# Palette: Distributing Tables in Software-Defined Networks

Yossi Kanizo
Technion
ykanizo@cs.technion.ac.il

David Hay
Hebrew University
dhay@cs.huji.ac.il

Isaac Keslassy
Technion
isaac@ee.technion.ac.il

***Abstract*—In software-defined networks (SDNs), the network controller first formulates abstract network-wide policies, and then implements them in the forwarding tables of network switches. However, fast SDN tables often cannot scale beyond a few hundred entries. This is because they typically include wildcards, and therefore are implemented using either expensive and power-hungry TCAMs, or complex and slow data structures.**

**This paper presents the Palette distribution framework for decomposing large SDN tables into small ones and then distributing them across the network, while preserving the overall SDN policy semantics. Palette helps balance the sizes of the tables across the network, as well as reduce the total number of entries by sharing resources among different connections. It copes with two NP-hard optimization problems: *Decomposing* a large SDN table into equivalent subtables, and *distributing* the subtables such that each connection traverses each type of subtable at least once. To implement the Palette distribution framework, we introduce graph-theoretical formulations and algorithms, and show that they achieve close-to-optimal results in practice.**

## I. Introduction

### A. Background

*Software-defined networking* (SDN) in general, and OpenFlow [1], [2] in particular, provide an abstraction of network devices and operations. This abstraction eases the development of new network protocols and policies. These protocols are implemented through the network *controller*, a single centralized device with a global view of the entire network. The network controller can be seen as a compiler that translates the abstract policies provided by network designers into specific rules in the table of each network switch.

Previous works typically assumed that the table of each switch can hold an infinite number of rules, which makes the compiler easy to design. In practice, however, this assumption does not hold, and the switch table sizes can become a significant bottleneck to scaling SDN networks. We note that many of these tables are implemented using ternary content-addressable memory (TCAM), which is extremely power-hungry and therefore of limited size. Typical implementations of OpenFlow, for example, limit the number of entries in each such table to only 750 [3], while handling about 100,000 concurrent flows.

*This paper introduces the Palette framework for distributing these rules into a network of heterogeneous switches with tables of limited size, while preserving the semantics of the SDN policy.* The Palette distribution framework is generic in the sense that it does not rely on the exact meaning of the rules, as long as the rules do not determine the routing of the packet. More specifically, the controller application should only specify whether the policy is routing/forwarding-agnostic or not, and should not deal with the implementation complexity of the distribution across the network switches. This is especially useful when the network topology changes, or equipment is replaced.

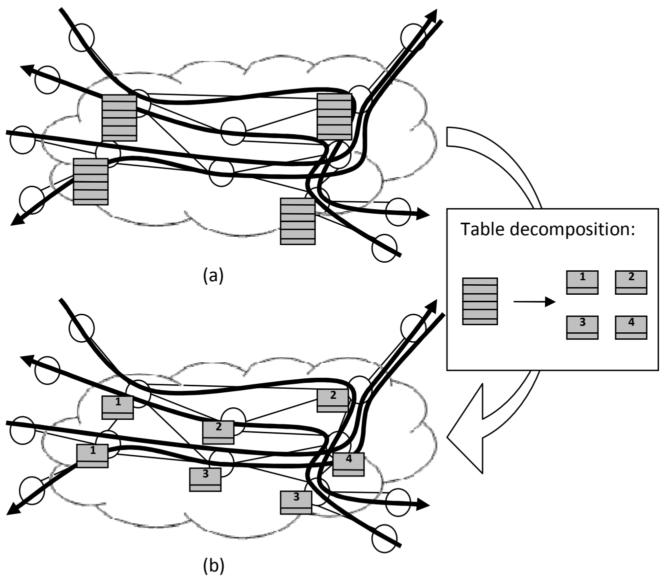

Fig. 1. (a) A common setting in which tables are installed at the network ingress nodes; (b) The result of applying Palette. Tables are decomposed into smaller subtables of different types (a.k.a. colors), which are then distributed across the network. A packet along each path meets each type of subtable at least once.

### B. The Palette Framework

We define an SDN policy as a collection of rules. Each rule consists of a *(pattern, action)* pair, i.e. a pattern of specific bits in the packet header along with an action to take upon a pattern match (e.g. drop the packet or increment a counter of some measurement). For aggregation purposes, *don't-care* bits, denoted by "$*$", are allowed in the pattern. Therefore, a given packet header may match more than one rule, and in that case an action is taken according to the highest-priority rule. Typically, the SDN tables evolve over time (that is, new rules are added and some rules are deleted). In addition, occasionally a switch can send notifications (e.g., measurements taken in one of its built-in counters) to the controller.

