How to Represent IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes

Sergey I. Nikolenko*, Kirill Kogan†, Gábor Rétvári‡, Erika R. Bérczi-Kovács§ and Alexander Shalimov¶
*Steklov Institute of Mathematics at St. Petersburg, NRU Higher School of Economics, St. Petersburg
†IMDEA Networks Institute, Madrid
‡MTA-BME Information Systems Research Group, Budapest University of Technology and Economics
§Department of Operations Research, Eötvös Loránd University, Budapest
¶Lomonosov Moscow State University

***Abstract*—The Internet routing ecosystem is facing substantial scalability challenges on the data plane. Various "clean slate" architectures for representing forwarding tables (FIBs), such as IPv6, introduce additional constraints on efficient implementations from both lookup time and memory footprint perspectives due to significant classification width. In this work, we propose an abstraction layer able to represent IPv6 FIBs on existing IP and even MPLS infrastructure. Feasibility of the proposed representations is confirmed by an extensive simulation study on real IPv6 forwarding tables, including low-level experimental performance evaluation.**

I. Introduction

Most methods that efficiently represent IP-based FIBs do not scale well to IPv6 [7], [19], [21], [24] due to its significantly larger 128-bit address width. It is hard to find IPv6-based FIB representations with comparable lookup time and memory requirements that are implemented in regular memory. This is a major reason why state-of-the-art solutions for IPv6 FIB are implemented in very expensive and power-hungry ternary content-address memories (TCAMs) [4], [22], which calls for more efficient software representations.

One recurring theme across IPv6 software FIB implementations is various forms of decision tree representations, like *prefix trees* [2], [6], [7], [18], [20], [21]. Unfortunately, prefix trees are inherently *sequential*, meaning that an longest prefix match (LPM) search involves multiple consecutive lookup steps, each step using the result of the previous one, and the total number of steps is bounded only by the width of the address space (128 in the case of IPv6). The reason is that a prefix tree can contain for any given IPv6 address multiple matching entries and our task is to find the most specific one LPM. Note that each step needs a separate random access to memory. There is an upper limit on the number of consecutive memory accesses that can be performed per packet at line-rate, and this imposes a strict budget on the number of levels any tree traversal can take. This is why most IPv4 FIB prefix tree representations are not extensible to IPv6 naively within the desired lookup time and memory requirements.

To address this limitation, IPv6 FIB representations can exploit the level of *parallelism* (how many parallel lookups can be done per packet at line-rate) that is an internal characteristic of each network element. A straightforward choice for parallelization of IPv6 FIB lookup would be classifying prefixes to groups, where each group would contain all prefixes of the same length [17], [23], [25], and perform lookups on each of the prefix groups simultaneously. Eventually, the match within the group of the longest prefix length is output as a result. This scheme is extremely appealing for implementations because lookup inside groups boils down to a simplistic exact match, where the key is taken as the initial bits of the address as specified by the group's prefix length, which can be easily done with a CAM or a hash. Unfortunately, the number of different prefix lengths, and hence the number of lookup threads needed to be executed in parallel, is too high in the IPv6 case, as most HW (except GPUs [10]) lacks the necessary number of independent cores.

This discussion brings us to the main question we address in this paper: how to find alternative decompositions for an IPv6 FIB, more efficient and more amenable to parallelization than by prefix length? The decomposition must be such that *(i)* lookups inside each group should be as simple as possible, preferably exact match, *(ii)* the number of groups must be reasonable, e.g., in the order of a couple of dozens, and *(iii)* the number of bits participating in classification in every group is significantly smaller than the original classification width (e.g., from 128 to 32 or even 13 bits) allowing representation of IPv6 FIBs on existing IP and MPLS infrastructure.

II. Notation

A *prefix classifier* $\mathcal{K}$ is an ordered set of rules $\mathcal{K} = (R_1, \ldots, R_N)$, where each rule R_i consists of a *prefix filter* F_i and a pointer to the corresponding *action* A_i. A filter F is an ordered set of w *bits* $F = \{f_0, f_1, \ldots, f_{w-1}\}$, each bit f_i taking the value 0, 1, or $*$ ("don't care"), and a *prefix filter* is such that all "don't care" bits appear in F only after a certain position p, i.e., $f_i = *$ if and only if $p \leq i < w$ for some $0 \leq p < w$. For instance, the filter $F_1 = (1\ 0\ *\ *)$ is a prefix filter while $F_2 = (*\ 0\ 0\ *)$ is not. A header H *matches* a filter F if for any bit of H the corresponding bit of F has the same value or $*$. For instance, the filter $F = (*\ 0\ 1\ 0)$ matches the headers $(0\ 0\ 1\ 0)$ and $(1\ 0\ 1\ 0)$, but not $(1\ 0\ 1\ 1)$. A header H matches a rule R_i if R_i's filter is matched. The rules have priorities represented by an ordering $\prec$; if a header matches both R_i and R_j for $R_i \prec R_j$, the action of rule R_i is applied. We shall use the convention that high priority rules precede

lower priority rules (appear at a lower indices) in the classifier: $R_i \prec R_j$ if and only if $i < j$. Then the longest-prefix-match semantics of IPv6 forwarding tables is implemented by ensuring that a more specific prefix rule always occurs lower than any overlapping less specific rule. We say that two filters F and F' *intersect* if there is at least one header that matches both F and F'; otherwise, F and F' are *disjoint*. A classifier whose any two filters are pairwise disjoint is called *disjoint* or *order-independent*. Two classifiers are *semantically equivalent* if every header leads to the same action in both.

