support of dynamic updates. We have already mentioned two major service categories: traffic forwarding and service policies that represent economic models and traffic signatures designed in advance (e.g., quality of service, access control, firewall). Dynamic updates are not too important for the second category; offline computation can be valid in most cases. What if we still do need dynamic updates? Deletions and insertions that maintain groupwise representations are simple.

	Rules	$l = 16$			$l = 24$			$l = 32$			$l = 64$			$l = 104$		
	$\beta =$	5	10	20	5	10	20	5	10	20	5	10	20	5	10	20
acl1	65331	85.1	96.0	99.8	93.5	97.8	99.9	93.8	97.7	99.7	92.1	98.6	98.7	72.6	86.2	95.1
acl2	96334	21.9	33.7	49.8	37.7	53.1	73.0	43.1	59.4	80.4	70.5	78.4	82.6	42.9	63.6	83.3
acl3	90208	56.4	77.5	85.8	73.9	89.9	97.7	75.7	91.3	97.6	72.9	82.1	86.2	49.8	67.2	83.5
acl4	84522	30.0	45.8	63.8	48.1	65.6	82.8	56.3	75.5	88.1	65.9	76.0	82.1	50.6	67.1	82.5
acl5	52240	13.5	25.8	46.4	33.5	54.1	72.4	47.1	68.4	85.4	88.4	98.2	100	76.0	90.4	98.7
fw1	165876	28.3	38.3	53.7	30.5	38.2	51.9	29.7	37.0	48.7	44.4	55.4	60.9	32.1	44.3	61.9
fw2	92730	61.5	81.5	83.2	84.3	86.3	87.9	80.9	87.5	92.7	40.5	46.3	49.3	73.7	89.8	98.5
fw3	128624	22.6	30.9	44.6	21.9	32.5	48.9	30.7	42.7	59.2	37.1	54.5	72.3	41.2	55.8	70.7
fw4	284747	28.0	41.0	48.1	51.2	53.0	54.6	51.6	62.9	74.7	13.4	17.4	23.3	31.5	44.8	61.4
fw5	106201	34.6	42.8	47.0	36.0	38.2	38.4	31.6	33.7	35.0	33.2	41.9	49.4	42.0	58.6	78.2
ipc1	67620	77.4	94.8	99.6	89.8	98.8	99.8	90.7	98.9	99.8	76.7	76.8	76.8	61.2	75.6	87.3
ipc2	50000	99.8	100	100	99.9	100	100	100	100	100	66.0	78.4	86.7	95.6	100	100

TABLE I
LPM RESULTS FOR DIFFERENT VALUES OF l AND β, COVERAGE %.

If a new rule cannot be added to an existing group, or the traditional part is full, the multigroup part can be recomputed. Implementation of these cases is straightforward, so due to space constraints we do not go into further details here; the update procedure is similar to the one proposed in [13, Section 7.2]. There are two important factors that make a specific representation feasible: update resolution (that defines which part of a classifier is affected) and update frequency. Since there are cases when recomputation of a multigroup representation is required, we prefer to limit applicability of our representations only to the second group of services.

VIII. Evaluation

		Memory expansion by 4 bits								
		$l = 16$			$l = 24$			$l = 32$		
	Rules	β =5	10	20	β =5	10	20	β =5	10	20
acl1	65331	85.1	96.1	99.8	93.7	97.1	99.7	93.9	97.7	99.7
acl2	96334	22.6	34.2	50.0	39.5	54.5	74.5	45.5	61.3	82.4
acl3	90208	57.5	78.4	86.2	74.6	91.2	98.9	77.6	92.6	98.5
acl4	84522	30.1	46.1	64.3	49.4	66.8	83.8	59.1	78.5	90.3
acl5	52240	13.5	25.8	46.4	33.6	54.3	73.1	47.7	69.1	86.7
fw1	165876	28.4	38.4	53.9	30.7	38.5	50.9	29.6	36.5	48.4
fw2	92730	61.5	81.5	83.2	85.6	90.4	93.6	82.9	87.7	91.7
fw3	128624	22.8	31.2	44.8	22.1	32.8	49.4	31.8	43.3	59.4
fw4	284747	28.0	41.1	48.2	48.7	50.4	52.0	62.2	73.4	84.7
fw5	106201	34.6	42.8	47.0	36.0	38.2	38.4	32.2	33.9	35.1
ipc1	67620	77.6	96.2	99.4	89.1	98.4	99.5	91.1	99.3	99.9
ipc2	50000	99.8	100	100	100.0	100.0	100.0	100.0	100	100
		Memory expansion by 8 bits								
		$l = 16$			$l = 24$			$l = 32$		
	Rules	β =5	10	20	β =5	10	20	β =5	10	20
acl1	65331	85.1	96.1	99.8	93.7	97.1	99.7	93.9	97.9	99.9
acl2	96334	22.7	34.7	50.6	39.3	54.5	74.5	46.2	62.1	83.6
acl3	90208	57.5	78.5	86.6	75.3	90.8	97.3	78.3	95.4	99.0
acl4	84522	30.2	46.3	64.8	50.2	68.4	85.5	59.6	78.9	91.1
acl5	52240	13.5	25.8	46.4	33.6	54.3	73.1	47.7	69.1	86.7
fw1	165876	28.4	38.4	53.9	31.3	39.6	53.5	30.6	38.7	52.3
fw2	92730	61.5	81.5	83.2	94.9	98.9	99.9	94.2	99.3	100.0
fw3	128624	22.8	31.2	44.8	22.2	32.8	49.3	31.8	43.3	59.4
fw4	284747	28.0	41.1	48.2	48.7	50.4	52.0	62.4	74.2	86.3
fw5	106201	34.6	42.8	47.0	36.3	38.7	38.9	33.7	37.6	38.5
ipc1	67620	77.1	95.5	99.2	89.8	98.4	99.0	91.8	99.3	99.8
ipc2	50000	99.8	100	100	100.0	100.0	100.0	100.0	100	100

