Policy Transformation in Software Defined Networks

Nanxi Kang
Computer Science Department
Princeton University
nkang@cs.princeton.edu
Advisor: Jennifer Rexford

1. INTRODUCTION

Traditional networks are built out of special-purpose devices running distributed protocols that provide functionality such as topology discovery, routing, traffic monitoring, and access control. In these devices, the distributed control-plane software that computes routes is tightly coupled with the data-plane logic that forwards packets. Moreover, network operators must separately configure every protocol on each individual device. Recent years, however, have seen growing interest in software-defined networks (SDNs), in which a logically-centralized controller manages the packet-processing functionality of a distributed collection of switches. SDNs make it possible for programmers to control the behavior of the network directly, by configuring the packet-forwarding rules installed on each switch [1]. Note that although the programmer has the illusion of centralized control, the controller is often replicated and distributed for scalability and fault tolerance [2].

SDNs can both simplify existing applications and also serve as a platform for developing new ones. For example, to implement shortest-path routing, the controller can calculate the forwarding rules for each switch by running algorithm on the graph of the network topology instead of using a more complicated distributed protocol [3]. To conserve energy, the controller can selectively shut down links or even switches after directing traffic along other paths [4]. To enforce fine-grained access control policies, the controller can consult an external authentication server and install custom forwarding paths for each user [5]. To balance the load between back-end servers in a data center, the controller can split flows over several server replicas and migrate flows to new paths in response to congestion [6, 7].

But although SDNs makes it possible to program the network, it does not make it easy. Today's SDN controller platforms [2, 8, 9, 10, 11, 12, 13, 14] force applications to manage the network at the level of individual switches by representing a high-level policy in terms of the rules installed in each switch. This forces programmers to reason about low-level details, such as the space

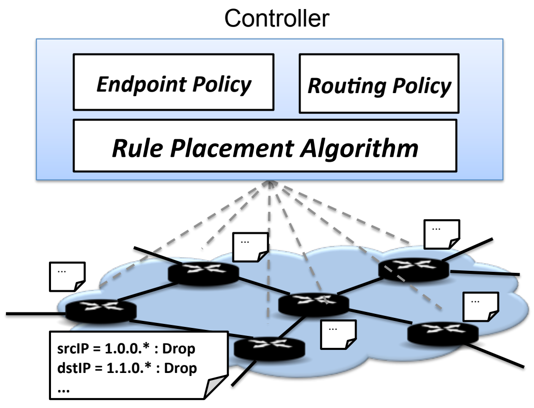


Figure 1: High-level policy and low-level rule placement

limits on each switch. More specifically, many SDN applications require rules that match on multiple packet header fields, with "wildcards" for some bits. For example, access-control policies match on the "five tuple" of source and destination IP addresses and port numbers and the protocol [5, 15]. These rules are naturally supported using Ternary Content Addressable Memory (TCAM), which can read all rules in parallel to identify the matching entries for each packet. However, TCAM is expensive and power hungry. The merchant-silicon chipsets in commodity switches typically support just a few thousand TCAM entries.

Rather than grappling with TCAM sizes, we argue that SDN application programmers should define high-level policies and have the controller platform manage the placement of rules on switches. We, and more broadly the community at large [16, 3, 17, 18], have observed that such high-level policies may be specified in two parts, as shown in Figure 1:

- **An endpoint policy:** Endpoint policies, like access control and load balancing, view the network as *one big switch* that hides internal topology details. The policy specifies which packets to drop, or to forward to specific egress ports, as well as any modifications of header fields.
- **A routing policy:** The routing policy specifies what paths traffic should follow through the net-

work between the ingress and egress ports of the big switch. The routing policy is driven by traffic-engineering goals, *e.g.,*minizing latency and bandwidth allocation.

Expressing these two parts of the policy separately enables greater modularity, allowing different parties to control different aspects of network management. Given two such specifications, the controller platform can apply a *rule-placement algorithm* to generate switch-level rules that realize both parts of the policy correctly, while staying within the table-size constraints. For example, if an access-control policy cannot "fit" in the first switch, the controller can place as many rules as possible at the edge, while dropping some packets at the second hop in the path.

Optimizing the placement of rules is challenging. Minimizing the number of rules for a *single* switch is computationally difficult [19], though effective heuristics exist [20].