Next, we introduce two operations on packet classifiers, namely, (width-)reduction and partitioning. Let B be an ordered set of bit indices, $B \subseteq \{0, \ldots, w-1\}$, referring to a subset of the bits in packet headers. For a header H, the (sub)header of H obtained by taking only bits of H with indices in B is called a *reduction of H to B*, denoted by H^B. Likewise, for a rule R (or a filter F) the *reduction of rule R* (*reduction of filter F*) to B, denoted by R^B (F^B, respectively), is the (sub)rule (subfilter) defined on the bit positions in B. Finally, for a classifier $\mathcal{K} = (R_1, \ldots, R_N)$, let $\mathcal{K}^B$ be the (sub)classifier obtained from $\mathcal{K}$ by replacing each rule R in the classifier by R^B. The notions of matching, intersection, and order-independence carry over to such subsequences.

Observe that any reduction of a prefix classifier is again a prefix classifier. Further notice that a reduced rule matches a larger set of packets than an underuced one (as certain, potentially valuable, bits in the filters are ignored), therefore a match in the reduced classifier does not immediately signal a match in the original classifier. Thus a match in a reduced classifier must always be validated using a false positive check, except for the case when only "don't care" bits are eliminated (like for prefix-length-based decompositions). We also introduce the notion of classifier partitions, as a division of the rules in a classifier into distinct groups of rules. Consider a classifier $\mathcal{K} = (R_1, \ldots, R_N)$ and a subset of the index set $C \subseteq \{1, \ldots, N\}$. Now, the classifier $\mathcal{K}_C = (R_i : i \in C)$ is called the *(classifier) group* of $\mathcal{K}$ defined by C. Given any partition $\mathcal{C} = \{C_1, \ldots, C_k\}$ of $\{1, \ldots, N\}$, where $\bigcup C_i = \{1, \ldots, N\}$ and $C_i \cap C_j = \emptyset$ whenever $i \neq j$, the *classifier partition* defined by $\mathcal{C}$ is given by exactly k classifier groups on $\mathcal{C}$: $\mathcal{K}_\mathcal{C} = \{\mathcal{K}_{C_1}, \ldots, \mathcal{K}_{C_k}\}$. Note that any partition of a prefix classifier is semantically equivalent with the original classifier, using the convention that from the matching rules of each group the output is chosen by longest prefix length.

III. Multi-group Representations

Intuitively, prefix-length-based decompositions enjoy an appealing structural property: *for every input each subclassfier contains at most one matching rule*. This property makes representation of every group simple since all prefixes in a group can be represented as exact values; this is *strong* order independence. Unfortunately, prefix-length-based decompositions can have too many groups (128 in the worst case for IPv6). One interesting question we tackle in this paper is the existence of structural properties in classifiers that would allow to consider prefixes with heterogeneous length to be mixed in the same group while still keeping the simplicity of representation. We argue that order-independent decompositions are a natural generalization of prefix-length-based decompositions in this context: given any classifier $\mathcal{K}$, it holds that $\mathcal{K}$ contains at most one matching rule for any header if and only if $\mathcal{K}$ is order-independent. This is a special type of order-independence that we call *weak*. This discussion motivates to limit our attention to the classifier decompositions that guarantee order-independence (weak or strong) inside each group on a subset of bit-identities that allows to mitigate time-space tradeoffs in representations of IPv6 FIBs.

Problem 1: Minimal weak (strong) order-independent reduction. Given a classifier $\mathcal{K}$ (not necessary order-independent) and a constant l, assign $\mathcal{K}$'s rules to a minimal number of disjoint groups such that different groups of prefixes can be based on subsets of at most l bit-identities, and each group is weak (strong) order-independent on these bits.

Finding a subset of bit-identities that keeps order-independence on these bits of order-independent (both weak and strong) classifier is already computationally hard for a single group (reduction from the minimum test set problem [8]).

In addition for a prefix classifier with N rules, this problem is not approximable within $(1-\epsilon)\ln N$ for any $\epsilon > 0$, if $NP \not\subset DTIME(N^{\log\log N})$ [3].

IV. Proposed Representations

A. Introduction and building blocks

Although finding exact minimal reductions is computationally hard, we propose several heuristics that can be used in practice. All proposed algorithms follow the same general scheme shown as Algorithm 1: one by one, they construct a group with some OneGroup procedure, remove it from the current set of prefixes, and then call OneGroup again until the stopping condition is satisfied, i.e., either all prefixes of a given classifier are already covered or the number of created groups has reached a given level of parallelism β. Another parameter is the bit *exact* that shows whether we are trying to construct groups with only exact values or groups where $*$ bits are allowed. The building block for our heuristics is the MaxOI procedure that finds, for a set of ranges $[s_i, f_i]$, $i = 1..n$, the maximal set of non-intersecting ranges; the greedy Algorithm 2 (earliest-deadline-first (EDF)) for this problem is optimal with $O(N \log N)$ time complexity [13].

Example 1: We consider the following running example for this section with original size $4 \times 5 = 20$ bits: $F_1 = 1001$, $F_2 = 1000$, $F_3 = 01**$, $F_4 = 11**$, $F_5 = 0***$. In this example, three different prefix lengths yield a decomposition into three groups with total size 13 bits.

