BDD-Based Algorithms for Packet Classification

Nina Narodytska
VMware Research

Leonid Ryzhyk
VMware Research

Igor Ganichev
Google

Soner Sevinc
Facebook

***Abstract*—Packet classifiers are the building blocks of modern networking. A classifier determines the action to take on a packet by matching its header against a set of rules. Efficient classification is achieved by using associative memory to perform the match operation in one clock cycle. This requires compressing large rule sets to fit in the small associative memory space available in modern network switches. We propose two symbolic rule set compression algorithms based on binary decision diagrams. Following McGeer and Yalagandula, we formalize the problem as that of obtaining a sequential cover of the rule set. We develop a simple BDD-based algorithm for computing sequential covers, which significantly outperforms state of the art algorithms in terms of compression ratio—a surprising result that highlights the unexplored potential of symbolic techniques in packet classification. Despite this improvement, very large industrial classifiers are still beyond reach. We decompose such classifiers into a pipeline of smaller classifiers over subsets of packet header fields. We then compress each classifier using the sequential cover technique. Our algorithm is able to compress industrial rule sets with hundreds of thousands rules to readily fit in the memory of network switches.**

I. Introduction

Modern datacenter, corporate, and telecom networks implement a rich set of functions, including firewalling, routing, load balancing, and service chaining, to name a few. Inside network switches, these functions are implemented as *packet classifiers*, which determine the next action to perform on the packet (e.g., drop or forward) by matching its headers against a set of rules, e.g., "*Drop UDP packets sent from hosts in the* $172.16.*.*$ *IP subnet with destination port numbers in the 554–680 range.*" Moreover, multi-terabit-per-second packet processing rates, supported in modern network switches, require packet classification to complete in a short and *constant* time. To this end, switches are equipped with *ternary content-addressable memories* (TCAMs), also known as *associative memories*, capable of matching a packet against multiple rules in a single clock cycle.

Due to their high cost and power consumption, TCAMs remain a limited resource. Current high-end switches are equipped with thousands to tens of thousands of TCAM entries. In contrast, real-world packet classifiers may consist of *hundreds of thousands rules*. Whenever the rule set does not fit in the TCAM of a single switch, the network administrator is left with two options: (1) partition the rule set across multiple switches, dramatically raising the cost and complexity of network management, or (2) sacrifice performance by moving a subset of rules to conventional memory.

Rule set compression mitigates the problem by computing a smaller rule set equivalent to the original classifier. It can be used to fit a large classifier in the TCAM of a single switch or to enable integrated functionality by configuring a single switch with multiple classifiers.

Rule set compression algorithms represent rules as hypercubes in the multidimensional space of packet headers, seeking to cover the entire classifier with as few cubes as possible. Since computing an optimal solution to this problem is NP-hard [1], greedy heuristics are employed to reduce the search space. Ultimately, finding a small cover within a reasonable time frame depends on the ability of the algorithm to efficiently compute unions and intersections of cubes.

Existing algorithms represent cubes explicitly. We argue that significant speedup can be achieved by symbolically representing and manipulating *sets of cubes* as Binary Decision Diagrams (BDDs) [2]. In fact, simply by compiling the rule set to a BDD we obtain a compact representation of all its cubes. Furthermore, BDDs allow efficiently extracting *prime implicants*, i.e., cubes that are not contained within any other cube of the rule set. Based on this observation, we develop two BDD-based algorithms for rule set compression. Following McGeer and Yalagandula [3], we formulate the compression problem as that of computing *sequential cover* of the rule set. We propose a BDD-based algorithm for computing sequential cover. Our algorithm outperforms state of the art rule set compression algorithms, e.g., where the best known algorithm achieves 50% compression ratio, we achieve 10x compression.

Next we apply the same approach to three very large real-world classifiers with hundreds of thousands rules. To the best of our knowledge, these are the largest rule sets reported in the literature. We observe that on these rule sets, our algorithm still does not achieve satisfactory compression ratio. We therefore propose a new compression algorithm that *decomposes* the rule set into a pipeline of packet classifiers, where each classifier matches on a subset of packet header fields and computes a *tag* used as input to the next classifier. Our decomposition is based on the BDD-based functional decomposition technique by Lai *et al.* [4]. We compress each classifier by computing its sequential cover, as above.

We apply this algorithm to our real-world classifiers and show that it achieves two to four orders of magnitude compression on these classifiers, reducing them to readily fit into the TCAM table space of modern switches.