Palette takes advantage of the fact that the controller has a global view of the network, and therefore, knows the paths taken by all packets. This allows us to share resources among different paths in an efficient way, by using the same rules for different paths in any common switch.

Fig. 1 illustrates our approach in a common setting used for *access control*. Access control consists of determining whether a given packet is permitted in the network or should be dropped. It is usually made by a switch, a router, or a designated Network Intrusion Detection/Prevention System (NIDS/NIPS) middlebox *at the edge of the network*. Specifically, as illustrated in Fig. 1(a), some access control is performed on all ingress nodes of the network. In an SDN setting, this typically translates into installing a table with all access control rules in all ingress nodes. However, as shown in Fig. 1(b), using our Palette framework, the rules can be distributed across the network switches.

## II. Related Work

Software Defined Networking (SDN) has become an important paradigm in contemporary networks. Its key concept lies in the management of the entire network as a unified abstraction (e.g., in a network controller), and the remote control of the network devices (namely, its switches and routers) through open protocols (such as OpenFlow) [4]. In recent years, SDN technology has been widely deployed in real-life large-scale networks, e.g. Google's G-scale network [5]. Switches and routers that support SDN/OpenFlow are now offered by a large number of vendors (e.g., [3], [6], [7]).

One of the major challenges in SDN is to develop a programming language for its software development. On one hand, this programming language should be sufficiently flexible and rich to allow new network applications, but on the other hand, it should be simple and modular to reduce development and debugging times. Frenetic [8] is a prime example of such a network programming language that gives high-level abstractions to the network programmer. For example, it allows systematic updates [9] and task composition [8]. Perhaps the most closely-related work to ours is the extension of Frenetic to allow policy transformation of rules across the networks [10]. The authors developed a complete and sound set of axioms to allow semantically-preserving rule-rewriting in a single switch or in a chain of switches. Our paper complements this work by providing a specific algorithmic framework in which such a rewriting system can work, and also shows how to spread the resulting rules across the network. An additional key aspect of our paper that can be useful for Frenetic is our order-oblivious decomposition, which facilitates the distribution of subtables across the network.

Another approach for distributing table rules across the network is DIFANE [11]. In DIFANE, special switches (called *authority switches*) are assigned non-overlapping flow ranges. Then, ingress switches redirect packets to the corresponding authority rule and cache the corresponding rule for future packets of the same flow. Our approach avoids the management and redirection overhead as well as duplications of the rules to the ingress switches.

To split the workload between the switches, cSamp [12], a generic framework for network measurement, monitors each flow only in one of the network switches. It uses a hash function with a certain distribution at each switch to determine whether the current switch has to perform the measurement. However, since the hash function is orthogonal to the monitoring rules, each such router needs to store the entire monitoring table, which we want to avoid in our approach.

## III. Order-Oblivious Table Decomposition

This section analyzes two approaches to dividing a large table into $c$ subtables: the *Pivot Bit Decomposition* (PBD) and the *Cut-Based Decomposition* (CBD).

### A. Decomposition Rules

Before starting, note that we only divide rules that correspond to policies which are *marked as safe to divide*. The rest of the rules remain in the corresponding sub-table, and will be re-composed with the rules assigned to that table after decomposition. There is a large body of work of how to compose several policies in one table (e.g., using a Cartesian product of the rules [13]).

Hence, we are left with an arbitrary table that is safe to divide. We assume that this table must be able to match all possible strings. For this, we distinguish between *default* and *non-default* rules. The default rule consists of *don't-care* bits only, and it uses a default blank action (e.g., a *permit* action in ACLs). The non-default rules are all other rules in the table. Clearly, if a default rule exists in the original table, it may be placed only at the end of the table. To follow our convention, after the decomposition we add a default rule automatically to each of the resulting tables.

The correctness of the decomposition implies that each string that matches a non-default rule in the original table, must match a non-default rule in exactly one of the subtables (and the default rule in the other resulting tables). Moreover, strings that match the default rule in the original table, must also match the default rule in all subtables.

### B. Pivot Bit Decomposition

The first method, called *Pivot Bit Decomposition (PBD)*, works by iteratively decomposing one table into two equivalent tables, thus increasing the total number of tables by $1$.