Algorithm 1: MULTIGROUP

Data: Classifier $\mathcal{K}$, max. width l, parallelism β, *exact*

begin
- $i \leftarrow 0$, $\mathcal{T} \leftarrow \mathcal{K}$, $\mathcal{G} \leftarrow \emptyset$
- **while** $\mathcal{T} \neq \emptyset$ *and* $i < \beta$ **do**
  - $\mathcal{G}_i = \mathcal{K}_{C_i}^{B_i} \leftarrow$ ONEGROUP($\mathcal{T}$, l, *exact*)
  - $\mathcal{G} \leftarrow \mathcal{G} \cup \{\mathcal{G}_i\}$
  - $\mathcal{T} \leftarrow \mathcal{T} \setminus \mathcal{G}_i$
  - $i \leftarrow i + 1$
- **return** $\mathcal{G}$

Algorithm 2: MAXOI

Data: Set of prefixes $\mathcal{K} = \{F_1, \ldots, F_n\}$

begin
- represent $\mathcal{K}$ as set of ranges $F_i = [a_i, b_i]$
- sort $\mathcal{K}$ by upper bounds b_i, set $k \leftarrow 1$, $S \leftarrow \emptyset$
- **for** $i = 2, \ldots, n$ **do**
  - **if** $a[i] > b[k]$ **then**
    - $S \leftarrow S \cup \{F_i\}$, $k \leftarrow i$
- **return** S

Algorithm 3: ONEGROUP, the MINSIMILAR heuristic.

Data: $\mathcal{K}$, max. width l, *exact*

begin
- **for** every bit $i = 1, \ldots, k$ **do**
  - Compute $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
- Choose $B = \{b_1, \ldots, b_l\}$ of l bit identities with minimal S_i
- $C \leftarrow \mathcal{K}$
- **if** *exact* **then**
  - $C \leftarrow \{F \in C \mid * \notin F^B\}$
- **return** MAXOI($\mathcal{K}_C^B$)

Algorithm 4: ONEGROUP, the MAXPAIR heuristic.

Data: $\mathcal{K}$, max. width l, *exact*

begin
- **for** every bit $i = 1, \ldots, k$ **do**
  - compute $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
- Choose $B = \{b_1, \ldots, b_l\}$ of l bit identities with maximal D_i
- $C \leftarrow \mathcal{K}$
- **if** *exact* **then**
  - $C \leftarrow \{F \in C \mid * \notin F^B\}$
- **return** MAXOI($\mathcal{K}_C^B$)

Throughout this section, we use five IPv6 FIBs from the Internet2 project [11] for evaluation of our algorithms.

B. Weak order-independence

We now introduce heuristics that assign prefixes of a given classifier to groups with fixed maximal width l. The first intuitive approach is to remove bit identities that have the most similar values in all prefixes. Specifically, we denote by $\mathbf{0}_i$ ($\mathbf{1}_i$) the number of prefixes whose ith bit is either 0 (1) or $*$. In addition, we denote $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$. Algorithm 3 removes bit identities with maximal value of S_i, finding l bits with minimal S_i, and then runs MAXOI on the classifier cut to these l bits. If we need exact values, we simply choose them from the reduced classifier. We call this heuristic MINSIMILAR because we minimize the number of similar bits.

Table I shows detailed simulation results for Algorithm 3. For five values of maximal width l, it shows Algorithm 3 results with *exact* = False, with *exact* = True, and, in the "Exp." columns, the results of straighforwardly expanding each prefix in the non-exact results; this results in copying each prefix 2^{*+1} times, where $*$ is the number of $*$ bits in the prefix. Both heuristics expose the tradeoff between number of groups and total resulting size of the classifier (the total size will obviously decrease with decreasing maximal width since it is bounded by (# rules)×(max. width)). The alternative approach is to choose bit identities that distinguish the maximal number of pairs of prefixes. Denoting by D_i a number of prefix pairs that differ in the ith bit, we get the MAXPAIR heuristic shown in Algorithm 4. Table II, with layout similar to Table I, shows the results of the MAXPAIR heuristic. The results are mostly similar to MINSIMILAR, they are two sides of the same coin; it appears that MINSIMILAR is better suited for small widths l, and MAXPAIR works better for larger l.

Example 2: In the classifier from Example 1, the MINSIMILAR heuristic computes $S_1 = 3$, $S_2 = 3$, $S_3 = 5$, $S_4 = 4$. For $l = 2$, it chooses bits 0 and 1 and computes the maximal order-independent set on these bits as $\{F_1, F_3, F_4\}$, getting the same results as REGROUP in Example 1. The MAXPAIR heuristic computes $S_1 = 6$, $S_2 = 4$, $S_3 = 0$, $S_4 = 1$ and then works in the same way for $l = 2$.

Algorithm 5: ONEGROUP, the MINSIMILAR$_\delta$ heuristic; $*_{F^B}$ denotes the number of $*$ bits in F^B.

Data: $\mathcal{K}$, l, δ, *exact*

begin
- **for** every bit $i = 1, \ldots, k$ **do**
  - Compute $S_i = \max\{\mathbf{0}_i, \mathbf{1}_i\}$
- Choose $B = \{b_1, \ldots, b_l\}$ of l bit identities with minimal S_i
- $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leq \delta\}$
- **return** MAXOI($\mathcal{K}_C^B$)

Algorithm 6: ONEGROUP, the MAXPAIR$_\delta$ heuristic; $*_F$ denotes the number of $*$ bits in F.

