Incremental Update for a Compositional SDN Hypervisor

Xin Jin
Princeton University
xinjin@cs.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

To realize the vision of SDN—an "app store" for network-management services—we need a way to compose applications developed for different controller platforms. For instance, an enterprise may want to combine a firewall written on OpenDaylight with a load balancer on Ryu and a monitoring application on Floodlight. To make this vision a reality, we propose a new kind of hypervisor that allows multiple applications to *collaborate* in processing the same traffic. Inspired by past work on Frenetic, our hypervisor supports a flexible configuration language that can combine packet-processing rules from different applications using sequential and parallel composition. A major challenge is efficiently combining updates to each prioritized list of OpenFlow rules, based on the hypervisor policy. Our key insight is that rule priorities form a convenient algebra that allows the hypervisor to compute the correct relative priorities of new rules *incrementally*, without shifting or rewriting the priorities of existing rules. We prove the correctness of our algorithms and show experimentally that these techniques can reduce computational overhead by 4X and the number of rule updates by 5X, compared to existing techniques.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Network virtualization; software-defined networking; hypervisor; composition; network update

HotSDN'14, August 22, 2014, Chicago, Illinois, USA.

http://dx.doi.org/10.1145/2620728.2620731.

1. INTRODUCTION

In its full glory, Software-Defined Networking (SDN) should allow network administrators to combine "best of breed" controller applications from different software developers. For example, a single network might run routing, access-control, monitoring, and load-balancing applications written by different programmers, using different programming languages and controller platforms. Each application can use OpenFlow [1] as a *lingua franca* for expressing its part of the network policy, and rely on an underlying *hypervisor* to mediate access to the switches. However, existing SDN hypervisors like FlowVisor [2] and OpenVirteX [3] split the traffic into disjoint "slices," each managed by a single application. While effective for allowing multiple tenants to share a network, slicing does not enable multiple applications to work together to process the *same* traffic.

Instead, future SDN hypervisors should go beyond slicing to support flexible policies for composing multiple applications, as shown in Figure 1. A network administrator should be able to select multiple application components (e.g., load balancing, routing, and monitoring). Each component generates a member policy represented as a prioritized list of OpenFlow rules for each switch, and adapts its policy over time in response to network events. Lying between the member components and the underlying switches, the hypervisor merges these rules as directed by a *composition configuration* to produce a single list of rules for each switch.

Inspired by past work in the Frenetic project [4, 5, 6, 7], we envision that the hypervisor can combine member policies in parallel (+) or in series (≫). For example, the hypervisor in Figure 1 applies the following composition configuration:

```
load_balancing >> (routing + monitoring)
```

Packets are processed first by the load-balancer policy (e.g., rewriting the destination IP address as part of selecting a backend server), and then simultaneously forwarded by the routing policy (e.g., mapping each destination IP prefix to an output port) and counted by the monitoring policy (e.g., counting traffic by source IP prefix). Based on the composition configuration, the hypervisor generates the rules for each switch, and updates these rules whenever any member policy changes.

The compositional hypervisor brings the benefits of composition *without requiring programmers to adopt the Frenetic programming environment.* Frenetic is based on a declarative (functional) API, where a function receives network events as inputs and generates a global network policy as an output. The Frenetic run-time system then compiles

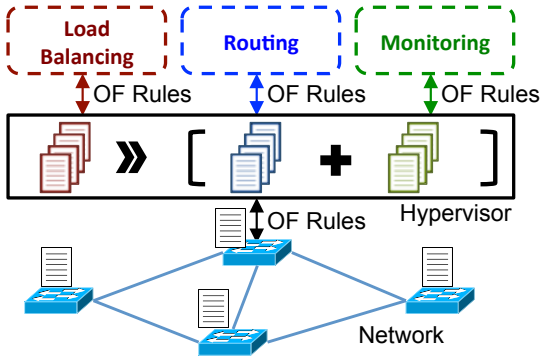


Figure 1: A compositional SDN hypervisor

the generated policy into OpenFlow rules that are sent to switches. In contrast, the hypervisor API is *imperative* to more closely match most other controller platforms such as Floodlight [8], NOX [9], POX [10], Ryu [11], and OpenDaylight [12]. When using an imperative interface, each component issues a stream of OpenFlow rules ("flowmods"). That is, components written in an imperative language can populate prioritized lists of OpenFlow rules, and have the hypervisor apply the functional composition operators on their behalf.

