An Efficient Distributed Implementation of One Big Switch

Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker
Computer Science Department, Princeton University

1 Introduction

Software Defined Networking (SDN) enables flexible network policies by allowing controller applications to install packet-handling rules across a distributed collection of switches. However, first-generation controller platforms like NOX, POX and Beacon, 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 limits on each switch. More specifically, many SDN applications require rules that match on multiple 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. 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 forcing users to reason about low-level details, the One Big Switch abstraction [1, 2, 3, 4] allows SDN application programmers to define high-level policies and have the controller platform manage the placement of rules on physical switches. The abstraction partitions high-level policies into two distinct pieces, as shown in Figure 1:

- **A network-wide policy** Q specifies end-to-end policy, like access control and load balancing, while viewing the whole network as a single virtual switch. The policy is often represented as a prioritized list of rules.
- **A routing policy** R decides on what paths traffic should follow through the network based on a variety of metrics: latency, congestion, bandwidth and etc.

Given these two 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. We seek ways to utilize the capacity of all switches. 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.

In this talk, we describe a new family of efficient rule-placement algorithms that support implementation of the One Big Switch abstraction. In particular, we show how to process policy Q and R and generate an optimized set of rules for deployment in an OpenFlow network. We also demonstrate how such optimized rule sets may be efficiently incrementally updated as either the global end-to-end policy changes or the underlying routes are modified based on changing network conditions. This talk will not assume background in advanced algorithm designs.

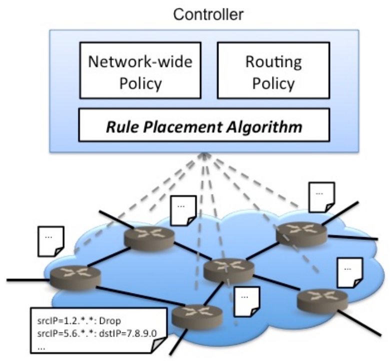


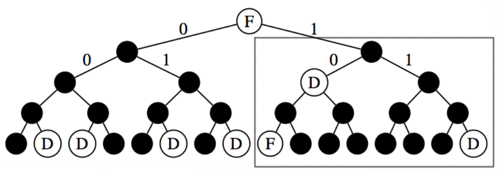
Figure 1: High-level policy and low-level rule placement

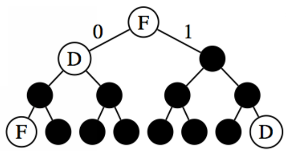
2 Rule placement algorithm

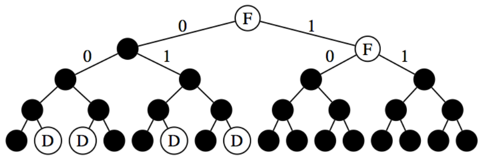
We designed a family of efficient rule-placement algorithms of increasing complexity, starting with the simple case of policies matching one packet-header field along a chain, and building up to the general case which involves arbitrary routing, multi-dimensional patterns and support to enable dynamic, incremental update of policies. We next walk through these algorithms.

(1). Single dimensional chains. We first develop an approximation algorithm for distributing network-wide policies depending on a single header field (one-dimensinoal policy) over a chain of switches. In this case, traffic flows in one direction through a sequence of switches. Thus, the routing policy is trivial—all packets arriving at the first switch in the chain are routed to the last switch unless the network policy requires to drop the packets. We represent the network policy as a prefix-tree on the header field. An example is shown in Figure 2(a) and (d). If a single switch cannot store all eight rules in Figure 2(d), we must divide the rules across multiple switches. Our algorithm recursively *covers* a subtree, *packs* the resulting rules into a switch, and *replaces* the subtree with a single node. In the "pack" phase, we select the *largest* subtree that can "fit" in one switch, as shown in Figure 2(a) (the subtree inside the rectangle). If the root of this subtree has no

This work was supported in part by the NSF under grant CNS-1111698 and ONR under award N00014-12-1-0757, and by a Google Research Award.


(a) Tree representation


(b) Pack operation


(c) Replace operation (without reoptimization)

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

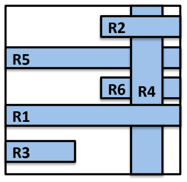
(d) Prioritized rule list

Figure 2: Distribute 1-dim network-wide policy over a chain

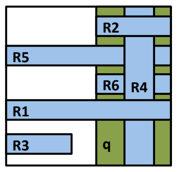
action, the root inherits the action of its lowest ancestor—in this case, the root of the entire tree (see Figure 2(b)). The resulting rules are then reoptimized, if possible, before assignment to the switch. We also need to install a *default* rule that forwards all unmatched packets to the next hop. More generally, we always greedily pack as many subtrees as possible into a single switch before proceeding to the next one. To ensure packets in the selected subtrees are handled correctly at downstream switches, we replace the subtree with a single predicate at the root of the subtree (e.g., a single "F" node in Figure 2(c)). Then, we can recursively apply the same pack-and-replace operations on the new tree to generate the rules for the second switch. The process ends until network-wide policy is trivial—the prefix tree only contains a single root node, whose action is Fwd. We prove that this algorithm gives near-optimal rule placement solutions. We also analyze the complexity of the algorithm in terms of running time.

(2). Multi-dimensional chains. We next develop principled heuristics that extends (1) to cover the case of multi-dimensional policies. Instead of using a tree to represent Q, we now use 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 3(a). As in the one-dimensional case, we select a predicate and pack its "projection" into a switch, subject to the capacity constraint (see Fig. 3(b) for the predicate q we select and Fig. 3(c) for the projection). Notice that after packing the projection of q in a switch, 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 3(c).

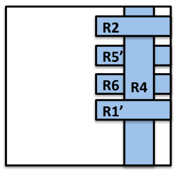
We also introduce a *cost-effectiveness* metric to guide us to select the most suitable covers in the "pack" phase. This is defined as the ratio between the total number of rules we need to install for a specific q and the total number of rules we can delete afterward. In the example of Figure 3, we install 5 rules for q (Fig 3(c)) and can delete 3 rules $R2$, $R4$, $R6$ afterward (Figure 3(d)). We keep using the greedy strategy for every switch, *i.e.,* we pack the most cost-effective cover in a switch until the switch is full and run the same algorithm for the next switch.



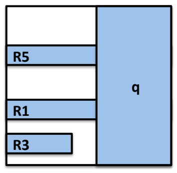
(a) Rules in 2-d



(b) Cover q



(c) Switch s_1



(d) After s_1

Figure 3: Processing 2-dim network-wide policy Q

(3). General graphs. Finally, we design an algorithm for handling multi-dimensional network-wide policies, arbitrary routing and general graphs. We use (2) as basic building block and linear programming techniques to solve our problem. Central to our algorithm is to *decompose* a graph problem into a collection of *paths.* A path refers to a chain of switches and a packet domain, which traverse this chain under the policies. After decomposition, the rule placement problem for general graphs is divided into two sub-problems:

1. How can we allocate the rule space to paths, especially when multiple paths share the same physical switch?
2. How can we place rules on each path when the rule space allocation scheme is given?

For the first subproblem, we formulate the "demand" of paths and "supply" of switches into a *linear programming* problem; For the second subproblem, we reuse (2) to give a collection of rule placement solutions. We show that this algorithm performs well with real policies and synthetic topologies.

References

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

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

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

[4] Nicira, "Networking in the era of virtualization." `http://nicira.com/sites/default/files/docs/Networking%20in%20the%20Era%20of%20Virtualization.pdf`, 2012.