Our contribution is a pair of algorithms that significantly improve the state of the art in packet classification. To the best of our knowledge, this work is the first to apply two well-known logic minimization techniques—symbolic prime cover computation and functional decomposition—in this space.

II. Background

Boolean logic Let $(z_1, \ldots, z_l)$ be a set of Boolean variables. A *literal* is a Boolean variable z_i or its negation $\overline{z_i}$. A *cube* is a conjunction of literals, e.g., $z_1\overline{z_2}z_3$. We denote $c_{|(z_{i_1},\ldots,z_{i_j})}$ the restriction of cube c on variables $(z_{i_1}, \ldots, z_{i_j})$, obtained by eliminating all literals over other variables. A cube c is an *implicant* of a boolean function $g : B^l \rightarrow B$ iff $c \Rightarrow g(z_1, \ldots, z_l)$; c is a *prime implicant* iff it cannot be reduced, i.e., the removal of any literal from c results in a non-implicant.

Packet classification A packet *header* is a fixed-size bit vector $X = (x_1, \ldots, x_n)$, where a Boolean variable x_i corresponds to the ith bit of the header. Individual header bits are grouped into *fields*, which represent source and destination IP addresses of the packet, port numbers, etc.: $X = (X_1, \ldots, X_l)$, where bitvector X_j is the jth field of the header. A packet *classifier* $f : B^n \rightarrow A$ maps a header into one of a finite set of *actions* A, where $B = \{0, 1\}$. To encode classifiers to BDDs, we represent f as a function $F : B^n \times A \rightarrow B$ such that $(F(X, Y) = \top) \equiv (f(X) = Y)$. We assume that A is encoded using Boolean variables, for example $A = \{ACCEPT, DENY\}$ is encoded using a single Boolean.

Classifiers are represented as ordered lists of *rules* of the form: $R = ((m_1, a_1), \ldots, (m_k, a_k))$, where $m_j : B^n \rightarrow B$ is a *match* function that identifies a subset of the header space and $a_j \in A$ is the action to perform on the packet when its header belongs to this subset (note that a_j are not required to be distinct). The corresponding classifier F_R picks the first rule that matches the header:

$$\begin{aligned} F_R(X, Y) = {} & m_1(X) \wedge (Y = a_1) \\ \vee \; & \overline{m_1(X)} \wedge m_2(X) \wedge (Y = a_2) \quad \ldots \\ \vee \; & \overline{m_1(X)} \wedge \ldots \wedge \overline{m_{k-1}(X)} \wedge m_k(X) \wedge (Y = a_k) \end{aligned} \tag{1}$$

To make sure that F_R defines a unique action for any header, the final rule (m_k, a_k) is a catch-all rule that matches all headers: $m_k(X) \equiv \top$.

Rule languages Network switches equipped with TCAMs are programmed using rules expressed in *ternary* form. Consider a header field $X_j = (z_1, \ldots, z_p)$. A *ternary constraint* fixes a subset of bits in X_j, leaving other bits unconstrained, i.e., it is a cube over X_j. *Prefix constraints*, frequently used in practice, are a special case of ternary constraints, that fix a prefix of the bit vector. A *ternary rule* is a rule whose match function is a conjunction of ternary constraints, one for each field X_j, $j \in [1, l]$. Equivalently, a ternary rule is a cube over X.

Example 1: Figure 1(a) shows an example ternary rule set where x_i are header variables and y is the action variable.■

In writing network policies, network administrators often use *range constraints*. A *range constraint* $c_1 \leq X_j \leq c_2$ interprets field X_j as an integer value stored using logarithmic encoding and requires this value to be in the range $[c_1, c_2]$. Rules with range constraints can be decomposed into multiple ternary rules by rewriting each range constraint as a disjunction of ternary constraints and taking a Cartesian product across all range constraints in the rule.

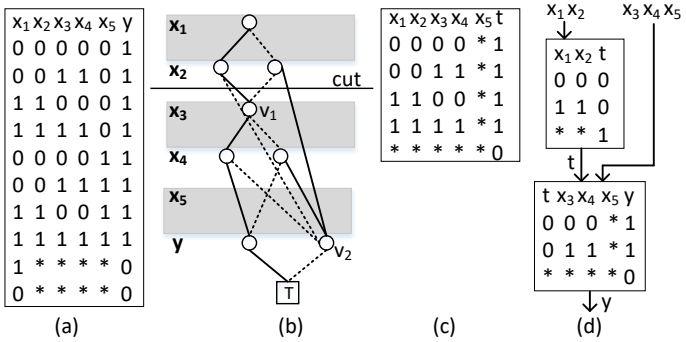

Fig. 1: Running example: (a) original classifier, (b) its BDD encoding, (c) compressed rule set, (d) decomposition into a sequence of two classifiers.