Data: $\mathcal{K}$, max. width l, *exact*

begin
- **for** every bit $i = 1, \ldots, k$ **do**
  - compute $D_i \leftarrow \max\{|(F, F')| \mid F, F' \in \mathcal{K}, F_i \neq F'_i, F_i \neq *, F'_i \neq *\}$
- Choose $B = \{b_1, \ldots, b_l\}$ of l bit identities with maximal D_i
- $C \leftarrow \{F \in \mathcal{K} \mid *_{F^B} \leq \delta\}$
- **return** MAXOI($\mathcal{K}_C^B$)

C. Strong Order-Independence: Tradeoff

In Algorithms 3 and 4, we explored two ways to achieve strong order-independence (i.e., no $*$ bits in the filters): either removing all prefixes with $*$ bits prior to running MAXOI or running the algorithm as usual and then expanding all $*$ bits with an exponential blowup to the classifier. Tables I and II suggest that these are the two extremes with respect to our two conflicting objectives: cutting the width down vs. reducing the number of filters. In the following heuristic, we consider the middle ground between these two extremes. Algorithm 5 shows the MINSIMILAR$_\delta$ heuristic where we

	Original		Per prefix		max width = 13				max width = 16				max width = 24				max width = 32				max width = 48			
					with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$
	# rules	kb	#gr.	kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb
hous	1475	184.4	36	69.7	44	12.7	59.8	69 12.3	27	15.3	123.4	47 17.5	9	25.6	4096.0	48 20.5	10	28.7	85015.5	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
clev	1475	184.4	36	69.7	44	12.7	59.8	69 12.3	27	15.3	123.4	47 17.5	9	25.6	4096.0	48 20.5	10	28.7	85015.5	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
kans	1475	184.4	36	69.7	44	12.7	59.8	69 12.3	27	15.3	123.4	47 17.5	9	25.6	4096.0	48 20.5	10	28.7	85015.5	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
losa	1475	184.4	36	69.7	44	12.7	59.8	69 12.3	27	15.3	123.4	47 17.5	9	25.6	4096.0	48 20.5	10	28.7	85015.5	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
seat	1476	184.5	36	69.8	42	12.8	59.6	69 12.2	27	15.3	107.2	45 17.9	9	25.6	4088.6	65 19.9	10	28.7	85015.6	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
atla	1476	184.5	36	69.8	45	12.7	59.7	64 12.3	27	15.3	121.2	46 17.6	9	25.6	4096.1	65 19.9	10	28.7	85015.6	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0
chic	1476	184.5	36	69.8	42	12.7	59.7	64 12.3	27	15.0	106.5	45 18.0	9	24.8	4062.7	64 20.3	10	27.4	78798.1	19 26.0	7	48.8	$3.9 \cdot 10^9$	15 30.0
newy	1477	184.6	36	69.9	42	12.8	59.9	68 12.2	27	15.3	120.6	45 18.0	9	25.6	4096.1	57 20.5	10	28.7	85015.7	20 26.2	7	51.0	$4.1 \cdot 10^9$	15 30.1
wash	1475	184.4	36	69.8	42	12.8	59.8	62 12.1	33	15.0	113.1	45 17.6	9	25.6	4096.0	57 20.5	10	28.7	85009.5	18 26.2	6	50.9	$4.1 \cdot 10^9$	15 30.1
salt	1475	184.4	36	69.7	44	12.7	59.8	69 12.3	27	15.3	123.4	47 17.5	9	25.6	4096.0	48 20.5	10	28.7	85015.5	18 26.1	7	50.9	$4.1 \cdot 10^9$	15 30.0

TABLE I: Simulation results, the MINSIMILAR heuristic (Algorithm 3).

	Original		Per prefix		max width = 13				max width = 16				max width = 24				max width = 32				max width = 48			
					with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$	with $*$		Exp.	without $*$
	# rules	kb	#gr.	kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb	#	kb	kb	# kb
hous	1475	184.4	36	69.7	44	12.7	59.8	67 14.0	27	15.3	123.4	54 18.3	9	25.6	4096.0	52 18.0	10	28.7	85015.5	28 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7
clev	1475	184.4	36	69.7	44	12.7	59.8	67 14.0	27	15.3	123.4	54 18.3	9	25.6	4096.0	52 18.0	10	28.7	85015.5	28 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7
kans	1475	184.4	36	69.7	44	12.7	59.8	67 14.0	27	15.3	123.4	54 18.3	9	25.6	4096.0	52 18.0	10	28.7	85015.5	28 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7
losa	1475	184.4	36	69.7	44	12.7	59.8	67 14.0	27	15.3	123.4	54 18.3	9	25.6	4096.0	52 18.0	10	28.7	85015.5	28 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7
seat	1476	184.5	36	69.8	42	12.8	59.6	49 14.7	27	15.3	107.2	64 18.3	9	25.6	4088.6	50 18.0	10	28.7	85015.6	28 25.8	7	50.9	$4.1{\cdot}10^9$	17 29.7
atla	1476	184.5	36	69.8	45	12.7	59.7	65 14.1	27	15.3	121.2	51 18.4	9	25.6	4096.1	51 18.0	10	28.7	85015.6	27 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7
chic	1476	184.5	36	69.8	42	12.7	59.7	50 14.8	27	15.0	106.5	59 18.3	9	24.8	4062.7	56 17.7	10	27.4	78798.1	26 25.8	7	48.8	$3.9{\cdot}10^9$	17 29.7
newy	1477	184.6	36	69.9	42	12.8	59.9	48 14.7	27	15.3	120.6	52 18.4	9	25.6	4096.1	57 17.6	10	28.7	85015.7	28 25.8	7	51.0	$4.1{\cdot}10^9$	17 29.6
wash	1475	184.4	36	69.8	42	12.8	59.8	66 14.0	33	15.0	113.1	53 18.4	9	25.6	4096.0	56 17.5	10	28.7	85009.5	28 25.8	6	50.9	$4.1{\cdot}10^9$	17 29.7
salt	1475	184.4	36	69.7	44	12.7	59.8	67 14.0	27	15.3	123.4	54 18.3	9	25.6	4096.0	52 18.0	10	28.7	85015.5	28 25.8	7	50.9	$4.1{\cdot}10^9$	16 29.7