TABLE II
LPM RESULTS WITH BIT EXPANSION FOR DIFFERENT VALUES OF l AND β.

To validate the proposed approach, we have run simulations on classifiers from the Classbench benchmark [14] generated with real parameters. Each classifier in the tables below has about 50,000 rules. The generated classifiers contained range-based fields, so in order to convert and operate on the entire multifield classifier we used the commonly used SRGE encoding scheme based on Gray coding [21]. Our code for processing and evaluating the classifiers is available at GitLab [22], [23]. We did not measure lookup times since our approach serves as an abstract layer to find bit identities for lookup, independently of the underlying LPM infrastructure. Our evaluations are intended to validate the feasibility of the proposed approach, so we also did not compare our implementations with, for instance, software-based solutions intended to reduce size. We used four algorithms in the evaluations: (1) RD is a greedy algorithm from [13] that iteratively removes bits that represent unique differences for the smallest number of rule pairs, breaking ties by the share of $*$ bits; (2) RD_{exact} is the same but removes only non-exact bits and does not stop until all bits are exact [13]; (3) RD-MC, our main heuristic, and (4) RD-MC-EXP have been discussed in Section VI.

Table I shows the results of our proposed heuristics without extra memory. We have covered five values of l: $l = 16$, $l = 24$, $l = 32$, and $l = 64$ for the composition of RD and LPM, and $l = 104$ which corresponds to "pure" LPM with no width limit, and have computed the number of rules covered by top β groups for three different practical values of β: 5, 10, and 20. Note that for small values of l, the wider are the filters the larger the groups become and the fewer groups are needed; this is a property mostly inherited from the RD part since it is easier to find disjoint rules when their filters are wider. On the other hand, for extra large l it becomes significantly harder to preserve prefix reorderability, and we see that by the time we reach l where RD, the tension between these two effects becomes evident: in some examples pure LPM loses to RD-LPM composition for $l = 32$; in others, vice versa. This effect is also illustrated on Fig. 4 that shows rules coverage in more detail: we see that the coverage is far from monotone in l. Another effect is that depending on the LPM infrastructure implementation, wider groups (larger w_{LPM}) may use significantly, sometimes exponentially more memory in order to decrease lookup time [8], [9], so in reality a larger number of more narrow groups may be preferable to a smaller number of wider groups. In Table I we see that this kind of tradeoff is also possible: e.g., for *fw2* 10 groups with $l = 24$ cover more rules than 5 groups with $l = 32$, and so on. Table II shows the result with bit expansion for 4 and 8 bits. We see that this new allotted memory has virtually no effect in our example of practical classifiers at the level of β., e.g., The effect here is as follows: algorithm `min_pmgr_mstep` attempts to minimize the antichain size while trying to keep memory requirements as low as possible. This means that, in practice, `min_pmgr_mstep` does indeed significantly reduce the total number of groups, but does so by merging small groups, and they do not appear in the top 5-10 groups shown in Table II.