Algorithm 1 BDD-based sequential cover computation

```
1: function SeqCover(F)
2:     F+ ← F, F− ← F, cover ← ()
3:     while F− ≠ ⊥ do
4:         c ← LargestCube(F−)
5:         p ← PrimeImplicant(c, F+)
6:         cover ← cover + p
7:         F+ ← F+ ∨ p|X
8:         F− ← F− ∧ ¬p|X
9:     return cover
```

Modern network management software supports even richer policy languages that allow various Boolean combinations of range and ternary constraints. Such general rules can be decomposed into ternary rules by flattening their Boolean structure to Disjunctive Normal Form. Such expansion can be too expensive in practice and therefore should be avoided.

BDDs A BDD [2] is a compressed canonical representation of the binary decision tree of a Boolean function in the form of a directed acyclic graph. Figure 1(b) shows a BDD of the classifier specified in Figure 1(a).

Problem statement The rule set compression problem can be defined as follows: given a rule set R, written using any rule language, compute a minimal *ternary* rule set R' such that $F_R \equiv F_{R'}$. Computing an optimal solution is NP-hard [1]; therefore in practice we aim to achieve the best possible compression within a reasonable time frame.

III. Compression via sequential cover

McGeer and Yalagandula [3] introduced the notion of *sequential cover* for rule set compression. A sequential cover of a classifier $F(X, Y)$ is a set of cubes p_j, $j \in [1, u]$ over variables X and Y such that (1) p_js cover the entire classifier: $F(X, Y) \Rightarrow \bigvee_{j \in [1,u]} p_j$, and (2) each p_j is a prime implicant of function $F(X, Y) \vee \bigvee_{i<j} p_{i|X}$.

This definition is similar to the classical notion of *prime cover*, studied in logic synthesis [5]. There is, however, an important distinction: the second condition above states that each rule must only agree with F on headers not covered by previous rules; the rest of the header space is the don't care set where the rule can disagree with F. In contrast, all implicants in a prime cover must agree with F.

Name	#R	#ExpR	#Nodes	MDTC			Flint		
				Rules	Ratio	Time(s)	Rules	Ratio	Time(s)
cernet1	21	21	42	6	0.29	0	7	0.33	0.07
cernet2	39	39	165	37	0.95	0	35	0.90	0.15
cernet3	44	44	31	19	0.43	0	16	0.36	0.06
cernet4	52	52	112	52	1.00	0	18	0.35	0.11
cernet5	175	175	720	167	0.95	0.03	159	0.91	0.42
cernet6	664	664	686	515	0.78	0.05	439	0.66	1.03
cernet7	943	945	5161	881	0.93	0.14	843	0.89	4
cernet8	1491	1491	1664	1363	0.91	0.35	914	0.61	1

TABLE I: CERNET rule sets.

Name	#R	#ExpR	#Nodes	Flint		
				Rules	Ratio	Time(s)
acl-small	936	1305	1754	100	0.08	5
acl-large	9125	12.32K	15.97K	1385	0.11	267
fw-small	847	3382	6474	1311	0.39	44
fw-large	8737	35.71K	14.64K	6108	0.17	2076
ipc-small	916	1343	8360	566	0.42	33
ipc-large	8623	11.86K	12.91K	981	0.08	606

TABLE II: Synthetic benchmarks (reports means across all rule sets in each benchmark).

Each p_j in the sequential cover of F can be interpreted to a rule $r'_j = (m'_j, a'_j)$, where $m'_j = p_{j|_X}$ and a'_j is obtained by extracting the action from $p_{j|_Y}$. The resulting rule set $R' = ((m'_1, a'_1), \ldots, (m'_u, a'_u))$ is equivalent to R. A minimal rule set is an optimal solution to the rule set compression problem.