Representing member policies as prioritized lists of OpenFlow rules, rather than total functions, enables a surprising optimization. To operate efficiently, the hypervisor should not recompute and reinstall the switch-level rules from scratch every time a member policy changes. Instead, we need an *incremental* algorithm for computing small changes to the existing rules. Our key insight is that the hypervisor can perform simple arithmetic on the rule priorities in the member policies to compute the priorities for the switch-level rules. Parallel composition amounts to "adding" the priorities of the corresponding rules in the member policies, and sequential composition amounts to "concatenating" the priorities. In addition to proving the correctness of our algorithm, we show experimentally that these techniques can reduce computation overhead by 4X and flowmods by 5X as compared to a strawman solution that computes the switch-level rules from scratch.

2. BACKGROUND: POLICY COMPOSITION

A policy is a prioritized list of rules with priority, pattern, and action list.[1] Applying a policy to a packet at a given location produces a set of packets (0, 1, or more) at new locations. In our system, each application component generates a member policy and the hypervisor combines these member policies to produce a composed policy. In this section, we review the parallel and sequential composition operators introduced in [4, 5, 6, 7]. We discuss policy updates in the next section.

2.1 Parallel Composition (+)

Parallel composition enables multiple policies to apply concurrently on separate logical copies of the same packet. Given two policies P and Q, for an arriving packet t, their parallel composition $P + Q$ produces the union of applying P and Q to t separately. For example, suppose we have a monitoring policy M, and a routing policy R, as shown in Figure 2. M counts packets according to source IP prefix. For instance, the first rule in the table has priority 1. It counts packets with source IP prefix 1.0.0.0/24. R forwards packets based on destination IP: packets with destination IP 2.0.0.1 are forwarded to port 1, packets with destination IP 2.0.0.2 are forwarded to port 2, and others are dropped. Their parallel composition $M + R$ acts as if we apply M and R concurrently. Specifically, it both forwards and counts packets with source IP prefix 1.0.0.0/24 and destination IP 2.0.0.1 or 2.0.0.2.

To calculate $P+Q$, we are essentially calculating the cross product of the two policies. We iterate over $(p_i, q_j) \in P \times Q$ where p_i and q_j are rules taken from P and Q by priority in decreasing order. If the intersection of their patterns is not empty, we produce a rule in $P + Q$ with the pattern being the intersection of their patterns and the action list that can produce the union of the packet sets by applying their action lists separately. All the calculated rules are assigned priority from top to down by decrement of 1. For example, to calculate $M + R$, we first try m_1 and r_1. The intersection of their patterns is $\{srcip = 1.0.0.0/24, dstip = 2.0.0.1\} \neq \emptyset$. So we generate the first rule in $M + R$ with action list $\{fwd(1), count\}$. Repeating this for all (m_i, r_j) gives the composed policy $M + R$ as in Figure 2.

2.2 Sequential Composition (≫)

Sequential composition enables multiple policies to apply one after another. Given two policies P and Q, for an arriving packet t, their sequential composition $P \gg Q$ produces packets that are equal to first applying P to t then applying Q to each packet in the resulting packet set. We use the sequential composition of a load-balancing policy L and a routing policy R to illustrate this, as shown in Figure 2. L matches on source IP and splits the traffic into two backend servers (2.0.0.1 and 2.0.0.2) with ratio 1:3 by prefix splitting on source IP address. R examines destination IP address and forwards packets to the corresponding backend servers. Their sequential composition $L \gg R$ acts as if we first apply L first then apply R. Specifically, for packets with source IP prefix 0.0.0.0/2 and destination IP 3.0.0.0, it rewrites destination IP to 2.0.0.1 and forwards to port 1. For other packets with destination IP 3.0.0.0, it rewrites destination IP to 2.0.0.2 and forwards to port 2. It drops all other packets.

