CoVisor: A Compositional Hypervisor for Software-Defined Networks

Xin Jin, Jennifer Gossels, Jennifer Rexford, David Walker
Princeton University

Abstract

We present CoVisor, a new kind of network hypervisor that enables, in a single network, the deployment of multiple control applications written in different programming languages and operating on different controller platforms. Unlike past hypervisors, which focused on *slicing* the network into disjoint parts for separate control by separate entities, CoVisor allows multiple controllers to *cooperate* on managing the same shared traffic. Consequently, network administrators can use CoVisor to assemble a collection of independently-developed "best of breed" applications—a firewall, a load balancer, a gateway, a router, a traffic monitor—and can apply those applications in combination, or separately, to the desired traffic. CoVisor also abstracts concrete topologies, providing custom virtual topologies in their place, and allows administrators to specify access controls that regulate the packets a given controller may see, modify, monitor, or reroute. The central technical contribution of the work is a new set of efficient algorithms for composing controller policies, for compiling virtual networks into concrete OpenFlow rules, and for efficiently processing controller rule updates. We have built a CoVisor prototype, and shown that it is several orders of magnitude faster than a naive implementation.

1 Introduction

A foundational principle of Software-Defined Networking (SDN) is to decouple control logic from vendor-specific hardware. Such a separation allows network administrators to deploy both the software and the hardware most suited to their needs, rather than being forced to compromise on one or both fronts because of the lack of availability of the perfect box. To fully realize this vision of freely assembling "best of breed" solutions, administrators should be able to run any combination of controller applications on their networks. If the optimal monitoring application is written in Python on Ryu [1] and the best routing application is written in Java on Floodlight [2], the administrator should be able to deploy both of them in the network.

A network hypervisor is a natural solution to this problem of bringing together disparate controllers. However, existing hypervisors [3, 4] restrict each controller to a distinct *slice* of network traffic. While useful in scenarios like multi-tenancy in which each tenant controls its own traffic, they do not enable multiple applications to collaboratively process the same traffic. Thus, an SDN hypervisor must be capable of more than just slicing. More specifically, in this paper, we show how to bring together the following key hypervisor features and implement them *efficiently* in a single, coherent system.

(1) Assembly of multiple controllers. A network administrator should be able to assemble multiple controllers in a flexible and configurable manner. Inspired by network programming languages like Frenetic [5], we compose data plane policies in three ways: *in parallel* (allow multiple controllers to act independently on the same packets at the same time), *sequentially* (allow one controller to process certain traffic before another), and *by overriding* (allow one controller to choose to act or to defer control to another controller). However, unlike Frenetic and related systems, our hypervisor is independent of the specific languages, libraries, or controller platforms used to construct client applications. Instead, the hypervisor intercepts and processes industry-standard OpenFlow messages, assembling and transforming them to match administrator-specified composition policies. Doing so efficiently requires new incremental algorithms for processing rule updates.

(2) Definition of abstract topologies. To protect the physical infrastructure, an administrator should be able to limit what each controller can see of the physical topology. Our hypervisor supports this by allowing the administrator to provide a custom virtual topology to each controller, thereby facilitating reuse of (physical) topology-independent code. For example, to a firewall controller the administrator may abstract the network as a "big virtual switch"; the firewall does not need to know the underlying topology to determine if a packet should be forwarded or dropped. In contrast, a routing controller needs the exact topology to perform its task effectively. In addition, topology abstraction helps the administrator implement complex functionality in a modular manner. Some switches, such as a gateway between an Ethernet island and the IP core, may play multiple roles in the network. The hypervisor can create one virtual switch for each role, assign each to a controller application precisely tailored to its single task, and compile policies written for the virtual network into the physical network.

(3) Protection against misbehaving controllers. In ad-

dition to restricting what a controller can see of the physical topology, an administrator may also want to impose fine-grained control on how a controller can process packets. This access control is important to protect against buggy or maliciously misbehaving third-party controllers. For example, a firewall controller should not be allowed to modify packets, and a MAC learner should not be able to inspect IP or TCP headers. The hypervisor enforces these restrictions by limiting the functionality of the virtual switches exposed to each controller.

The primary technical challenge surrounding the creation of such a fully featured hypervisor is efficiency. The hypervisor must host tens of controllers, each of which installs tens of thousands of rules. Complicating matters further, these controllers are updating rules constantly, as dictated by their application logic (e.g., traffic engineering, failure recovery, and attack detection [6, 7, 8, 9, 10, 11, 12, 13, 14]). The naive hypervisor design is to recompile the composed policy from scratch for every rule update, and then install in each switch's flow table the difference between the existing and updated policies. This strawman solution is prohibitively expensive in terms of both the time to compile the new policy and the time to install new rules on switches.