Example 2: Figure 1(c) shows a sequential cover of the classifier in Figure 1(a). Note that the last rule $(* * * * * 0)$ is *not* an implicant of the classifier, since not all assignments of $x_1 \ldots x_5$ map to action 0. However, it *is* an implicant of the classifier with the header space covered by previous rules added to its don't care set, since all assignments of $x_1 \ldots x_5$ not covered by previous rules map to 0.■

Algorithm 1 shows our BDD-based algorithm for computing a sequential cover. It takes a packet classifier F and returns a subset F^- of F not covered by the implicants computed so far, and a superset F^+ that includes the *don't care* set covered by the previous implicants. Sets F, F^+ and F^- are represented as BDDs and all operations on them are performed in the BDD form. The main loop of the algorithm computes a single new implicant on every iteration by first extracting a largest cube (i.e., a cube with the smallest number of literals) from F^-, which corresponds to a shortest path from the root to a leaf of the BDD. It then extends this cube to a prime implicant of F^+ by greedily removing literals from the cube so that the resulting cube is still in F^+. Note that the algorithm is not guaranteed to produce a minimal sequential cover.

Performance We implemented Algorithm 1 in a tool called Flint using the CUDD BDD library [6]. We encode packet classifier F_R into a BDD using formula 1. We use the group sifting [7] dynamic variable reordering heuristic, grouping variables that belong to the same field, to reduce the size of the resulting BDD. Due to the sensitive nature of real-world rule sets, very few of them are publicly available. We therefore use a mix of synthetic and real-world benchmarks in our evaluation. First, we use two datasets from [8]. The first of these datasets consists of 8 classifiers extracted from the CERNET network. The second dataset consists of 12 synthetic access control list (ACL) rule sets generated using the ClassBench tool [9]. We generate 5 additional synthetic datasets using ClassBench. These datasets fall into three categories: ACL, firewall (FW), and IP Chain (IPC). Each dataset consists of 20 rule sets. The header space in all rule sets consists of 5 fields and 104 bits. Table I compares Flint against MDTC [8], the most efficient rule set compression algorithm reported in the literature—on the CERNET dataset. It shows the size of the original rule set (**R**ules) and the expanded rule set obtained by decomposing range constraints into ternary constraints (**ExpR**ules), the number of nodes in the BDD encoding of the classifier (**Nodes**), the compressed rule set size, compression ratio, and the run time of MDTC (based on [8]) and Flint. As can be seen from the table, Flint improves the average compression ratio of MDTC by 21% on average on these rule sets. Table II reports results for synthetic benchmarks. For space reasons, we report mean values across all rule sets in each benchmark. The first row ('acl-small') shows results for the synthetic dataset from [8]. Flint dramatically outperforms MDTC on this dataset, achieving 13x compression on average, while Zhu et al. report 0.45 mean compression ratio for MDTC. The remaining rows show results for the 5 new synthetic datasets. Flint achieves over 2x compression in all cases and 5x-13x compression on the large datasets with > 8000 rules. Next, we apply Flint to three very large real-world customer rule sets with 70K, 599, and 1952 complex rules. Many of these rules have complex Boolean structure, so that their expanded representation, had we constructed it explicitly, would contain 276K, 64467K, and 194660K rules respectively. The header space in these rule sets consists of 9 fields and 366 bits. Flint compressed the first rule set into 3636 ternary rules; however on the two other rule sets we interrupted Flint after generating 200K rules. Clearly, a more efficient compression technique is required to handle such rule sets. We develop such a technique in the next section.

IV. Compression via functional decomposition

The following example illustrates the main source of explosion in the number of rules in the real-world rule sets. Consider a rule (m, a), where m is of the form $m_{X_1} \wedge m_{X_2}$, where m_{X_1} and m_{X_2} are arbitrary constraints over header fields X_1 and X_2. Assume that m_{X_1} and m_{X_2} can be decomposed into i and j cubes respectively. In this case, an optimal ternary encoding of rule (m, a) may require up to $i \times j$ ternary rules. Alternatively, it can be decomposed into a sequence of two classifiers that check that header X_1 is in m_{X_1} and X_2 is in m_{X_2} respectively. This requires a total of $i + j$ rules. Furthermore, TCAM tables matching individual header fields are narrower, and hence require less TCAM memory, than the monolithic table. This optimization is compatible with modern network switches, which support both pipelined classifiers and configurable-width TCAMs [10].

In general, we would like to decompose a classifier $f : B^n \rightarrow A$ into a sequence of classifiers f_i, $i \in [1, l]$ where f_i matches on a single header field X_i and returns a tag T_i used as input to the next classifier: $f(X_1, \ldots, X_l) \equiv f_l(f_{l-1}(\ldots f_2(f_1(X_1), X_2), \ldots), X_l)$.