In this paper, we take on the general challenge of solving the rule placement problem for an entire *network* of switches, given independent endpoint and routing policies. We make a number of important contributions that range from new algorithm design, to complexity analysis to implementation and empirical analysis on both synthetic and real-world data sets.

In the next section, we formally introduce the rule-placement problem. Then, Section 3 presents an algorithm that optimizes rule placement for general topologies. Our experiments with real firewall and routing policies, as well as synthetic benchmarks, are discussed in Section 4. We conclude the paper in Section 5.

Related work: The paper is most closely related to vCRIB [21] and Palette [22], two recent papers that study distributed rule placement. While vCRIB takes an endpoint policy as input, the algorithm assumes control over routing to direct traffic through intermediate switches that enforce portions of the policy. That is, vCRIB saves table space at the expense of longer paths and higher network load. In contrast, we view routing policy something the SDN *application* should control, based on higher-level goals like traffic engineering, *e.g.,*minizing latency and bandwidth allocation. As such, our algorithms must grapple with optimizing rule placement while obeying a specified routing policy.

Similar to our solution, Palette gives the SDN application complete control over both the endpoint policy and routing. However, Palette enforces the *entire* endpoint policy (such as access control) on *every* path through the network, even though only a small subset of the traffic actually traverses each path.

2. THE RULE-PLACEMENT PROBLEM

Abstractly, the rule placement problem converts a global, high-level policy to a low-level set of rules for each switch. In this section, we introduce a precise formulation of the problem and its constituent parts.

Network topology: The underlying network consists of n switches, each with a set of ports. We refer a port at a switch as a *location* (loc). The locations that are connected to the outside world are *exposed locations.* A packet will enter the network from an exposed location called *ingress* and leave at an exposed location called *egress.*

Packets: A packet (pkt) includes multiple header fields. Examples of header fields include source IP (src_ip) and destination IP (dst_ip). Switches decide how to handle traffic based on the header fields, and do not modify any other part of the packet; hence, we equate a packet with its header fields.

Switches: Each switch has a single, prioritized list of rules $[r_1, \ldots, r_k]$, where rule r_i has a predicate $r_i.p$ and an action $r_i.a$. A *predicate* is a boolean function that maps a packet header and a location (pkt, loc) into $\{\text{true}, \text{false}\}$. A predicate can be represented as a conjunction of clauses, each of which does *prefix* or *exact* matching on a single field or location. An action could be either "drop" or "modify and forward". The "modify and forward" action specifies how the packet is modified (if at all) and where the packet is forwarded. Upon receiving a packet, the switch identifies the *highest-priority* rule with a matching predicate, and performs the associated action. A packet that matches no rules is dropped by default.

Endpoint policy (Q): The endpoint policy operates over the set of exposed locations as if they were ports on one big abstract switch. A endpoint policy is a prioritized list of rules $Q \triangleq [r_1, ..., r_m]$, where $m = \|Q\|$ is the number of rules. We assume that at the time a given policy Q is in effect, each packet can enter the network through at most one ingress point (*e.g.,* the port connected to the sending host, or an Internet gateway).

Routing policy (R): A routing policy R is a function $R(\text{loc}_1, \text{loc}_2, \text{pkt}) = s_{i_1} s_{i_2} ... s_{i_k}$, where loc_1 denotes packet ingress, loc_2 denotes packet egress.The sequence $s_{i_1} s_{i_2} ... s_{i_k}$ is the path through the network. The routing policy may direct all traffic from loc_1 to loc_2 over the same path, or split the traffic over multiple paths based on packet-header fields.

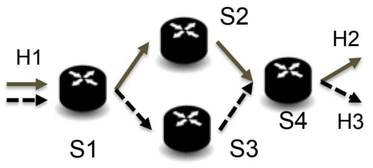
Rule-placement problem: The inputs to the rule-placement problem are the network topology, the endpoint policy Q, the routing policy R, and the maximum number of rules each physical switch can hold. The output is a list of rules on each switch such that the network (i) obeys the endpoint policy Q, (ii) forwards the packets over the paths specified by R, and (iii) does not exceed the rule space on each switch.