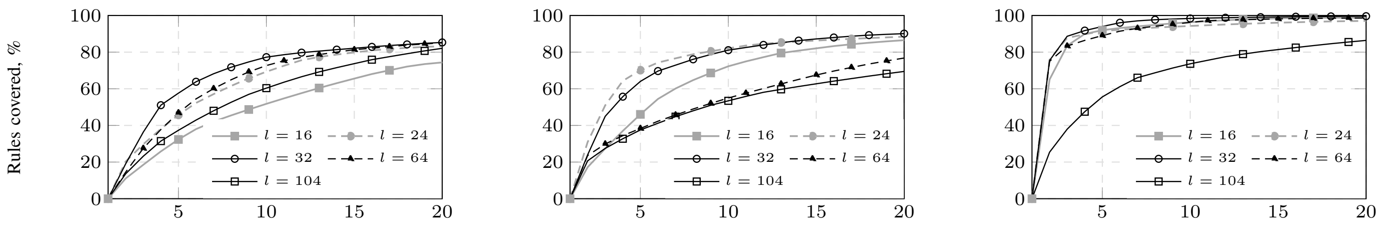
Figure 5 shows a comparison of the four algorithms in terms of rule coverage for three characteristic examples. Again, we see that the relative performance of algorithms highly depends on the specific instance, but the RD-MC-EXP heuristic proves to be most stable overall, across all examples. In all cases, we see that the proposed algorithms provide huge practical improvements in terms of TCAM size, covering, in most cases, a vast majority of the input classifier with but a few groups.

IX. Implementation in P4

We have implemented the proposed representations in the commonly used domain-specific programming language P4 that implements match-action pipelines [15]. We used the *Behavioral Model Version 2* (BMV2) framework that allows to implement a P4-programmable architecture as a virtual software switch; BMV2 is coming with the *simple_switch* target architecture [24]. However, the problem with P4 is that one has to specify lookup tables, headers, etc. in advance, and the actual table content is added separately at "run time". Fixing bit indices implementing rule- disjointness and prefix reorderability before the actual table content is provided can significantly degrade efficiency for some instances of classifiers. To address this limitation, we extended the BMV2 table filling interface with the optimization engine that is target-independent; the only prerequisite is a P4 intermediate representation and the actual data. The engine replaces the original lookup table and its key structure specified in P4 with the equivalent representation based on data content, given a target specific maximal number of groups. We have released our implementation in open source [25].

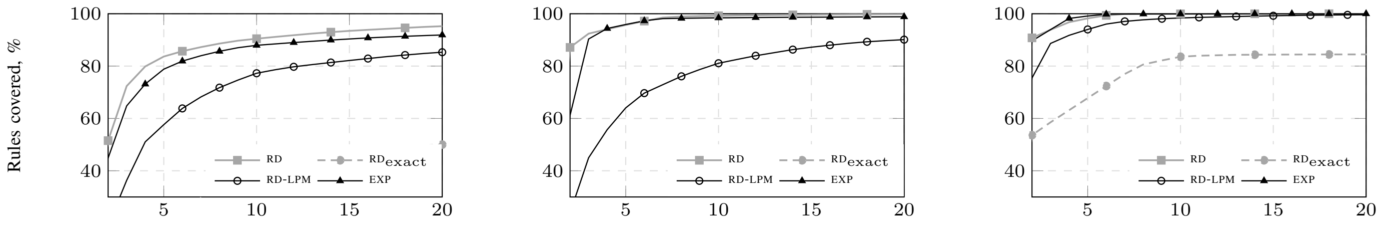
X. Related Work

Research towards efficient implementations of packet classifiers falls into two major categories: algorithmic solutions (usually software based) and TCAM-based solutions; comprehensive surveys can be found in [26], [1]. Algorithmic solutions mainly rely on one of three techniques: decision trees, hashing, or coding-based compression. The works [27], [12] suggest how to partition the multi-dimensional rule space. Possible matching rules are found by tracing a path in a decision tree. Techniques to balance the partition in each node exist, but rule replication often cannot be avoided; see a related approach in [28]. There is an inherent tradeoff between space and time complexities in these approaches. Song and Turner's ABC algorithm for filter distribution offers higher throughput with lower memory overhead and can tune the implementation for better time complexity or better space complexity [29]. The works [30], [31] discuss hash-based solutions to match a packet to its possible matching rules. Efficient coding-based representations are shown in [32], [33]. TCAMs have no native support for range representations, so range encoding encompass an important direction in this domain [21], [34], [11]. Different approaches have been described to reduce number of entries: removing redundancies [35], [36], applying block permutations in the header space [37], transformations [36], [38], [39]. In particular [13], [40], [41] considered representations based on rule disjointness that we compose with prefix-reorderability to cut down classification width.



(a) *acl2*: β (b) *fw3*: β (c) *ipc1*: β

Fig. 4. A comparison of group sizes for the `min_group_partition` algorithm with different l: characteristic examples.



(a) *acl2*: β (b) *fw3*: β (c) *ipc1*: β

Fig. 5. A comparison of the algorithms with $l = 32$: characteristic examples.

XI. Conclusion

We have proposed alternatives that allow to implement ternary bit strings with general priorities on commodity LPM infrastructure, completely transparently to its internals, while removing or significantly reducing the need in TCAM. Our approach is built around *prefix reorderability*, a novel structural property of classifiers. We extend our results to classifiers of arbitrary width by composing prefix reorderability with rule disjointness. Feasibility of the proposed representations is supported by evaluations on practical classifiers; in addition, we release a P4 implementation of our transformations.