This iterative decomposition is done by selecting one *pivot bit* (equivalently, one column) in the table, and splitting the rules into two sets: the first table holds all rules in which the pivot bit is $0$, while the second table holds all rules in which the pivot bit is $1$. Rules in which the pivot bit is "don't care" ("$*$") are rewritten as two complementary rules: one in which the pivot bit is replaced by $0$ (and therefore, is part of the first table) and another in which it is replaced by $1$ (and therefore, is part of the second table).

Naturally, the efficiency of the decomposition depends on the joint selection of the table and of the pivot bit at each iteration. *Our goal is to greedily minimize the maximum*

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\varphi_1$ | * | 0 | 1 | 0 | * | 0 | 0 |
| $\varphi_2$ | 0 | * | 1 | * | * | * | 0 |
| $\varphi_3$ | * | 1 | * | * | 1 | 0 | 1 |
| $\varphi_4$ | 1 | 1 | 1 | * | 1 | * | * |
| $\varphi_5$ | 1 | 1 | * | 0 | * | * | * |
| $\varphi_6$ | 1 | 0 | 0 | 1 | 0 | 1 | * |
| $\varphi_7$ | * | * | * | * | * | * | * |

Fig. 2. An example rule-set of an SDN table.

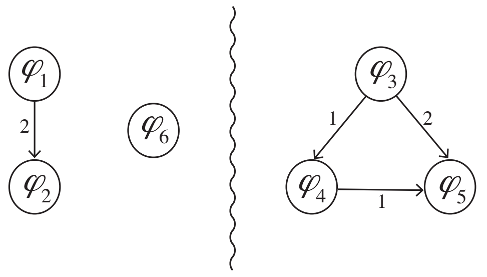

Fig. 3. The dependency graph and cut of the table in Fig. 2.

*table size among all the small tables*. Therefore, we always decompose the largest table. The pivot bit is then selected to minimize the size of the largest table among the two resulting subtables.

Finally, note that if after the decomposition the maximum table size is not reduced, we refrain from decomposing the table and move to the next table. The process ends either when we reach $c$ tables, or when all possible decompositions do not result in table size reduction.

**Example 1.** *We demonstrate our decomposition using the example table depicted in Fig. 2. Assume we want to divide the table into $c = 2$ partitions. We disregard the default rule ($\varphi_7$), and choose bit number 1 as the pivot bit. Rule $\varphi_2$ has * in bit 1, and therefore it is duplicated to rule $\varphi_2' = 001{*}{*}{*}0$ and $\varphi_2'' = 011{*}{*}{*}0$. After the division, the rules $\varphi_1$, $\varphi_2'$, and $\varphi_6$ are assigned to the first sub-table, while the rules $\varphi_2'', \varphi_3, \varphi_4$, and $\varphi_5$ are assigned to the second sub-table. We also need to add default rule $\varphi_7$ to both of the resulting sub-tables, so their final sizes would be 4 and 5, respectively.*

In [14], we have shown the correctness of our decomposition and the fact that it is order-oblivious.

We point out two main drawbacks in the basic PBD scheme. First, the largest subtable may be asymptotically $c$ times larger than the largest subtable in the optimal decomposition, as we show in [14]. Second, the basic PBD scheme divides the table at each iteration so that the maximum size of the resulting two subtable sizes is minimized. Therefore, the sizes of the resulting two subtables after each iteration tend to be almost equal. As a result, when $c$ is not a power of two, it is expected that the partition sizes would be *imbalanced*.

To solve this problem, we generalize the PBD scheme in the following way: Given the target number of subtables $c$, we first find the largest integer $p$ such that $2^p < c$. Then, we find a pivot bit that attempts to divide the table such that the ratio between the resulting table sizes will be $2^p{:}(c{-}2^p)$. [1] We recursively use this generalization of PBD on each of the two subtables, aiming to decompose the first subtable into $2^p$ smaller subtables, and the second subtable into $c - 2^p$ ones.

[1]For example, for $c = 7$, the goal is to have two tables, one holding approximately 4/7 of the entries and the other 3/7. Thus the ratio between the tables is 4:3, while the basic PBD scheme aims to achieve a ratio 1:1 between the tables.

## C. Cut-Based Decomposition

We now offer a second approach to decomposing the table, called *cut-based decomposition* (CBD). This decomposition is based on representing the set of rules in a *directed* dependency graph.