In this paper we present CoVisor, a hypervisor that exploits efficient new algorithms to compile and update policies from upstream controllers, each of which has its own view of the network topology. Figure 1 illustrates the CoVisor architecture. CoVisor serves as a transparent layer between controllers and the physical network. Each of the five applications shown at the top of Figure 1 is an unmodified SDN program running on its own controller; each controller outputs OpenFlow rules for the virtual topology shown below it, without any knowledge that this virtual topology does not physically exist. CoVisor intercepts the OpenFlow rules output by all five controllers and compiles them into a single policy for the physical network via a two-phase process.

First, CoVisor uses a novel algorithm to incrementally compose applications in the manner specified by the administrator. The key insight is that rule priorities form a convenient algebra to calculate priorities for new rules, obviating the need to recompile from scratch for every rule update. Note that the problem of incremental update also exists in Frenetic and is not solved efficiently today [5]. Second, CoVisor translates the composed policy into rules for the physical topology. Specifically, we develop a new compilation algorithm for the case of one physical switch mapped to multiple virtual switches. Existing solutions handle this reactively by sending the first packet of every flow to the controller [15]. At both stages, CoVisor employs efficient data structures to further reduce compilation overhead by exploiting knowledge of the structure of policies provided by the access-control restrictions. After compiling the policy, CoVisor sends the necessary rule updates to switches.

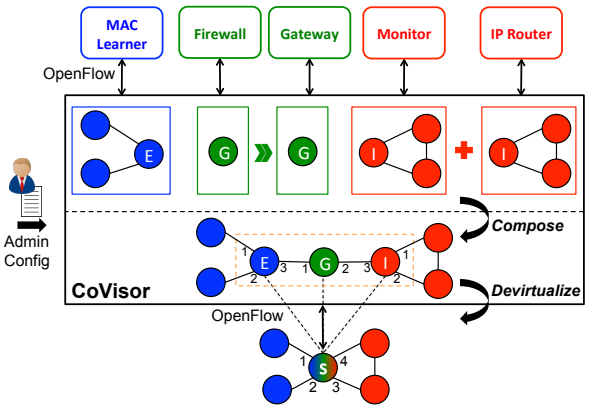


Figure 1: CoVisor overview.

At the far left of Figure 1, CoVisor takes configuration input from the administrator. These configuration responsibilities are threefold: (1) define how the policies of the controllers should be assembled; (2) create each controller's virtual network by specifying the components to be included and the physical-virtual mapping; and (3) state access control limitations for each controller.

In summary, we make the following contributions.

- We define the architecture of a new kind of compositional hypervisor, which allows applications written in different languages and on different controllers to process packets collaboratively.
- We develop a new algorithm to compile the parallel, sequential, and override operators introduced in earlier work [5, 16, 17] incrementally (§3).
- We develop a new, incremental algorithm to compile policies written for virtual topologies into rules for physical switches (§4).
- We employ customized data structures that leverage access-control restrictions, often a source of overhead, to further reduce compilation time (§5).

We describe our prototype in §6 and evaluation in §7. We review related work in §8 and conclude in §9.

2 CoVisor Overview

This section provides an overview of CoVisor. CoVisor's features fall into two categories: (i) those that combine applications running on multiple controllers to produce a single flow table for each physical switch (§2.1); and (ii) those that limit an individual controller's view of the topology and packet-processing capabilities (§2.2).

To implement these features, CoVisor relies on a two-phase compilation process. The first phase assembles the policies of individual controllers, written for a virtual network, into a composed policy for the virtual network. The second phase compiles this virtual policy into a pol-

icy for the physical network that realizes the intent expressed by the virtual policy. Algorithms for these phases are covered in §3 and §4, respectively.

2.1 Composition of Multiple Controllers

CoVisor allows network administrators to combine the packet-processing specifications of multiple controllers into a single specification for the physical network. We call these "packet-processing specifications" output by each controller *member policies* and the single specification a *composed policy*. In practice, the member policies are defined by OpenFlow commands issued from a controller to CoVisor. We use the terms *policy implementation* or just *implementation* to refer specifically to the list of OpenFlow rules used to express a policy.

The network administrator configures CoVisor to compose controllers with a simple language of commands. Let T range over policies defined in the command language. This language allows administrators to specify that some default action (a) should be applied to a set of packets, that a particular member policy (x) should be applied, that two separate policies should be applied in parallel ($T_1 + T_2$), that two separate policies should be applied in sequence ($T_1 \gg T_2$), or that one member policy should be applied, and if it fails to match a packet, some other policy should act as a default ($x \rhd T$). The following paragraphs explain these policies in greater detail.