3. RULE PLACEMENT ALGORITHMS

Our algorithm solves the rule placement over a gen-

r_1 : (dst_ip = 00*, ingress = H_1 : Permit, egress = H_2)
r_2 : (dst_ip = 01*, ingress = H_1 : Permit, egress = H_3)

(a) An example endpoint policy Q



(b) An example routing policy R

$P_1 = s_1s_2s_4, D_1 = \{\text{dst\_ip} = 00*\}$
$P_2 = s_1s_3s_4, D_2 = \{\text{dst\_ip} = 01*\}$

(c) Paths and flow spaces computed from Q and R

Figure 2: An example decomposition

eral topology in three steps: first, the topology is decomposed into multiple paths and policy for each path is computed; then, the rule capacity of each switch is split in a balanced way and assigned to every path crossing it; finally, the rule placement for each path is solved independently.

3.1 Decomposing Policies into Paths

Our algorithm starts by finding all paths in the graph $P_1, P_2, ..., P_\ell$, where path P_i is a chain of switches $s_{i_1}s_{i_2}...s_{i_k}$ connecting an exposed location to another. By examining the "cross-product" of endpoint policy Q and routing policy R, we can associate path P_i with a *flow space* D_i, *i.e.,* all the packets that use P_i belong to D_i.[1] An example is shown in Figure 2.

Any two flow spaces D_i and D_j are disjoint, since each packet can enter the network via a single ingress location and exit at most one egress location (Section 2). Thus, the endpoint policy is converted into a set of subpolicies Q_i only defined on flow space D_i:

$$Q_i(\text{pkt}) = \begin{cases} Q(\text{pkt}) & \text{if pkt} \in D_i \\ \perp & \text{otherwise,} \end{cases} \tag{1}$$

where $\perp$ means no operation is performed. Based on this representation, we reduce a general graph problem into a set of path problems: for each path P_i, the endpoint policy is Q_i and the routing policy is simple—as long as a packet is not dropped, it is passively forwarded to the next hop.

However, in addition to solving ℓ instances of rule placement over a path, we must ensure that switches correctly forward traffic along all paths, *i.e.,* routing policy is realized. The algorithm achieves this by placing low-priority *default rules* on switches. These default rules enforce "forward any packets in D_i to the next hop in P_i", such that packets that are not handled by higher-priority rules will traverse along the desired path.

[1]we can associate a dropped packet with the most natural path it belongs to (*e.g.,* the path taken by other packets with the same destination address

$$\begin{array}{lll} \text{max:} & \perp & \\ \text{s.t:} & \forall i \le n: \ \sum_{j \le \ell} h_{i,j} \cdot x_{i,j} \le 1 & \text{(C1)} \\ & \forall j \le \ell: \ \sum_{i \le n} h_{i,j} \cdot x_{i,j} \cdot c_j \ge m_j & \text{(C2)} \end{array}$$

Figure 3: The linear program formulation for the resource allocation

3.2 Allocating Resource Across Path

Ideally, we would simply solve the rule-placement problem separately for each path and combine the results into a complete solution, since rules placed by one path do not affect another path. However, multiple paths can traverse the same switch. The paths do need to share *rule space* in the common switch. For each path P_i, we need to allocate enough rule space at each switch in P_i to successfully implement the policy Q_i, while respecting the capacity constraints of the switches.

Estimating Resource Demands Per Path Checking the feasibility of all possible capacity allocation would be expensive. Fortunately, in practice, we observe that the *total* available rule space across all of the hops in the path is a relatively good proxy for determining whether a feasible rule placement exists. That is, when the total available rule capacity to path P_i is large enough compared to $\|Q_i\|$, a feasible solution is likely to exist. Therefore, we estimate the total rule capacity demand of path P_i based on $\|Q_i\|$.

Linear Program for Resource Allocation Given the estimation results, we need to decide how to split the rule capacity of each switch to satisfy the demand of all paths. The resource allocation decision can be formulated as a linear programming problem.

We introduce some notations here. Switch s_i can store c_i rules except rules for routing. Let m_j be the estimated total rule capacity required by path P_j. We define $\{h_{i,j}\}_{i \le n, j \le \ell}$ as indicator variables so that $h_{i,j} = 1$ if and only if s_i is on the path P_j. The variables are $\{x_{i,j}\}_{i \le n, j \le \ell}$, where $x_{i,j}$ represents the portion of rules at s_i that shall be allocated to P_j. For example, when $c_4 = 1000$ and $x_{4,3} = 0.4$, we need to allocate $1000 \times 0.4 = 400$ rules at s_4 for the path P_3. The linear program (Figure 3) has two constraints:

The capacity constraint (C1): Each switch does not assign more rules than it has to the paths.