Calculating $P \gg Q$ is similar to calculating $P + Q$. We also iterate over $(p_i, q_j) \in P \times Q$ where p_i and q_j are rules taken from P and Q by priority in decreasing order. But we don't inspect the intersection of their patterns. Instead, we inspect the intersection of q_j's pattern and the pattern computed by applying p_i's action list to p_i's pattern. A nonempty intersection means packets after being applied p_i would match q_j and we generate a rule in the composed policy merging these two rules together. Finally, all the calculated rules are assigned priority from top to down by decrement of 1. For example, to calculate $L \gg R$, we first try l_1 and r_1. By applying l_1's action list to l_1's pattern, we get pattern $\{srcip = 0.0.0.0/2, dstip = 2.0.0.1\}$. The intersection of this pattern and r_1's pattern is $\{srcip = 0.0.0.0/2, dstip = 2.0.0.1\} \neq \emptyset$. So we generate the first rule in $L \gg R$ with pattern $\{srcip = 0.0.0.0/2, dstip = 3.0.0.0\}$ and action list $\{dstip = 2.0.0.1, fwd(1)\}$. Repeating this for all (l_i, r_j) gives the composed policy $L \gg R$ in Figure 2.

[1] In this paper, we limit the scope to OpenFlow 1.0 rules [13].

Monitoring M	
1. srcip=1.0.0.0/24	→ count
0. *	→ drop

Routing R	
1. dstip=2.0.0.1	→ fwd(1)
1. dstip=2.0.0.2	→ fwd(2)
0. *	→ drop

Load-balancing L	
3. srcip=0.0.0.0/2,dstip=3.0.0.0	→ dstip=2.0.0.1
1. dstip=3.0.0.0	→ dstip=2.0.0.2
0. *	→ drop

Parallel composition: $M + R$	
5. srcip=1.0.0.0/24,dstip=2.0.0.1	→ fwd(1),count
4. srcip=1.0.0.0/24,dstip=2.0.0.2	→ fwd(2),count
3. srcip=1.0.0.0/24	→ count
2. dstip=2.0.0.1	→ fwd(1)
1. dstip=2.0.0.2	→ fwd(2)
0. *	→ drop

Sequential composition: $L \gg R$	
2. srcip=0.0.0.0/2,dstip=3.0.0.0	→ dstip=2.0.0.1,fwd(1)
1. dstip=3.0.0.0	→ dstip=2.0.0.2,fwd(2)
0. *	→ drop

Figure 2: Example of parallel and sequential composition, adapted from [6]

3. ARITHMETIC ON RULE PRIORITIES

Network management is a dynamic process: Each component regulated by the hypervisor dynamically updates its member policy by inserting, deleting, or modifying rules. Whenever there is a change in a member policy, the hypervisor must update the composed policy accordingly. A naive policy-update mechanism can incur large overhead even for a single rule change in one member policy. In this section, we first introduce a strawman solution that has high update overhead. Then we describe a solution that enables efficient updates using a simple algebra of rule priorities.

