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 \( K \) is an ordered set of rules \( K = (R_1, \ldots, R_N) \), 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 \( < \); if a header matches both \( R_i \) and \( R_j \) for \( R_i < R_j \), 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: \( R_i \preceq R_j \) 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 \( K = (R_1, \ldots, R_N) \), let \( K^B \) be the (sub)classifier obtained from \( K \) 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 underused 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 \( K = (R_1, \ldots, R_N) \) and a subset of the index set \( C \subseteq \{1, \ldots, N\} \). Now, the classifier \( K_C = (R_i : i \in C) \) is called the (classifier) group of \( K \) defined by \( C \). Given any partition \( C = \{C_1, \ldots, C_k\} \) 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 \( C \) is given by exactly \( k \) classifier groups on \( C \): \( K_C = \{K_{C_1}, \ldots, K_{C_k}\} \). 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 the longest prefix length.

### III. Multi-group Representations

Intuitively, prefix-length-based decompositions enjoy an appealing structural property: for every input each subclassifier 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 \( K \), it holds that \( K \) contains at most one matching rule for any header if and only if \( K \) 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 \( K \) (not necessary order-independent) and a constant \( l \), assign \( K \)'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 \( \beta \). 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.
because we minimize the number of similar bits. From the reduced classifier, we call this heuristic $M$.

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 $O_i (1)$ the number of prefixes whose $i$th bit is either 0 (1) or *. In addition, we denote $S_i = \max\{O_i (1)\}$. 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 $\text{MinSimilar}$ 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 straightforwardly expanding each prefix in the non-exact results; this results in copying each prefix $2^{\#\text{rules}}$ times, where $\#\text{bits}$ 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 $\#\text{rules}\times(\text{max. width})$). 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 $i$th bit, we get the $\text{MaxPair}$ heuristic shown in Algorithm 4.

Example 2: In the classifier from Example 1, the $\text{MinSimilar}$ 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 $\text{Regroup}$ in Example 1. The $\text{MaxPair}$ heuristic computes $S_1 = 6$, $S_2 = 4$, $S_3 = 0$, $S_4 = 1$ and then works in the same way for $l = 2$.

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 $\text{MinSimilar}_d$ heuristic where we...
The table shows the simulation results for the MINSIMILAR heuristic (Algorithm 3).

<table>
<thead>
<tr>
<th>Original</th>
<th>Per prefix</th>
<th>max width = 13</th>
<th>max width = 16</th>
<th>max width = 24</th>
<th>max width = 32</th>
<th>max width = 48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># rules</td>
<td>#gr. kb</td>
<td># # gr. kb</td>
<td># # gr. kb</td>
<td># # gr. kb</td>
<td># # gr. kb</td>
</tr>
<tr>
<td>house</td>
<td>1475 184.4</td>
<td>36 69.7</td>
<td>44 12.7 59.8</td>
<td>69 12.3</td>
<td>27 15.3 123.4 47 17.5</td>
<td>9 25.6 4096.0 48 20.5</td>
</tr>
<tr>
<td>city</td>
<td>1475 184.4</td>
<td>36 69.7</td>
<td>44 12.7 59.8</td>
<td>69 12.3</td>
<td>27 15.3 123.4 47 17.5</td>
<td>9 25.6 4096.0 48 20.5</td>
</tr>
<tr>
<td>kan</td>
<td>1475 184.4</td>
<td>36 69.7</td>
<td>44 12.7 59.8</td>
<td>69 12.3</td>
<td>27 15.3 123.4 47 17.5</td>
<td>9 25.6 4096.0 48 20.5</td>
</tr>
<tr>
<td>losa</td>
<td>1475 184.4</td>
<td>36 69.7</td>
<td>44 12.7 59.8</td>
<td>69 12.3</td>
<td>27 15.3 123.4 47 17.5</td>
<td>9 25.6 4096.0 48 20.5</td>
</tr>
<tr>
<td>seat</td>
<td>1476 184.5</td>
<td>36 69.8</td>
<td>42 12.8 59.6</td>
<td>69 12.2</td>
<td>27 15.3 107.2 45 17.9</td>
<td>9 25.6 4088.6 65 19.9</td>
</tr>
<tr>
<td>atla</td>
<td>1476 184.5</td>
<td>36 69.8</td>
<td>42 12.8 59.7</td>
<td>69 12.3</td>
<td>27 15.3 121.2 46 17.6</td>
<td>9 25.6 4061.6 65 19.9</td>
</tr>
<tr>
<td>chic</td>
<td>1476 184.5</td>
<td>36 69.8</td>
<td>42 12.7 59.7</td>
<td>69 12.3</td>
<td>27 15.0 106.5 45 18.0</td>
<td>9 24.8 4062.7 64 20.3</td>
</tr>
<tr>
<td>newy</td>
<td>1476 184.5</td>
<td>36 69.9</td>
<td>42 12.8 59.9</td>
<td>68 12.2</td>
<td>27 15.3 120.6 45 18.0</td>
<td>9 25.6 4061.5 57 20.5</td>
</tr>
<tr>
<td>wash</td>
<td>1475 184.5</td>
<td>36 69.8</td>
<td>42 12.8 59.8</td>
<td>62 12.1</td>
<td>33 15.0 113.1 45 17.6</td>
<td>9 25.6 4060.5 57 20.5</td>
</tr>
<tr>
<td>salt</td>
<td>1475 184.4</td>
<td>36 69.7</td>
<td>44 12.7 59.8</td>
<td>69 12.3</td>
<td>27 15.3 123.4 47 17.5</td>
<td>9 25.6 4060.6 48 20.5</td>
</tr>
</tbody>
</table>
based on 24 or 32 bits can be used during processing of 14 levels of LPM6 implementation. For this experiments we use `rtelpm6_lookup()` (or `rtelpm_lookup`) functions with two calls of `rtelpm_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 @ 2.60GHz, 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 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, `mmap`, `page_alloc` etc). Second, one can use built-in prefetch instructions so that CPUs will load these tables in cache in advance (e.g., DPDK has the `rt_prefetch()` API). Third, comparison in LPM tables can be rewritten with `mvec` 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 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