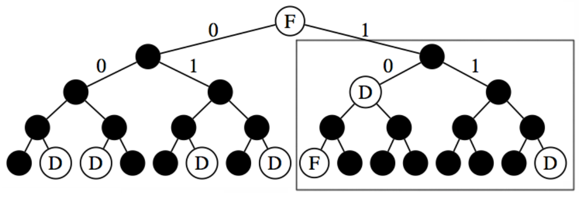
Path constraints (C2): Each path P_j has at least m_j pieces of rules it can use.

But it does not have any specific objective function, since we are happy with any assignment that satisfies all the constraints.(This is still equivalent to standard linear programs; see [23].)

In summary, in the resource allocation stage, the algorithm starts with estimating the total rule capacity required by each path. Given the estimation, it runs LP to give a resource allocation that balance the demand of all paths.

r_1 : (0001 : Drop)	r_4 : (0111 : Drop)	r_7 : (10∗ : Drop)
r_2 : (0010 : Drop)	r_5 : (1000 : Fwd)	r_8 : (∗ : Fwd)
r_3 : (0101 : Drop)	r_6 : (1111 : Drop)	

(a) Prioritized list of rules in a switch



(b) Tree representation of an access-control policy

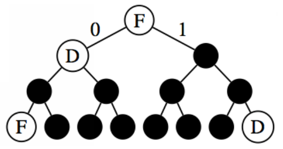
Figure 4: Example one-dimensional policy on a switch

3.3 Placing Rules Along a Path

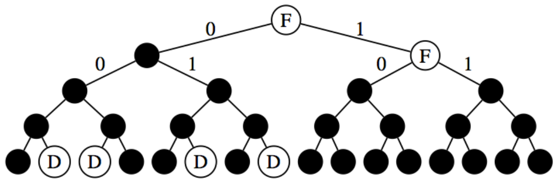
In this section, we introduce how to place rules along a path. Our goal is to install as few rules as possible to realize the policies, while staying within the capacity allocated by all switches. We start with one-dimensional (endpoint) policies (since routing policy is trivial), which depends on exactly one header field. We present an algorithm that *covers* the rules and *packs* groups of rules into switches along the path. The algorithm's performance is provably near-optimal. For multi-dimensional policies, optimizing the rules is thought to be NP-hard, even for a *single* switch [24]. We generalize our "cover-and-pack" approach to design a heuristic that is computationally efficient and offers good performance. For simplicity, we assume flow space associated with the path is the *full space*. Generalization is discussed at the end of the section.

One-Dimensional Policies on a Path We consider one-dimensional policies Q —those that match on a single header field, such as the IP destination prefix. For one-dimensional policies, there are known algorithms [24] for computing an optimal list of rules for a *single* switch. The function Q can be represented as a prefix-tree on the header field, as shown for an example access-control policy in Figure 4(a). Each internal node corresponds to a prefix, and each leaf node corresponds to a fully-specified header value; each prefix can be interpreted as a predicate. A node can specify an action (*e.g.,*forward or drop) for that prefix or descendants that do not fall under a more-specific prefix that includes an action. Computing the prioritized list of rules involves associating each labeled node with a predicate (*i.e.,*the prefix) and an action (*i.e.,*the node label), and assigning priority based on prefix length (with longer prefixes having higher priority), as shown in Figure 4(b).

If a single switch cannot store all eight rules in Figure 4, we must divide the rules across multiple switches. Our algorithm recursively *covers* a portion of the tree, *packs* the resulting rules into a switch, and *replaces* the subtree with a single node, as shown in Figure 5. In the "pack" phase, we select a subtree that can "fit" in



(a) Pack operation



(b) Replace operation (without reoptimization)

Figure 5: Pack-and-replace for one-dimensional policies

one switch, as shown in Figure 4(b) (the subtree inside the rectangle). If the root of this subtree has no action, the root inherits the action of its lowest ancestor—in this case, the root of the entire tree (see Figure 5(a)). The resulting rules under this subtree are then reoptimized, if possible, before assignment to the switch. In this example, switch 1 would have a prioritized list of four rules—(1000, Fwd), (1111, Drop), (10*, Drop) and (1*, Fwd). More generally, we may pack multiple subtrees into a single switch.[2] To ensure packets in this subtree are handled correctly at downstream switches, we replace the subtree with a single predicate at the root of the subtree with forward action (*e.g.,*a single "F" node in Figure 5(b)). Then, we can recursively apply the same pack-and-replace operations on the new tree to generate the rules for the second switch.