Acknowledgments: This work was supported by the Russian Science Foundation grant no. 17-11-01276, "Networking and distributed systems and algorithms and related fundamental problems".

References

[1] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.

[2] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *SIGCOMM*, 1999, pp. 147–160.

[3] K. Huang, L. Ding, G. Xie, D. Zhang, A. X. Liu, and K. Salamatian, "Scalable tcam-based regular expression matching with compressed finite automata," in *ANCS*, 2013, pp. 83–93.

[4] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng, "Fast regular expression matching using small TCAM," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 94–109, 2014.

[5] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *ANCS*, 2011, pp. 24–35.

[6] "Netlogic Microsystems. Content addressable memory," http://www.netlogicmicro.com.

[7] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *ISPASS*, 2006, pp. 120–129.

[8] "Cisco CRS forwarding Processor Cards," http://www.cisco.com/c/en/us/products/collateral/routers/ carrier-routing-system/datasheet-c78-730790.pdf.

[9] "The Cisco flow processor: Cisco's next generation network processor solution overview," http://www.cisco.com/c/en/us/products/collateral/routers/ asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html.

[10] F. Baboescu and G. V. G., "Scalable packet classification," in *SIGCOMM*, 2001, pp. 199–210.

[11] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM*, 1998, pp. 191–202.

[12] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput," in *SIGCOMM*, 2010, pp. 207–218.

[13] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *SIGCOMM*, 2014, pp. 15–26.

[14] "ClassBench: A packet classification benchmark," http://www.arl.wustl.edu/classbench/.

[15] "P4$_{16}$ language specification," https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[16] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge: Cambridge university press, 1990. [Online]. Available: http://opac.inria.fr/record=b1077513

[17] R. P. Dilworth, "A decomposition theorem for partially ordered sets," *Annals of Mathematics*, vol. 51, no. 1, pp. 161–166, 1950. [Online]. Available: http://www.jstor.org/stable/1969503

[18] D. R. Fulkerson, "Note on dilworth's decomposition theorem for partially ordered sets," *Proceedings of the American Mathematical Society*, vol. 7, no. 4, pp. 701–702, 1956. [Online]. Available: http://www.jstor.org/stable/2033375

[19] J. E. Hopcroft and R. M. Karp, "A n5/2 algorithm for maximum matchings in bipartite," in *SWAT*, 1971, pp. 122–125.

[20] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 1251–1264, 2016.

[21] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using Gray coding," *IEEE Trans. Computers*, vol. 61, no. 1, pp. 18–30, 2012.

[22] "Optimization library for simulations." [Online]. Available: https://gitlab.com/pschuprikov/classifiers-lib/tree/ton-lpm

[23] "Code for simulations." [Online]. Available: https://gitlab.com/pschuprikov/lpm/tree/ton-lpm

[24] "The Bmv2 simple switch target," https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.

[25] "Code for simulations," https://github.com/icnp2017/submission.

[26] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[27] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, 2000.

[28] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *INFOCOM*, 2010.

[29] H. Song and J. S. Turner, "ABC: Adaptive binary cuttings for multidimensional packet classification," *IEEE/ACM Trans. Netw.*, vol. 21, no. 1, pp. 98–109, 2013.

[30] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *ACM/IEEE ANCS*, 2006.

[31] Y. Kanizo, D. Hay, and I. Keslassy, "Optimal fast hashing," in *Infocom*, 2009.

[32] A. Korösi, J. Tapolcai, B. Mihálka, G. Mészáros, and G. Rétvári, "Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously," in *ICNP*, 2014, pp. 332–343.

[33] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables for datacenter scalability," *IEEE JSAC*, vol. 32, no. 1, pp. 138–151, 2014.

[34] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *Infocom*, 2013.

[35] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *Infocom*, 2008.

[36] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and speed tradeoffs in TCAM hierarchical packet classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.

[37] R. Wei, Y. Xu, and H. J. Chao, "Block permutations in Boolean space to minimize TCAM for packet classification," in *Infocom Mini-Conference*, 2012.

[38] K. Kogan, S. I. Nikolenko, P. T. Eugster, and E. Ruan, "Strategies for mitigating TCAM space bottlenecks," in *HOTI*, 2014, pp. 25–32.

[39] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 237–250, 2011.

[40] K. Kogan, S. I. Nikolenko, P. T. Eugster, A. Shalimov, and O. Rottenstreich, "FIB efficiency in distributed platforms," in *ICNP*, 2016, pp. 1–10.

[41] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Bérczi-Kovács, and A. Shalimov, "How to represent ipv6 forwarding tables on ipv4 or mpls dataplanes," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016, pp. 521–526.