Strawman solution: A straightforward solution to the policy update problem is to recalculate the entire composed policy whenever there is a change in a member policy. After obtaining the new composed policy, the hypervisor deletes all the existing rules on switches and installs new rules. This incurs tremendous switch overhead as even inserting one rule in one member policy would lead to deleting all old rules and installing the new rules. A quick fix is to calculate and install the difference between the old policy and the new one. However, simply calculating the difference without changing how we compute the composed policy has limited benefits. The main drawback is due to the priority assignment. Consider the table labelled "Routing R" in Figure 3 and assume we insert the bold rule that matches $dstip$ = 1.0.0.3 and has action $fwd(3)$. Most rules in the new composed policy (Table $M + R$ in Figure 3) have different priorities than in the original policy ($M + R$ in Figure 2) even if though they have the same pattern and action list. Thus the delta (rules in bold in $M + R$ in Figure 3) includes 7 out of the 8 rules in $M + R$. Consequently, we must delete the first 5 rules in the old $M + R$ policy and install the first 7 rules in the new $M + R$ policy. Only the last rule remains unchanged. Therefore, the strawman solution has both (1) high computation overhead as it has to recompute the entire composed policy and then decide the priority for each rule and (2) high rule-update overhead as it may have to make changes to rule priorities even if the rules it is generated from do not change.

The root cause of the inefficiency in the strawman solution is its priority assignment. It assigns priorities for rules in R based on the order they are computed. This unnecessarily ties the priority of each rule to the other rules. Additions or deletions of other rules may change the order of a rule in the composed policy, and thus change its priority even if the rules it is composed from do not change. Ideally, we would like to calculate rule priorities in a way that is purely based on the rules that the rule is composed from, so any changes in other rules do not affect this rule. Based on this idea, we give an algorithm for computing rule priorities. We also prove it is correct and show how to use it to handle different update operations.

3.1 Add for Parallel Composition

In a parallel composition $P + Q$, if rule r_k is generated from rule $p_i \in P$ and rule $q_j \in Q$, we assign $r_k.priority$ to be the sum of $p_i.priority$ and $q_j.priority$, i.e.,

$$r_k.priority = p_i.priority + q_j.priority. \quad (1)$$

To illustrate this concept, we show how to compute table $A = M + R$ in Figure 4. Here, rule a_1 in A is computed from m_1 and r_1 in Figure 2, so it is assigned priority $m_1.priority + r_1.priority = 2$. When we insert a new rule r_3 that matches $dstip$ = 1.0.0.3 in to R (the rule in bold in routing table R in Figure 3), we only need to compose this rule with the rules in M. This results in two rules as shown in bold in Figure 4: one is computed from r_3 and m_1, the other is computed from r_3 and m_2. We only need to compute and install these two rules on the switch for this update, because other rules do not have any change in their content or priorities.

One may wonder whether such a simple algorithm could introduce ambiguities or even faults into the composed policy. Consider the following example.

p_1.priority=12 q_2.priority=8 r_1.priority=12+8=20
p_2.priority=8 q_1.priority=12 r_2.priority=8+12=20

r_1 is composed from p_1 and q_2 and is assigned priority 20; r_2 is composed from p_2 and q_1 and is also assigned priority 20. One may wonder whether a packet can match both r_1 and r_2, and since they have the same priority, whether the composed policy would be ambiguous. This is not possible. Recall that the pattern of r_1 is the intersection of p_1 and q_2 and that the pattern of r_2 is the intersection of the patterns of p_2 and q_1. Consequently, if a packet can match r_1 and r_2 then it matches all of p_1, q_2, p_2, q_1. Hence the packet will also match a rule in R with priority 24, which is produced during the composition of rule p_1 and q_1. Formally, we can prove that as long as the member policies are not ambiguous[2], the composed policy is also not ambiguous and is correct, as stated in the following lemma.

LEMMA 1. *Let p_i be the highest priority rule in P that matches a packet t and q_j be the highest priority rule in Q that matches t. Let r_k be composed from p_i and q_j in $R = P + Q$ with priority calculated by Equation 1. Then r_k is the highest priority rule in R that matches t.*

[2]A policy is called ambiguous if there are two rules that can match the same packet and have the same priority.