TABLE II: Simulation results, the MAXPAIR heuristic (Algorithm 4).

remove the filters with at most $\delta *$ bits before running MAXOI; all the rest is exactly like MINSIMILAR. Algorithm 6 does the same for MAXPAIR. Tables III and IV show the results for $\delta = \{1, 2, 3, 4\}$. Although generally, of course, the expanded size of the classifier grows with δ, there are cases, especially with small values of l, when increasing δ actually lets us save both size and number of groups; compare, for instance, the four δ values for $l = 13$ in Table IV.

Example 3: In the classifier from Example 1, the MINSIMILAR and MAXPAIR heuristics with *exact* $=$ true and $l = 3$ both choose bits 0, 1, and 3 and then have to discard all rules except F_1 and F_2, which form the first group, and then split the rest into two more groups, getting three groups with total size 4 bits. For $l = 3$ and $\delta = 1$, however, the first group for $l = 3$ will contain $\{F_1, F_2, F_3, F_4\}$, and the width will be reduced to 3, getting a total size of 12 bits in two groups.

D. Dynamic Updates

Another important aspect of the practical applications is the availability of dynamic updates: whether it is possible to add new filters on the fly.

In our settings it is trivial to add a new filter if it is still order-independent with one the existing groups on the same bits as the group is constructed. However, if we simply recompute the groups every time a new filter does not fit into an existing group, it may become infeasible. Clearly keeping a bigger subset of bit-identities to implements order-independence in the same group or increasing number of potential groups decrease chances to recompute belonging of filters to different groups. As a result we propose a simple heuristic: keep a few groups in reserve and then, when a new filter arrives, do the following (we do not formulate it as an algorithm since any of the previous algorithms may be used in adding the new filters): try to add it to one of the existing groups; if it does not fit, try to add it to the reserve groups, in order; if all reserve groups are full, recompute all groups and empty the reserve groups.

E. Evaluation in DPDK

For experimental evaluation, we adopt the Intel Data Plane Development Kit (DPDK) [5]. The DPDK is a set of data plane libraries and drivers for fast packet processing. The DPDK offers low-overhead access to raw network packets using direct access to NIC, pollmod drivers, etc. The DPDK processes packets outside the Linux Networking Stack avoiding costly packet encapsulation in the Linux kernel and its expensive system calls. The DPDK provides LPM6 library that implements LPM table search method for 128-bit keys. The LPM6 implementation uses modification of the *DIR-24-8* algorithm [9] for IPv4 that trades memory usage for improved LPM lookup speed. In the IPv6 case, instead of using 2 levels as in the IP case, the 14 levels are used; the first level is indexed using the first 24 bits of the IPv6 address, while the rest of the levels, are indexed using the rest IPv6 address, in chunks of 8 bits.

The goal of experiment is to study the impact of "pseudo parallelism" achieved by multi-group representations vs. multiple serial lookups used by LPM6. Both implementations use instruction-level parallelism from processor's pipeline, but in "serial" case the pipeline stalls while it is waiting to know memory address of next level in LPM data structure. In the pseudo-parallel case, all addresses are known before the actual execution and might be efficiently prefetched by the processor. In particular, our goal is to understand how many groups