As illustrated in Fig. 3, which shows the dependency graph of the table in Fig. 2, the nodes in this graph represent the rules. Moreover, there is an edge from node $u$ to node $v$ if and only if rule $u$ has higher priority than rule $v$, and there is at least one key that matches both rules. Namely, the edges of the graph represent *dependencies* between the rules. Our goal is to decompose the graph, which corresponds to the original table, into component subgraphs, which will correspond to the resulting subtables, such that there are no edges between the components. That is, no key matches rules in different components.

First, since all rules match the default rule, and it is allowed to match the default rule in all subtables, we omit the node corresponding to this default rule in the graph.

Second, we assign a weight to each edge. The weight corresponds to the cost of *breaking* this edge: an edge can be broken by changing the rules in such a way that no dependency remains between the rules, and the semantic is preserved.

Specifically, let $b_i^u$ denote the $i$-th bit of node $u$. For any node $v$, define the following set of dependency bits: $C_{u,v} = \{i \mid b_i^v = * \text{ and } b_i^u \neq *\}$. The weight of the edge between node $u$ and node $v$, denoted by $w(u, v)$, is $|C_{u,v}|{-}1$. The weight $w(u, v)$ corresponds to a possible way of resolving the dependency between $u$ and $v$ by adding $w(u, v)$ nodes to the graph: for each bit $i$ in $C_{u,v}$, we can write a rule that is identical to $v$, except the $i$-th bit that is replaced by $1 - b_i^u$. These rules do not have a dependency with $u$. In addition, each key that matches $v$ in the original rule-set will match at least one of these rules. Note that when removing a single edge from the graph, we create a new graph: edges that touch node $v$ in the original graph might be duplicated to the new $|C_{u,v}|{-}1$ nodes; the weight of these duplicated edges can only decrease.

Another operation that we also allow in this scheme is a *node expansion*, that is, given a set of $t$ * bits in some rule, we replace the rule with $2^t$ new rules by replacing the * bits with a binary enumeration of possible combinations of 0s and 1s. By definition, this operation does not change the semantics

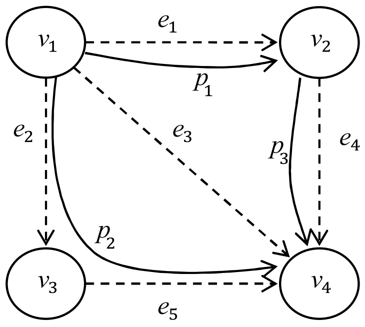

Fig. 4. Illustration of the model. Network $G = \langle V, E \rangle$ has vertex set $V = \{v_1, \ldots, v_4\}$ for the switches, and an edge set $E = \{e_1, \ldots, e_5\}$ for the links. There are three paths in the path set $P = \{p_1, p_2, p_3\}$, and, for example, $S(p_1) = \{v_1, v_2\}$ and $L(p_1) = \{e_1\}$.

of the original table. However, it reduces the connectivity of the dependency graph, facilitating the graph partitioning.

Since graph partitioning is an NP-hard problem, we have obtained approximate solution using the off-the shelf program METIS [15], as further detailed in [14].

## IV. The Rainbow Path Coloring Problem

After showing how to decompose the initial table into subtables, we now turn to show *how to spread the subtables in the network*.

We model the network as a directed graph $G = \langle V, E \rangle$ with a vertex set $V$ and an edge set $E$, where $V = \{v_1, \ldots, v_n\}$ represents the set of $n$ switches and $E = \{e_1, \ldots, e_m\}$ the set of $m$ links. Further, let $P = \{p_1, \ldots, p_f\}$ be the set of all flow paths in the network.

Our basic problem is to maximize the number of *colors* $c$, such that we can color each switch in one of these colors (or with no color), subject to the constraint that each path must contain all $c$ colors. In [14], we have formally defined the problem, and shown that it is *NP-hard* for general graphs. Therefore, we now want to introduce a suboptimal iterative GREEDY algorithm.

At each iteration, which corresponds to a new color, GREEDY continuously picks uncolored nodes one by one, until each path contains at least one of the picked nodes in this iteration. In such a case, the nodes picked are colored with a new color, and the algorithm continues to the next iteration. If at some iteration, even after picking all uncolored nodes, there is at least one path that does not contain any of the picked nodes, then those nodes remain uncolored ($\perp$), and the algorithm stops. Note that, in any case, the algorithm never stops in the first iteration, that is, it always succeeds to color the nodes using at least one color.