Monitoring M	
1. srcip=1.0.0.0/24	→ count
0. *	→ drop

Routing R	
1. dstip=2.0.0.1	→ fwd(1)
1. dstip=2.0.0.2	→ fwd(2)
1. dstip=2.0.0.3	**→ fwd(3)**
0. *	→ drop

Load-balancing L	
3. srcip=0.0.0.0/2,dstip=3.0.0.0	→ dstip=2.0.0.1
2. srcip=0.0.0.0/1,dstip=3.0.0.0	**→ dstip=2.0.0.3**
1. dstip=3.0.0.0	→ dstip=2.0.0.2
0. *	→ drop

Parallel composition: $M + R$	
7. srcip=1.0.0.0/24,dstip=2.0.0.1	**→ fwd(1),count**
6. srcip=1.0.0.0/24,dstip=2.0.0.2	**→ fwd(2),count**
5. srcip=1.0.0.0/24,dstip=2.0.0.3	**→ fwd(3),count**
4. srcip=1.0.0.0/24	**→ count**
3. dstip=2.0.0.1	**→ fwd(1)**
2. dstip=2.0.0.2	**→ fwd(2)**
1. dstip=2.0.0.3	**→ fwd(3)**
0. *	→ drop

Sequential composition: $L \gg R$	
3. srcip=0.0.0.0/2,dstip=3.0.0.0	**→ dstip=2.0.0.1,fwd(1)**
2. srcip=0.0.0.0/1,dstip=3.0.0.0	**→ dstip=2.0.0.3,fwd(3)**
1. dstip=3.0.0.0	→ dstip=2.0.0.2,fwd(2)
0. *	→ drop

Figure 3: Example of updating policy compositions. Inserted rules are in bold. Strawman solution unnecessarily updates rules that only change priorities.

Parallel composition: $A = M + R$	
2. srcip=1.0.0.0/24,dstip=2.0.0.1	→ fwd(1),count
2. srcip=1.0.0.0/24,dstip=2.0.0.2	→ fwd(2),count
2. srcip=1.0.0.0/24,dstip=2.0.0.3	**→ fwd(3),count**
1. srcip=1.0.0.0/24	→ count
1. dstip=2.0.0.1	→ fwd(1)
1. dstip=2.0.0.2	→ fwd(2)
1. dstip=2.0.0.3	**→ fwd(3)**
0. *	→ drop

Sequential composition: $B = L \gg R$	
25. srcip=0.0.0.0/2,dstip=3.0.0.0	→ dstip=2.0.0.1,fwd(1)
17. srcip=0.0.0.0/1,dstip=3.0.0.0	**→ dstip=2.0.0.3,fwd(3)**
9. dstip=3.0.0.0	→ dstip=2.0.0.2,fwd(2)
0. *	→ drop

Figure 4: Example of incremental update. By carefully assigning priorities in composed policies, only rules that change their content are updated.

PROOF. We prove this by contradiction. Suppose there is a rule $r_{k'} \in R$ that matches t and has a higher priority than r_k, i.e.,

$$r_{k'}.priority > r_k.priority.$$

Let $r_{k'}$ be computed by $p_{i'} \in P$ and $q_{j'} \in Q$. Since $r_{k'}$ matches t, so $p_{i'}$ matches t and $q_{j'}$ matches t. We have the following two equations for their priorities.

$$r_k.priority = p_i.priority + q_j.priority,$$

$$r_{k'}.priority = p_{i'}.priority + q_{j'}.priority.$$

Therefore, we have

$$p_{i'}.priority + q_{j'}.priority > p_i.priority + q_j.priority$$

With this inequality, we can derive that either

$$p_{i'}.priority > p_i.priority \text{ or } q_{j'}.priority > q_j.priority.$$

But this contradicts the fact that p_i is the highest priority rule in P that matches t and q_j is the highest priority rule in Q that matches t. □

3.2 Concatenate for Sequential Composition

In a sequential composition $P \gg Q$, if r_k is composed from $p_i \in P$ and $q_j \in Q$, then we assign $r_k.priority$ to be the concatenation of $p_i.priority$ and $q_j.priority$, i.e.,

$$r_k.priority = p_i.priority \circ q_j.priority. \quad (2)$$

In the equation, "$\circ$" concatenates two priorities, where each priority is represented as a fixed-width bit string. Doing so enforces a *lexicographic ordering* on the pair of priorities. Specifically, let $a_1 = b_1 \circ c_1$ and $a_2 = b_2 \circ c_2$, then $a_1 > a_2$ only if ($b_1 > b_2$, or $b_1 = b_2$ and $c_1 > c_2$), and $a_1 = a_2$ only if $b_1 = b_2$ and $c_1 = c_2$.