Our algorithm is *greedy*. Specifically, we pack the largest subtree subject to the switch's capacity constraints and we pack as many subtrees as possible in a single switch before proceeding to the next switch.

The algorithm has provable performance bounds on the rule space. Namely, we derive an *upper* bound on the total number of installed rules (C) by running our algorithm, arguing that $C \le (1+\epsilon)\|Q\|$, where $\epsilon \to 0$ as $\|Q\|$ tends to ∞.

Two-Dimensional Policies on a Path We next focus on the case of two-dimensional policies that match on the source and destination IP prefix. We visualize the endpoint policy Q by a two-dimensional space where each rule is represented by a rectangle and higher priority rectangles lie on top of lower priority rectangles, as shown in Figure 6.[3]

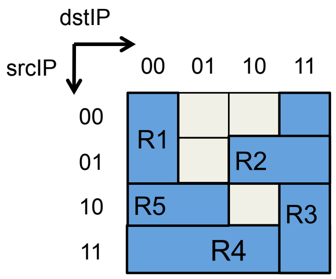
We extend our "Pack-and-Replace" approach to the two-dimensional case. A multi-dimensional predicate can lie completely inside another (*e.g.,* $R2$ and $R3$ lie

[2]For packets that do not fall into any packed subtrees, the default rules in switch will forward them to the downstream switches. See Section 3.1.

[3]Rule r_6 is intentionally omitted in the figure, since it covers the whole rectangle

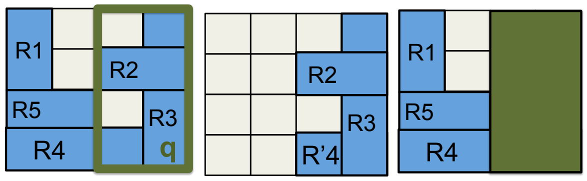
r_1 : (src_ip = 0∗, dst_ip = 00∗ : Fwd)
r_2 : (src_ip = 01∗, dst_ip = 1∗ : Fwd)
r_3 : (src_ip = ∗, dst_ip = 11∗ : Drop)
r_4 : (src_ip = 11∗, dst_ip = ∗ : Fwd)
r_5 : (src_ip = 10∗, dst_ip = 0∗ : Fwd)
r_6 : (src_ip = ∗, dst_ip = ∗ : Drop)

(a) Prioritized rule list of an acess-control policy



(b) Rectangular representation of the policy

Figure 6: An example two-dimensional policy



(a) Cover q (b) Switch s_1 (c) After s_1

Figure 7: Processing 2-dim endpoint policy Q

completely inside q), or partially overlap (*e.g.,* $R4$ and $R6$ partially overlap with q). If our algorithm selects q as a cover, we must include all of these rules when packing the projection of q (see Figure 7(b)). That is, Q_q is expressed as follows:

$r_1 : (q \wedge R2.p, R2.a)$	$r_2 : (q \wedge R3.p, R3.a)$
$r_3 : (q \wedge R4.p, R4.a)$	$r_4 : (q \wedge R6.p, R6.a)$

After packing the projection Q_q in a switch, we append a rule (q, Fwd) with the highest priority to Q to avoid re-processing packets matching q. But we can only remove those rules that are completely contained inside q—a rule that partially overlaps with q cannot be removed. The new residual network-wide policy is shown in Figure 7(c) and the corresponding rule list for the second switch is:

$r_1 : (q, \text{Fwd})$	$r_2 : (R1.p, R1.a)$
$r_3 : (R3.p, R3.a)$	$r_4 : (R5.p, R5.a)$

We end this section by highlighting a number of extensions to our algorithm.

Smaller flow space: When the flow space D_i for path P_i is not the entire flow space, we can still use the algorithm by requiring the rectangular covers chosen for path P_i must lie completely inside D_i.

Higher dimensions: Our algorithm continues to work when Q is a d-dimensional function for $d \geq 3$ by either cutting the along the two dimensions of source and destination IP prefix with all rules projected to rectangles, or using "hypercube" predicates instead of rectangular and cutting on all dimensions.