We next present two variants of this algorithm, which differ in the way the nodes are selected at each iteration. In the first variant, which we call 1-GREEDY, at each choice, we pick the node $v$ that maximizes the number of paths that contain $v$ but do not contain the new color. 1-GREEDY runs in $O\left(n^2 \cdot |P|\right)$ time complexity.

The second variant, $q$-GREEDY, generalizes the 1-GREEDY algorithm by considering, at each step, a set of up to $q$ nodes (instead of a single node). It chooses the set of nodes that maximizes the number of paths for which there is at least one node in the set. Notice that the larger $q$ is, the longer it takes to run $q$-GREEDY, whose time complexity is $O\left(n^{q+1} \cdot |P|\right)$.

The next example shows an execution of 1-GREEDY and an execution of 2-GREEDY that differ in their outcome. This demonstrates the tradeoffs in fixing the parameter $q$.

**Example 2.** *Consider the example in Fig. 4. We first run* 1-GREEDY. *At the first iteration, nodes $v_1$, $v_2$, and $v_3$ belong to two paths; assume that* 1-GREEDY *first picks $v_1$ and colors it in the first color. Then, in order to color $p_3$, it picks $v_2$ and colors it in the first color as well. Note that all nodes of the first path $p_1$ are now colored, implying that any additional iteration will fail, resulting in a valid coloring of only one color.*

*In contrast,* $q$-GREEDY *with $q = 2$ first picks nodes $v_2$ and $v_3$, since all three paths traverse through either one of these nodes. Then, $v_1$ and $v_4$ can be colored with an additional color, resulting in a valid coloring with 2 colors.*

Note that although the presented problem is NP-hard, we are able to find the optimal solution for small instances of the problem by using an algorithm based on dynamic programming [14]. This algorithm, which might be applicable to some real-life networks, is used in Section V as a baseline for evaluating the performance of the 1-GREEDY and $q$-GREEDY. Specifically, our simulations show an average margin of error within 2%.

## V. Experimental Results

We now turn to evaluating our algorithms. We first check the decomposition algorithms (Section III) and then the table distribution algorithms (Section IV).

### A. Table Decomposition

We first consider the PBD and CBD algorithms for decomposing tables, as presented in Section III.

We define the *quality* of a table decomposition algorithm as the ratio between the number of rules in the original table, and the product of the largest resulting subtable size by the number of subtables $c$. The quality is therefore between 0 and 1, where higher quality values implies a better decomposition. Specifically, a quality of 1 means that the largest subtable size has exactly an ideal fraction $1/c$ of the number of the original rules. Note that this quality can only be used to compare among different algorithms for the same value of $c$. Furthermore, it is most likely that the quality is decreased when $c$ is increased.

We compare PBD and CBD algorithms with a *bit groups* algorithm based on [16]. Through an exhaustive search, this algorithm selects the $\log_2 c$ pivot bits that maximize the quality. Thus, it only works for values of $c$ that are powers of 2.

Fig. 5 shows the quality of the three algorithms as the number $c$ of partitions grows. For the simulations, we have created 100 random logically-minimized rule-sets with 12 bits

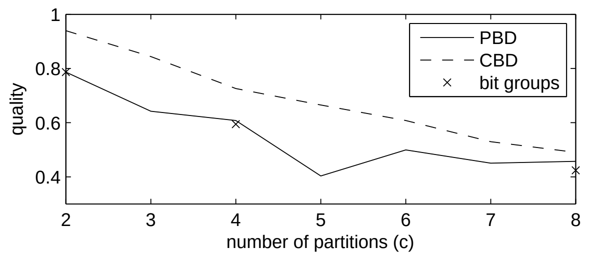

Fig. 5. Evaluation of the quality metric of the PBD, CBD, and a non-iterative algorithm that selects all pivot bits at once [16], [17]. The input is a synthetic rule set.