In practice, we calculate "$\circ$" as follows. Let q_j be in the range $[0, MAX_Q)$ where $MAX_Q - 1$ is the highest priority that Q may use[3]. Then we calculate $r_k.priority$ with the following equation

$$r_k.priority = p_i.priority \cdot MAX_Q + q_j.priority. \quad (3)$$

It implements "$\circ$". We illustrate the mechanism with $B = L \gg R$. Let $MAX_R = 8$. We calculate $L \gg R$ as shown in Figure 4. l_1 and r_1 generates b_1 with priority $l_1.priority \times 8 + r_1.priority = 25$. Suppose we add a new rule in L that directs traffic with source IP prefix 0.0.0.0/1 to backend server with $dstip = 2.0.0.3$. We compose it with all rules in R in Figure 3. It generates the second rule b_2 in Figure 4 which is generated from from l_2 and r_3 with priority $l_2.priority \cdot 8 + r_3 = 17$. As in the parallel composition, the newly inserted rules do not affect the priorities of other rules. We only need to compute and update this rule. Similarly, we can prove that as long as the member policies are not ambiguous, the composed policy is also not ambiguous and is correct, as stated in the following lemma.

LEMMA 2. *Let p_i be the highest priority rule in P that matches a packet t and q_j be the highest priority rule in Q that matches the packet set after applying p_i to t. Let r_k be composed from p_i and q_j in $R = P \gg Q$ with priority calculated by Equation 2. Then r_k is the highest priority rule in R that matches t.*

PROOF. We prove this by contradiction. Suppose there is a rule $r_{k'} \in R$ that matches t and has a higher priority than r_k, i.e.,

$$r_{k'}.priority > r_k.priority$$

[3]In practice, because of the limited bits of priority, the hypervisor must limit the priorities each component may use.

Let $r_{k'}$ be constructed from $p_{i'} \in P$ and $q_{j'} \in Q$. Since $r_{k'}$ matches t, we know that $p_{i'}$ matches t and $q_{j'}$ matches the packet set after applying $p_{i'}$ to t. We have the following two equations for their priorities.

$$r_k.priority = p_i.priority \circ q_j.priority,$$

$$r_{k'}.priority = p_{i'}.priority \circ q_{j'}.priority.$$

Therefore, we have

$$p_{i'}.priority \circ q_{j'}.priority > p_i.priority \circ q_j.priority$$

With this inequality, we can derive that either

$$p_{i'}.priority > p_i.proirity$$

or

$$p_{i'}.priority = p_i.priority \;\&\&\; q_{j'}.priority > q_j.priority.$$

In the former case, it contradicts the fact that p_i is the highest priority rule in P that matches t. In the latter case, $p_{i'}.priority = p_i.proirity$ implicates $p_{i'} = p_i$ because policies are unambiguous. Then $q_{j'}$ becomes the highest priority rule in Q that matches the packet set after applying p_i to t. This contradicts the fact that q_j be the highest priority rule in Q that matches the packet set after applying p_i to t. □

REMARK 1. *By lemma 1, lemma 2 and induction on the structure of policies, we can generalize our algorithm to compute compositions of more than two policies connected by parallel and sequential operators, e.g.,* $(P + Q) \gg (R + S)$.

REMARK 2. *The algorithm remains correct when member policies are ambiguous. More specifically, when member policies are ambiguous it is straightforward to show that a packet will match a rule generated from* one of *the rules with highest priority in each member policy.*

3.3 Processing Update Operations

Using the rule priority arithmetic described above, a hypervisor can handle a variety of different kinds of member policy updates. Let P and Q be two member policies and R be the composed policy. We handle rule updates in P or Q as follows.