	Original	$\delta = 1$										$\delta = 2$									
		$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$			$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$		
	#	kb	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.
hous	1475	184.4	37	30.9	70.5	34	37.1	86.6	36	39.7	89.9	18	46.0	92.5	14	56.1	114.0	35	31.3	71.8	28	63.9	136.4	28	51.0	107.1	17	39.6	84.1	14	42.6	114.8
clev	1475	184.4	37	30.9	70.5	34	37.1	86.6	36	39.7	89.9	18	46.0	92.5	14	56.1	114.0	35	31.3	71.8	28	63.9	136.4	28	51.0	107.1	17	39.6	84.1	14	42.6	114.8
kans	1475	184.4	37	30.9	70.5	34	37.1	86.6	36	39.7	89.9	18	46.0	92.5	14	56.1	114.0	35	31.3	71.8	28	63.9	136.4	28	51.0	107.1	17	39.6	84.1	14	42.6	114.8
losa	1475	184.4	37	30.9	70.5	34	37.1	86.6	36	39.7	89.9	18	46.0	92.5	14	56.1	114.0	35	31.3	71.8	28	63.9	136.4	28	51.0	107.1	17	39.6	84.1	14	42.6	114.8
seat	1476	184.5	39	32.8	73.5	37	32.1	77.2	34	37.4	84.8	19	46.0	92.4	14	56.1	114.0	37	29.8	70.0	29	53.0	111.8	29	45.7	104.7	18	39.9	84.6	14	39.7	84.9
atla	1476	184.5	37	34.7	76.5	35	49.6	100.7	35	40.2	90.6	18	46.1	92.5	14	56.1	114.0	34	28.0	67.8	29	68.7	141.5	28	45.2	102.1	17	39.6	84.1	15	42.6	114.7
chic	1476	184.5	38	32.7	72.7	35	33.3	79.9	37	37.7	86.2	19	43.0	88.0	13	57.6	116.8	38	30.7	71.3	30	53.0	111.8	29	51.3	107.2	18	30.7	66.3	15	55.4	110.9
newy	1477	184.6	40	32.8	72.8	35	49.9	100.6	41	53.7	108.9	19	32.4	66.6	14	61.7	123.3	35	30.9	70.6	30	53.9	115.3	27	51.8	110.1	17	30.4	67.7	14	39.5	113.2
wash	1475	184.4	36	33.3	73.7	34	32.4	78.3	37	56.7	114.6	20	34.3	71.3	15	44.4	88.9	33	26.9	65.9	30	67.2	140.6	28	48.0	105.8	19	27.5	105.0	14	55.6	111.3
salt	1475	184.4	37	30.9	70.5	34	37.1	86.6	36	39.7	89.9	18	46.0	92.5	14	56.1	114.0	35	31.3	71.8	28	63.9	136.4	28	51.0	107.1	17	39.6	84.1	14	42.6	114.8
		$\delta = 3$										$\delta = 4$									
		$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$			$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$		
	#	kb	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.
hous	1475	184.4	33	20.9	87.8	26	64.9	388.3	24	46.5	115.8	18	27.4	181.0	12	42.7	320.8	33	21.1	92.9	24	23.9	251.2	21	48.8	159.1	16	27.5	351.9	11	45.9	192.6
clev	1475	184.4	33	20.9	87.8	26	64.9	388.3	24	46.5	115.8	18	27.4	181.0	12	42.7	320.8	33	21.1	92.9	24	23.9	251.2	21	48.8	159.1	16	27.5	351.9	11	45.9	192.6
kans	1475	184.4	33	20.9	87.8	26	64.9	388.3	24	46.5	115.8	18	27.4	181.0	12	42.7	320.8	33	21.1	92.9	24	23.9	251.2	21	48.8	159.1	16	27.5	351.9	11	45.9	192.6
losa	1475	184.4	33	20.9	87.8	26	64.9	388.3	24	46.5	115.8	18	27.4	181.0	12	42.7	320.8	33	21.1	92.9	24	23.9	251.2	21	48.8	159.1	16	27.5	351.9	11	45.9	192.6
seat	1476	184.5	35	21.8	78.9	27	63.6	383.4	22	45.4	121.1	17	38.4	90.2	13	45.0	94.0	33	20.2	93.4	26	32.0	203.6	21	45.6	171.7	17	27.9	352.6	11	46.5	335.4
atla	1476	184.5	33	20.5	84.0	28	62.1	375.0	23	50.3	114.3	18	27.4	181.0	12	42.7	320.8	33	19.9	88.9	23	22.8	251.7	22	46.5	160.7	16	27.3	351.5	11	45.9	192.7
chic	1476	184.5	33	19.1	73.3	27	65.0	385.1	24	45.7	124.4	17	41.1	96.3	12	43.0	321.2	33	19.0	93.7	28	30.2	193.1	21	45.9	155.1	14	28.6	354.3	11	45.9	192.7
newy	1477	184.6	31	27.3	72.8	26	64.1	395.0	23	46.4	121.5	17	41.8	99.0	12	46.3	315.9	30	25.1	86.7	27	32.4	200.2	23	49.2	159.9	16	32.0	104.2	11	46.3	335.9
wash	1475	184.4	35	20.3	79.1	26	67.4	414.6	22	49.2	127.3	17	41.1	96.2	12	45.0	182.4	33	20.7	87.2	23	25.4	252.1	22	49.3	162.5	16	27.5	351.9	12	45.3	191.1
salt	1475	184.4	33	20.9	87.8	26	64.9	388.3	24	46.5	115.8	18	27.4	181.0	12	42.7	320.8	33	21.1	92.9	24	23.9	251.2	21	48.8	159.1	16	27.5	351.9	11	45.9	192.6

TABLE III: Simulation results, the MINSIMILAR$_\delta$ heuristic (Algorithm 5).