Algorithm 2 Decompose classifier f into $f_1, \ldots, f_l$

```
1: function DecomposeClassifier(f)
2:     g ← f
3:     for i = 1 to l − 1 do
4:         (f_i, g) ← Decompose(g, X_i)
5:     f_l ← g
6:     return (f_1, ..., f_l)
```

Name	#R	#ExpR	#Nodes	Flint (rules)	Flint-DC		
					Rules	Ratio	Time(s)
real-world1	70.25K	275.99K	6830	3636	2680	0.00971	392
real-world2	599	64467.64K	171.60K	–	10.71K	0.00017	3118
real-world3	1952	194660.96K	282.68K	–	28.03K	0.00014	3995

TABLE III: Large real-world rule sets.

Example 3: Figure 1(d) shows a decomposition of the classifier in Figure 1(a) into a sequence of two classifiers. The first classifier summarizes the values of variables x_1, x_2 using tag t, which is passed as input to the second classifier.■

We perform the decomposition iteratively, one field at a time, as shown in Algorithm 2. The main building block of the algorithm is the DECOMPOSE procedure that decomposes a function $g(Z_1, Z_2)$ into a pair of functions (g_1, g_2) such that $g(Z_1, Z_2) \equiv g_2(g_1(Z_1), Z_2)$. Notice that this is exactly the *functional decomposition* operation, thoroughly studied in logic synthesis [11]. We use a classical BDD-based functional decomposition algorithm by Lai et al. [4], outlined below.

Consider BDD representation $G(Z_1, Z_2, W)$ of function g (variables W encode the output of g), ordered with variables Z_1 on top. Consider the *cut* of the graph below variables Z_1 and the set of *boundary nodes* $v_1, \ldots, v_j$, located below the cut that have incoming edges from nodes above the cut. Figure 1(b) shows an example cut with boundary nodes v_1 and v_2. An assignment of Z_1 corresponds to a path from the root of the BDD to some nodes v_i. Assignments that lead to the same v_i are indistinguishable to G and can be encoded using one tag. Conversely, due to the canonicity of BDDs, assignments that lead to different v_is are not equivalent and are encoded with distinct tags. Thus, we have partitioned valuations of Z_1 into j equivalence classes. We denote $DD(v_i)$ a subgraph of G that consists of all paths from the root to v_i. Converting $DD(v_i)$ into canonical form and replacing v_i by terminal node $\top$, we obtain a BDD for the ith equivalence class.

We introduce fresh variables T, $|T| = \lceil log(j) \rceil$, to encode the j equivalence classes, and add constraints to G to compute the value of T: $G'(Z_1, T, Z_2, W) = G(Z_1, Z_2, W) \wedge \bigwedge_{i \in [1,j]} [DD(v_i) \Rightarrow (T = i)]$. We finally obtain the decomposition of G into $G_1(Z_1, T) = \exists Z_2, W.\ G'(Z_1, T, Z_2, W)$ and $G_2(T, Z_2, W) = \exists Z_1.\ G'(Z_1, T, Z_2, W)$, where G_1 computes the tag that encodes the equivalence class of variables Z_1, whereas G_2 evaluates the original function G, using the equivalence class of Z_1 instead of its concrete value. The final step of our compression procedure, after computing $f_1, \ldots, f_l$, is to encode each f_i into ternary rules using Algorithm 1.

Performance We applied the decomposition algorithm (**Flint-DC**) to the three real-world benchmarks. We decompose fields in the order in which they appear in the BDD encoding of the classifier, i.e., the order produced by group sifting. Flint-DC produces compact ternary rule sets in all three cases (Table III). The biggest generated rule set contains 28K rules, which fits readily in the TCAM of modern switches [10]. Due to the lack of space, we do not report results for smaller rule sets from Section 3, where Flint-DC performs on a par with Flint. This indicates that functional decomposition works best for large rule sets. The following observation explains the effectiveness of Flint-DC on real-world rule sets. The algorithm depends on the ability to factor the set of values of a field into few equivalence classes, one for each boundary node between BDD layers. While some BDD layers in our rule sets have many nodes, boundary layers are always very thin. For example, in our largest rule set, only 6 bits per field were required on average to encode equivalence classes.