4. EVALUATION

Our evaluation of rule space optimization focuses on understanding the *overhead* of our algorithms, defined as $\frac{(C-\|Q\|)}{\|Q\|}$ where C is the minimum total rule capacity needed by running the algorithm and $\|Q\|$ is the size of the endpoint policy. For example, imagine we want to implement a network-wide policy of size m using two switches. If the overhead of our algorithm is 4%, which means we need additional $0.04m$ rules on the two switches in total.

Place One-Dimensional Policy on a Path We use real data from Route Views project [25]. The endpoint policy contains 407,534 rules. The results show that the overheads grow linearly in the length of the path, but even when there are 16 switches in the path, the actual overhead is less than 0.03%.

Place Multi-Dimensional Policy on a Path We use real firewall configurations from a large university network and synthetic data set from ClassBench [26]. Most of the data have 5000 to 14,000 rules. We see a similar trend with the one-dimensional testcases—the overhead grows steadily as the number of switches in the chain grows. Given an endpoint policy with 14,000 rules, our algorithm can distribute it over 8 switches, using at most 1,900 rules per switch. In the real data set, the average overhead for the 8-hop case is approximately 10% to 15% and the worst case overhead is around 25%.

Rule Placement over the Graph We use GT-ITM [27] to generate topologies containing 100 switches. We still use the real and synthetic data as the endpoint policies. The routing policy is shortest-path routing: the resulting average path length is 7. Due to the space limits, we only show the running time of our heuristics. There are two components in our algorithm: solving a linear program and running two-dimensional heuristics for all paths. In the real data set, LP takes at most 0.5s to complete, while heuristics takes between 0.5s to 9s depending on data sets.

5. CONCLUSION

Our rule-placement algorithms help raise the level of abstraction for creating SDN applications by shielding programmers from the details of distributing rules across switches. Our algorithm performs well on real and synthetic workloads, and has reasonable running time. In our ongoing work, we are exploring ways to parallelize our algorithm. For example, we can solve multiple instances of the path problem in parallel. This optimization should allow our algorithm to handle even larger networks and policies efficiently.

6. REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[2] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *OSDI*, Oct 2010.

[3] S. Shenker, "The future of networking and the past of protocols," Oct 2011. Invited talk at Open Networking Summit.

[4] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *NSDI*, Apr 2010.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *Trans. on Networking*, vol. 17, Aug 2009.

[6] N. Handigol, M. Flajslik, S. Seetharaman, R. Johari, and N. McKeown, "Aster*x: Load-balancing as a network primitive," in *Workshop on Architectural Concerns in Large Datacenters*, Jun 2010.

[7] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Hot-ICE*, Mar 2011.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM CCR*, vol. 38, no. 3, 2008.

[9] "Beacon: A Java-based OpenFlow control platform." `http://www.beaconcontroller.net`.

[10] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable OpenFlow control," Tech. Rep. TR10-08, Rice University, Dec 2010.

[11] A. Voellmy and P. Hudak, "Nettle: Functional reactive programming of OpenFlow networks," in *PADL*, Jan 2011.

[12] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?," in *OSDI*, Oct 2010.

[13] "POX." `http://www.noxrepo.org/pox/about-pox/`.

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

[15] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control in enterprise networks," in *WREN*, Aug 2009.

[16] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *PRESTO*, ACM, 2010.

[17] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *NSDI*, Apr 2013.

[18] N. Kang, J. Reich, J. Rexford, and D. Walker, "Policy transformation in software defined networks," in *poster abstract, ACM SIGCOMM*, Aug 2012.

[19] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *ACM-SIAM SODA*, pp. 1066–1075, 2007.

[20] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 490–500, Apr 2010.

[21] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "VCRIB: Virtualized rule management in the cloud," in *NSDI*, Apr 2013.

[22] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE Infocom Mini-conference*, Apr 2013.

[23] C. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*. Dover books on mathematics, Dover Publications, 1998.

[24] S. Suri, T. Sandholm, and P. Warkhede, "Compressing two-dimensional routing tables," *Algorithmica*, vol. 35, no. 4, pp. 287–300, 2003.

[25] "Route views project." `http://www.routeviews.org`.

[26] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE INFOCOM*, 2004.

[27] K. Calvert, M. B. Doar, A. Nexion, and E. W. Zegura, "Modeling Internet topology," *IEEE Communications Magazine*, vol. 35, pp. 160–163, 1997.