	Original	$\delta = 1$										$\delta = 2$									
		$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$			$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$		
	#	kb	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.
hous	1475	184.4	39	75.7	153.3	34	81.9	165.1	28	52.2	105.3	26	28.3	71.7	14	51.8	118.8	37	77.1	159.4	25	72.6	152.4	32	50.4	111.4	22	33.6	73.4	13	36.8	207.7
clev	1475	184.4	39	75.7	153.3	34	81.9	165.1	28	52.2	105.3	26	28.3	71.7	14	51.8	118.8	37	77.1	159.4	25	72.6	152.4	32	50.4	111.4	22	33.6	73.4	13	36.8	207.7
kans	1475	184.4	39	75.7	153.3	34	81.9	165.1	28	52.2	105.3	26	28.3	71.7	14	51.8	118.8	37	77.1	159.4	25	72.6	152.4	32	50.4	111.4	22	33.6	73.4	13	36.8	207.7
seat	1476	184.5	38	78.1	157.9	33	81.5	164.3	28	53.1	107.2	24	29.1	73.2	15	52.7	121.2	36	77.3	160.4	30	72.2	151.8	31	54.0	117.3	20	33.8	74.4	15	45.3	182.6
atla	1476	184.5	45	76.4	154.8	33	79.6	160.4	29	53.1	107.3	37	44.5	90.6	14	51.9	118.8	39	77.4	160.2	28	72.6	152.4	33	54.3	117.4	20	34.2	74.8	14	36.8	207.7
chic	1476	184.5	40	74.9	151.8	31	81.4	163.9	27	52.1	105.3	22	28.9	73.0	14	52.9	121.5	35	77.6	160.0	28	71.8	151.0	30	54.0	117.3	19	33.5	73.8	14	53.2	155.7
newy	1477	184.6	40	77.2	156.2	33	82.3	165.8	29	50.2	100.9	24	28.1	70.9	15	52.7	121.2	37	76.2	157.9	26	74.5	153.1	29	49.9	111.4	20	33.5	73.3	14	36.9	207.9
wash	1475	184.4	41	77.5	156.6	33	83.4	168.0	28	52.2	105.5	22	26.7	68.6	17	38.9	79.8	36	76.8	159.3	32	74.9	157.4	32	54.9	118.7	22	33.5	73.3	12	30.6	200.1
salt	1475	184.4	39	75.7	153.3	34	81.9	165.1	28	52.2	105.3	26	28.3	71.7	14	51.8	118.8	37	77.1	159.4	25	72.6	152.4	32	50.4	111.4	22	33.6	73.4	13	36.8	207.7
		$\delta = 3$										$\delta = 4$									
		$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$			$l = 13$			$l = 16$			$l = 24$			$l = 32$			$l = 48$		
	#	kb	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.	#gr.	kb	exp.
hous	1475	184.4	30	65.2	142.2	23	72.5	159.5	22	27.0	173.1	21	29.2	112.3	12	45.3	340.3	32	61.4	148.0	19	94.4	261.8	19	36.9	358.7	19	27.3	329.4	11	45.2	536.5
clev	1475	184.4	30	65.2	142.2	23	72.5	159.5	22	27.0	173.1	21	29.2	112.3	12	45.3	340.3	32	61.4	148.0	19	94.4	261.8	19	36.9	358.7	19	27.3	329.4	11	45.2	536.5
kans	1475	184.4	30	65.2	142.2	23	72.5	159.5	22	27.0	173.1	21	29.2	112.3	12	45.3	340.3	32	61.4	148.0	19	94.4	261.8	19	36.9	358.7	19	27.3	329.4	11	45.2	536.5
seat	1476	184.5	31	65.1	143.6	22	71.8	156.3	23	29.2	176.1	24	29.5	118.4	12	44.8	339.3	35	62.6	152.1	22	92.8	266.2	17	36.1	361.6	18	27.7	194.1	12	45.1	548.3
atla	1476	184.5	31	65.6	143.4	22	70.6	156.0	23	28.8	175.3	25	30.9	118.9	12	45.4	340.5	34	62.6	150.9	22	92.5	265.6	17	37.3	362.7	19	28.7	329.3	11	45.5	537.1
chic	1476	184.5	32	66.9	144.3	22	71.3	156.2	23	29.7	177.0	21	27.7	177.4	12	38.1	407.6	33	63.1	151.3	22	92.9	266.3	16	36.0	361.6	21	28.6	328.7	12	39.7	565.3
newy	1477	184.6	32	64.7	142.5	21	70.2	157.1	23	25.4	170.2	23	29.0	115.9	12	38.6	281.1	32	63.9	152.9	20	94.1	260.5	17	36.1	361.5	19	30.3	211.6	12	46.5	539.2
wash	1475	184.4	34	75.7	165.1	25	74.4	164.6	23	25.9	171.5	26	30.5	118.2	12	33.0	282.0	31	62.6	150.8	19	94.0	260.7	19	26.7	309.5	20	30.0	210.9	12	34.6	434.4
salt	1475	184.4	30	65.2	142.2	23	72.5	159.5	22	27.0	173.1	21	29.2	112.3	12	45.3	340.3	32	61.4	148.0	19	94.4	261.8	19	36.9	358.7	19	27.3	329.4	11	45.2	536.5

TABLE IV: Simulation results, the MAXPAIR$_\delta$ heuristic (Algorithm 6).

based on 24 or 32 bits can be used during processing of 14 levels of LPM6 implementation. For this experiments we use *rte_lpm6_lookup()* (or *rte_lpm_lookup()*) functions with two calls of *rte_get_tsc_cycles()* function. Difference between TSC values is the number of cycles to execute the set of lookups.

We run every test in each experiments 1000 times and then average the cycles. As a testbed, we use a physical server with two 8 core Intel Xeon CPU E5-2670 @ 2.60GH, 96 GB DDR3 RAM, Ubuntu 14.04.2 LTS. Results of our experiments show that it is possible to run multi-group representation on 14-18 groups in the same time as IPv6 lookup, which together with other results of this section leads to significant size reduction with no loss for performance.

Moreover, in future work and practical implementations there are ways to make pseudo-parallel lookups even faster and smoother in the number of cycles. First, every x86 server has at least 4 independent memory channels; we can physically allocate the groups' LPM tables in different memory banks to make multiple accesses faster (using platform-dependent code, *bootmem*, *page_alloc* etc). Second, one can use builtin prefetch instructions so that CPUs will load these tables in cache in advance (e.g., DPDK has the *rte_prefetch()* API). Third, comparison in LPM tables can be rewritten with *movcc* assembly instructions on x86.