V. Related work

There exists a body of research on packet classification. See [9], [12]–[14] for a sample of work in this area. TCAM-based rule set compression is a special case of the problem, where the resulting classifier is encoded using ternary rules. Previous approaches represent sets of rules using lists [15]–[18] or trees [8], [19]. We show that a BDD-based representation allows computing more compact rule sets efficiently. Norige *et al.* [20] compile classifiers to BDD, but then decompile the BDD back into a tree or list representation to perform compression. In contrast, our algorithms are fully symbolic, i.e., operate on the BDD representation only. Prakash *et al.* [21] use BDDs as intermediate representation in converting packet classifiers to circuits. Smolka *et al.* [22] introduce a variant of BDDs, Forwarding Decision Diagrams, as an intermediate representation for compiling a high-level network programming language into OpenFlow. They extract cubes from FDDs via path enumeration. In contrast, Flint generates fewer larger cubes by generalizing them to prime implicants. Meiners et al. compile packet classifiers to FDDs and slice them into a series of smaller classifiers. The algorithms they use to decompose and compress classifiers differ from ours. For example, they construct a separate FDD for each boundary node and concatenate these FDDs to obtain a classifier. In contrast, Flint compiles each classifier in the decomposition into a monolithis BDD and applies prime decomposition to it. Petrovska *et al.* [23], proposed a state of the art SAT-based logic minimization algorithm. In our preliminary experiments, the algorithm does not scale on large benchmarks.

The notion of sequential cover for rule set compression was introduced in [3]. We propose an efficient BDD-based method of computing sequential cover. Chang *et al.* [24] proposed an optimized encoding of range constraints into ternary constraints. This work is complementary to ours and can potentially be used to replace the simple encoding we use.

VI. Conclusion

We developed two algorithms that significantly improve the state of the art in TCAM-based packet classification. While most previous solutions rely on specialized data structures and algorithms, we reduce the problem to standard problems in logic minimization and solve it using symbolic logic minimization techniques.

REFERENCES

[1] K. Kogan, S. I. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards efficient implementation of packet classifiers in sdn/openflow," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013* (N. Foster and R. Sherwood, eds.), pp. 153–154, ACM, 2013.

[2] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[3] R. McGeer and P. Yalagandula, "Minimizing rulesets for TCAM implementation," in *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pp. 1314–1322, IEEE, 2009.

[4] Y. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," in *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993.* (A. E. Dunlop, ed.), pp. 642–647, ACM Press, 1993.

[5] O. Coudert, "Two-level logic minimization: An overview," *Integration, the VLSI Journal*, vol. 17, pp. 97–140, Oct. 1994.

[6] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0.."

[7] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993* (M. R. Lightner and J. A. G. Jess, eds.), pp. 42–47, IEEE Computer Society / ACM, 1993.

[8] H. Zhu, M. Xu, Q. Li, J. Li, Y. Yang, and S. Li, "MDTC: an efficient approach to tcam-based multidimensional table compression," in *Proceedings of the 14th IFIP Networking Conference, Networking 2015, Toulouse, France, 20-22 May, 2015* (R. Kacimi and Z. Mammeri, eds.), pp. 1–9, IEEE Computer Society, 2015.

[9] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.

[10] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013* (D. M. Chiu, J. Wang, P. Barford, and S. Seshan, eds.), pp. 99–110, ACM, 2013.

[11] R. L. Ashenhurst, "The decomposition of switching functions.," in *International Symposium on the Theory of Switching*, pp. 74 – 116, 1959.

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

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

[14] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pp. 648–656, IEEE, 2009.

[15] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary cams can be smaller," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006* (R. A. Marie, P. B. Key, and E. Smirni, eds.), pp. 311–322, ACM, 2006.

[16] A. X. Liu and M. G. Gouda, "Complete redundancy removal for packet classifiers in tcams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 4, pp. 424–437, 2010.

[17] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 488–500, 2012.

[18] J. Daly, A. X. Liu, and E. Torng, "A difference resolution approach to compressing access control lists," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 610–623, 2016.

[19] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: a systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, 2010.

[20] E. Norige, A. X. Liu, and E. Torng, "A ternary unification framework for optimizing tcam-based packet classification systems," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 657–670, 2018.

[21] A. Prakash, R. Kotla, T. Mandal, and A. Aziz, "A high-performance architecture and bdd-based synthesis methodology for packet classification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 698–709, 2003.

[22] S. Smolka, S. A. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for netkat," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015* (K. Fisher and J. H. Reppy, eds.), pp. 328–341, ACM, 2015.

[23] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, and P. Ienne, "Progressive generation of canonical irredundant sums of products using a SAT solver," in *Advanced Logic Synthesis* (A. I. Reis and R. Drechsler, eds.), pp. 169–188, Springer, 2018.

[24] Y. Chang, C. Su, Y. Lin, and S. Hsieh, "Efficient gray-code-based range encoding schemes for packet classification in TCAM," *IEEE/ACM Trans. Netw.*, vol. 21, no. 4, pp. 1201–1214, 2013.