Rule addition: When a rule p^* is inserted in to P (or q^* in to Q), the hypervisor will compose this rule with each rule in Q (or P). For each computed new rule in R, it is assigned priority according to Equation 1 or 2 depending on the composition operator. The newly generated rules are sent to switches. Existing rules are left untouched.

Rule deletion: When a rule p^* is deleted from P (or q^* from Q), the hypervisor locates all rules in R that are generated from this rule and deletes them from the switch. All other rules are left untouched.

Rule modification: When a rule p^* is modified in P (or q^* in Q), the hypervisor will compose this rule with each rule in Q (or P). For each (p^*, q_i) pair (or (p_i, q^*) pair), there are 4 possibilities. (1) It generates a rule for R before but does not generate a rule now (e.g., because of a pattern change): we delete the old rule. (2) It generates a rule for R before and generates a rule now: we modify the rule if it changes. (3) It does not generate a rule for R before but generate a rule now: we insert the new rule. (4) It does not generate a rule for R before and does not generate a rule now: no change is needed.

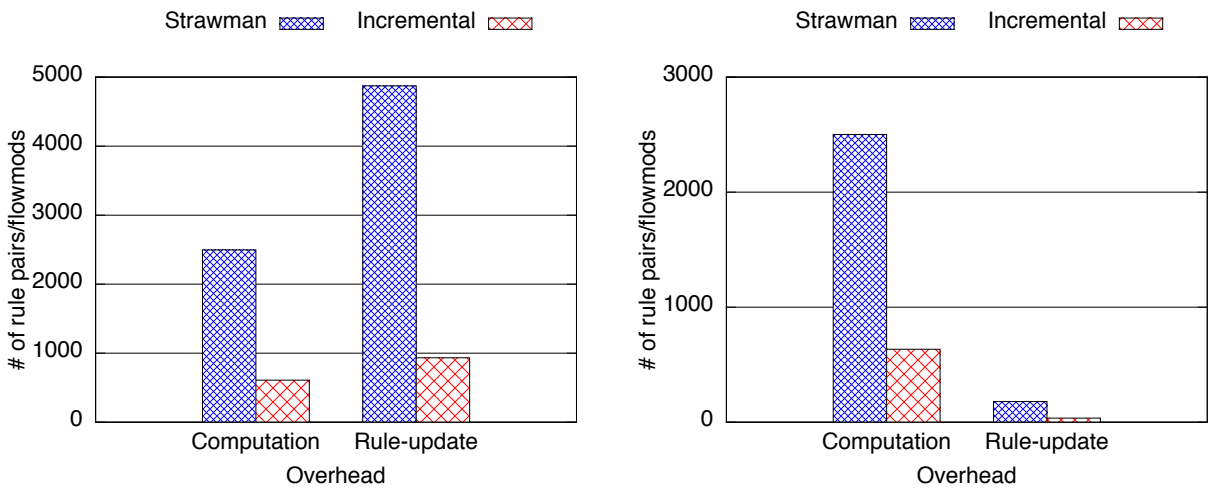
4. EVALUATION

In this section, we present a preliminary evaluation of our incremental update algorithm and compare it against the strawman solution. The experiments include policy updates on two representative composition examples: (1) a parallel composition of a monitoring policy and a routing policy, and (2) a sequential composition of a load balancing policy and a routing policy. We compare the following two metrics in the experiments.

- **Computation overhead:** The number of rule pairs examined by the algorithm while generating the new, composed policy.
- **Rule-update overhead:** The number of flowmods generated by the algorithm for the new, composed policy.