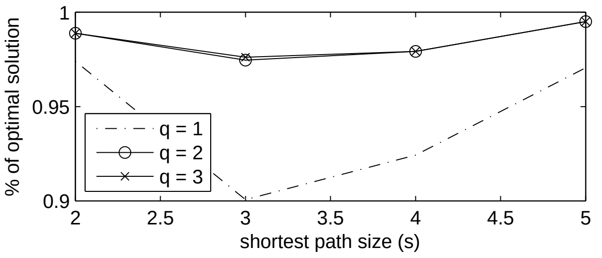

Fig. 6. Fraction of the optimal valid coloring size that can be achieved by the 1-GREEDY, 2-GREEDY, and 3-GREEDY algorithms. Parameters are $n = 7$, $f = 7$ and $p_n = \frac{5}{7}$.

and 30 rules each. PBD slightly outperforms *bit groups*, except when $c = 2$, where they perform similarly. CBD clearly outperforms both over the entire range.

We have also evaluated in [14] PBD and CBD with the twelve standard classification benchmark rule-sets of ClassBench [13], [18]. We find that the quality of our schemes is greatly affected by the specific rule-set. Namely, CBD outperforms PBD when its dependency graph is sparse (e.g., with an average degree of less than 3), while PBD is better than CBD otherwise.

### B. Table Distribution

To analyze the performance of our table distribution algorithm, we have produced random instances in the following manner: given a number of switches $n$ and a number of paths $f$, we add each switch to each path with probability $p_n$ independently of the other switches or paths. In this section, we study our algorithms on small networks, where we are able to compute the *optimal* valid coloring size.

Fig. 6 shows the number of colors found by the greedy approach in terms of percentage of the optimal solution. The parameters in this case are $n = 7$, $f = 7$ and $p_n = \frac{5}{7}$, and we ran the simulation 1000 times. Our results yield that, in these cases, that the greedy approach finds a valid coloring whose size exceeds (on average) 98% of the optimal solution.

## VI. CONCLUSION

This paper proposed Palette, a framework to decompose and distribute SDN tables across the network. Palette is especially important as switch table sizes can become a bottleneck in scaling SDNs. Moreover, it facilitates handling the heterogeneity of switches in the network and the changes of equipment.

We modeled the problem in a graph-theoretic manner, and proposed several algorithms, both for decomposing one table to semantically-equivalent subtables and for spreading these subtables across the network. Our algorithms were evaluated both under random and real-life instances.

## ACKNOWLEDGMENT

The work was partly supported by the European Research Council Starting Grants n°$210389$ and n°$259085$, the Alon Fellowship, the ATS-WD Career Development Chair, the Loewengart Research Fund, the Intel ICRI-CI Center, the Israel Ministry of Science and Technology, and the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11). We would like to thank Jennifer Rexford for her useful comments on an earlier version of this paper, and Anat Bremler-Barr for kindly accepting to run several simulations.

## REFERENCES

[1] "Openflow switch specification." [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] "NEC Univerge PF5240 and PF5820." [Online]. Available: http://www.openflow.org/wp/switch-nec/

[4] S. Shenker, "The future of networking, and the past of protocols," in *Open Networking Summit*, 2011.

[5] "Google G-scale network." [Online]. Available: http://www.eetimes.com/electronics-news/4371179/Google-describes-its-OpenFlow-network

[6] "HP Procurve Switch." [Online]. Available: http://www.openflow.org/wp/switch-hp/

[7] "Interop 2012 openflow roundup," 2012. [Online]. Available: http://www.openflowhub.org/blog/blog/2012/05/10/interop2012/

[8] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *SIGPLAN ICFP*, 2011.

[9] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM Sigcomm*, 2012, pp. 323–334.

[10] N. Kang, J. Reich, J. Rexford, and D. Walker, "Policy transformation in software defined networks," in *ACM Sigcomm*, 2012, pp. 309–310.

[11] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM Comput. Commun. Rev.*, vol. 41, no. 4, 2010.

[12] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "cSamp: A system for network-wide flow monitoring," in *USENIX NSDI*, 2008.

[13] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst-case TCAM rule expansion," *IEEE Trans. Comput.*, 2012.

[14] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," Technion, Tech. Rep. TR12-05, 2012. [Online]. Available: http://webee.technion.ac.il/~isaac/papers.html

[15] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[16] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang, "Dppc-re: TCAM-based distributed parallel packet classification with range encoding," *IEEE Trans. Comput.*, vol. 55, pp. 947–961, 2006.

[17] K. Zheng, C. Hu, H. Liu, and B. Liu, "An ultra high throughput and power efficient TCAM-based ip lookup engine," in *IEEE Infocom*, 2004.

[18] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," in *IEEE Infocom*, 2005.