V. RELATED WORK

Some of the fastest IP lookup engines use space- and lookup-efficient encodings of prefix trees [2], [6], [7], [21]. Unfortunately, the lookup time on prefix trees increases proportionally with the address size, which is a problem for IPv6 due to the increased address space. Therefore, most prefix tree implementations for IPv6 carefully adapt the basic IP prefix tree data structure to the special needs of IPv6. Unfortunately,

a naive application of IPv4 level-compression algorithms [1], [21], [24] to IPv6 do not work straight away, as IPv6 prefix trees are considerably sparse and therefore extending nodes' stride naively yields a lot of empty nodes, leading to an explosion in storage size. Consequently, one needs to find a way to encode sparse subtrees inside prefix tree nodes. Tree bitmaps, which use compact bitmaps to mark existing and missing children, have proved very useful in this context [7]. Extensions of tree bitmaps to IPv6 include shape shifting tries [20], which employ a succinct bitmap encoding to implement children membership queries, and FlashTrie [2], using a hash-based data structure for this purpose. While very efficient these schemes are still just prefix trees under the hood, inherently sequential and hence not amenable to parallelization.

Prefix-length-based decompositions are much more appealing for parallel execution [17], [23], since prefixes grouped together in a single execution thread are independent and thus allow fast and efficient exact matching data structures to be used in implementations. The idea was proposed in [23], where the prefix groups were organized into a (sequential) binary search scheme to find the most specific match. Note that this scheme requires special dummy entries (called markers) to be stored at each level to signal the presence of more specifics. Our scheme performs the lookups in parallel and therefore does not need markers. The original scheme [23] was extended to IPv6 in [25], using a TCAM to store each group, and [17], which uses a separate Bloom-filter to eliminate the levels that do not have matching entries in performing the binary search. Our scheme is complementary to these proposals; by generalizing prefix-length-based decompositions to reduced order-independent decompositions we could realize all the appealing properties (independence and exact match semantics in groups) with much fewer groups and significantly smaller per-group lookup tables.

[12], [16] addressed efficient time-space tradeoff for multi-field classification, where fields are represented by ranges. Note that limiting consideration to LPM filters significantly simplify representations and allow to consider new special cases as exact values on l bits. [14], [15] exploit Boolean minimization techniques to reduce required memory to represent packet classifiers.

VI. Conclusion

Identifying appropriate invariants around which to implement lookup can significantly affect lookup time and space requirements for the corresponding data structures. In this paper we exploit weak and strong forms of order independence in order to find decompositions for an IPv6 FIB that allow to exploit a system level parallelism. The methods that we suggest define an abstraction layer which remains transparent to methods used for representation of lookup tables in network elements. Our experimental evaluations suggest that the proposed approach can lead to drastic savings in the memory footprint of lookup tables with no loss to performance.

Acknowledgments

The work of Sergey Nikolenko was supported by the President Grant for Young Ph.D. Researchers MK-7287.2016.1 and by the Government of the Russian Federation grant 14.Z50.31.0030. Erika R. Bérczi-Kovács is supported by grant no. K 109240 from the National Development Agency of Hungary, based on a source from the Research and Technology Innovation Fund.

References

[1] H. Asai and Y. Ohara. Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup. In *ACM SIGCOMM*, pages 57–70, 2015.

[2] M. Bando and J. H. Chao. Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps. In *IEEE INFOCOM*, pages 821–829, 2010.

[3] P. Berman, B. DasGupta, and M. Kao. Tight approximability results for test set problems in bioinformatics. *JCSS*, 71(2):145–162, 2005.

[4] Cisco CRS forwarding Processor Cards. http://www.cisco.com/c/en/us/products/collateral/routers/carrierrouting-system/datasheetc78730790.pdf.

[5] Data plane development kit (dpdk). http://dpdk.org, 2015.

[6] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[7] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *Computer Communication Review*, 34(2):97–122, 2004.

[8] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to np-completeness, 1979.

[9] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *IEEE INFOCOM*, pages 1240–1247, 1998.

[10] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[11] Internet2 - forwarding information base. http://vn.grnoc.iu.edu/Internet2/fib/index.cgi.

[12] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal. Space and speed tradeoffs in TCAM hierarchical packet classification. *J. Comput. Syst. Sci.*, 79(1):111–121, 2013.

[13] J. M. Kleinberg and É. Tardos. *Algorithm design*. Addison-Wesley, 2006.

[14] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan. Towards efficient implementation of packet classifiers in sdn/openflow. In *ACM HotSDN*, pages 153–154, 2013.

[15] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan. Strategies for mitigating TCAM space bottlenecks. In *IEEE HOTI*, pages 25–32, 2014.

[16] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. SAX-PAC (scalable and expressive packet classification). In *ACM SIGCOMM*, 2014.

[17] K. Lim, K. Park, and H. Lim. Binary search on levels using a bloom filter for ipv6 address lookup. In *ACM/IEEE ANCS*, pages 185–186, 2009.

[18] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE JSAC*, 17(6):1083 –1092, 1999.

[19] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM*, pages 111–122, 2013.

[20] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster IP route lookup. In *IEEE ICNP*, pages 358–367, 2005.

[21] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perform. Eval. Rev.*, 26(1):1–10, 1998.

[22] The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor. http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000seriesaggregationservicesrouters/solution_overview_c22-448936.html.

[23] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *ACM SIGCOMM*, pages 25–36, 1997.

[24] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. In *ACM SIGCOMM*, pages 39–50, 2014.

[25] G. Zhu, S. Yu, and J. Dai. Binary search on prefix covered levels for ip address lookup. In *WiCOM*, pages 3859–3862, 2009.