Parallel composition: In this experiment, we compose a monitoring policy M and a routing policy R in parallel as similar to Table 2. Initially, M and R have 50 rules each. They are randomly generated. Then we randomly generate 5 to 15 update operations for each of M and R. Each update operation either adds, deletes, or modifies a rule with the same probability. We repeat the update 100 times and show the average of computation overhead and rule-update overhead in Figure 5(a). We can easily see that the incremental update solution outperforms the strawman solution. It reduces the computation overhead by 4X and rule-update overhead by 5X. Moreover, we can see that the rule-update overhead is higher than the computation overhead. For the strawman solution, the reason is that it deletes almost all existing rules and then installs new rules. Thus, it almost doubles the overhead. For the incremental update solution, the reason is that it uses pointers to delete rules in the composed policy that correspond to deleted rules in member policies, and thus very little computation is incurred for deletion operations.

Sequential composition: In this experiment, we compose a load balancing policy L and a routing policy R sequentially, as in Table 2. Similarly to the previous experiment, initially we have 50 rules in each of L and R. They are randomly generated. Then we randomly generate 5 to 15 update operations for each of L and R. Each operation either adds, deletes or modifies a rule with the same probability. We repeat the update 100 times and present the average of computation overhead and rule-update overhead in Figure 5(b). Again our incremental update solution outperforms the strawman solution. It reduces the computation overhead by 4X and rule-update overhead by 5X. There are also two other interesting observations. First, the overheads are smaller than the ones in the parallel composition experiment. This is because in sequential composition many pairs of member rules do not generate a composed rule. Hence there are fewer total rules in the composed policy than in the parallel composition experiment. Second, the rule-update overhead is smaller than the computation overhead, which is different than in the parallel composition experiment. The reason is also similar to the first observation. When there is a member rule change, we have to compose this member rule with all member rules in the other member policy, but most of them do not generate a composed rule. Therefore, the rule-update overhead is smaller than the computation overhead.



(a) Parallel composition: monitoring + routing

(b) Sequential composition: load balancing ≫ routing

Figure 5: Comparison of strawman solution and incremental update solution in updating policy compositions

5. CONCLUSION

A compositional network hypervisor can simplify network management by allowing different applications written in different languages, or on different platforms, to work together to process the same traffic. A major challenge that arises when implementing such a hypervisor involves developing algorithms for correct, efficient, and real-time processing of rule updates from different applications. This paper presents a novel algorithm for such updates. Moreover, we analyze the correctness of our algorithm and show experimentally that it can significantly reduce update overhead as compared to the strawman solution. There are many interesting topics for future work, such as adding support for rule timeouts, adding support for OpenFlow 1.3 [14], and integrating the incremental update mechanism with consistent update mechanisms to enable efficient, network-wide updates that preserve consistency properties [15, 16, 17].

6. ACKNOWLEDGMENTS

We would like to thank Mojgan Ghasemi, Nanxi Kang, Naga Katta, Srinivas Narayana, Cole Schlesinger, and the anonymous HotSDN reviewers for their feedback on earlier versions of this paper. This work was supported by NSF grant TC-1111520 and DARPA grant MRC-007692-001.

7. 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, 2008.

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

[3] "OpenVirteX." `http://tools.onlab.us/ovx.html`.

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

[5] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM POPL*, 2012.

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

[7] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *ACM POPL*, 2014.

[8] "Floodlight OpenFlow Controller." `http://floodlight.openflowhub.org/`.

[9] "NOX OpenFlow Controller." `http://www.noxrepo.org/`.

[10] "POX OpenFlow Controller." `http://www.noxrepo.org/pox/about-pox/`.

[11] "Ryu OpenFlow Controller." `http://osrg.github.io/ryu/`.

[12] "OpenDaylight Platform." `http://www.opendaylight.org/`.

[13] "OpenFlow switch specification 1.0.0." `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`.

[14] "OpenFlow switch specification 1.3.0." `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`.

[15] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, 2012.

[16] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM SIGCOMM HotSDN Workshop*, August 2013.

[17] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *ACM SIGCOMM HotNets Workshop*, 2013.