Actions (a): The most basic composed policy is an atomic packet-processing action a. Such actions include any function from a packet to a set of packets implementable in OpenFlow, such as the actions to drop a packet ($drop$), to forward a packet out a particular port ($fwd(3)$), or to send a packet to the controller ($to\_controller(x)$).

Parallel operator ($+$): The parallel composition of two policies $T_1 + T_2$ operates by logically (though not necessarily physically) copying the packet, applying T_1 to one copy and T_2 to the other, and taking the union of the results. For example, let M be a monitoring policy and Q be a routing policy. If M counts packets based on source IP prefix and Q forwards packets based on destination IP prefix, $M + Q$ does both operations on all packets.

Sequential operator ($\gg$): The sequential operator enables two controllers to process traffic one after another. For example, let L be a load-balancing policy, and let Q be a routing policy. More specifically, for packets destined to anycast IP address 3.0.0.0, L rewrites the destination IP to a server replica's IP based on source IP prefix, and Q forwards packets based on destination IP prefix. To obtain the combined behavior of L and Q—to first rewrite the destination IP address and then forward the rewritten packet to the correct place—the network administrator uses the policy $L \gg Q$.

Command	Parameters
`createVSw`	pSw_1 <pSw_2, ..., pSw_n>
`createVPort`	vSw <pSw pPort>
`createVLink`	vSw_1 $vPort_1$ vSw_2 $vPort_2$
`connectHost`	vSw vPort host

Table 1: API to construct a virtual network. Brackets <> indicate optional arguments.

```
E = createVSw S              // vswitch E
G = createVSw S              // vswitch G
I = createVSw S              // vswitch I
E1 = createVPort E S 1       // port 1 on E
E2 = createVPort E S 2       // port 2 on E
E3 = createVPort E           // port 3 on E
G1 = createVPort G           // port 1 on G
L1 = createVLink E 3 G 1     // link E−G
...      remaining commands omitted for brevity.
```

Figure 2: Administrator configuration to create (a subset of) the physical-virtual mapping shown in Figure 1.

Override operator ($\rhd$): Each controller x provides CoVisor with a member policy specifying how x wants the network to process packets. The policy $x \rhd T$ attempts to apply x's member policy to any incoming packet t. If x's policy does not specify how to handle t, then T is used as a default. For example, suppose one controller x is running an elephant flow routing application and another controller y is running an infrastructure routing application. If we want x to override y for elephant flow packets, y to route all regular traffic, and any packet not covered by either policy to be dropped, we use the policy $x \rhd (y \rhd drop)$.

2.2 Constraints on Individual Controllers

In addition to composing member policies, CoVisor allows the administrator to virtualize the underlying topology and restrict the packet-processing capabilities available to each controller. This helps administrators hide infrastructure information from third-party controllers, reuse topology-independent algorithms, and provide security against malicious or buggy control software.

2.2.1 Constraints on Topology Visibility

Rather than exposing the full details of the physical topology to each controller, CoVisor provides each with its own virtual topology. Table 1 shows the API to construct a custom virtual network. `createVSw` creates a virtual switch. It can be used to create two kinds of physical-virtual mappings as follows. (1) *many-to-one* (many physical switches map to one virtual switch): call the function once with a list of physical switch identifiers; (2) *one-to-many* (a single physical switch

maps to many virtual switches): call the function multiple times with the same physical switch identifier. `createVPort` creates a virtual port. To map it to a physical port, the administrator includes the corresponding physical switch and port number. `createVLink` creates a virtual link by connecting two virtual ports. `connectHost` connects a host to a virtual port.

Example. Consider the example physical-virtual topology mapping shown in Figure 1. The physical topology represents an enterprise network consisting of an Ethernet island (shown in blue in Figure 1) connected by a gateway router (multicolored and labeled *S*) to the IP core (red). We abstract gateway switch *S* to three virtual switches: *E*, *G*, and *I*. Figure 2 shows how the administrator uses CoVisor's API to create the virtual mapping.

These four commands allow the administrator to create one level of virtual topology on top of a physical network. To create multiple levels of topology abstraction, the administrator can run one CoVisor instance on top of another. Supporting this behavior in a single instance of CoVisor is part of our future work.

2.2.2 Constraints on Packet Handling

CoVisor imposes fine-grained access control on how a controller can process packets by virtualizing switch functionality. The administrator sets custom capabilities on each controller's virtual switches, thereby choosing which functionalities of the physical network to expose on a controller-by-controller basis.

Pattern: The administrator specifies which header fields a controller can match and how each field can be matched (i.e., exact-match, prefix-match, or arbitrary wildcard-match). CoVisor currently supports the 12 fields in the OpenFlow 1.0 specification, with prefix-match an option only for source and destination IP addresses.

Action: The administrator specifies the actions a controller can perform on matched packets. CoVisor currently supports the actions in the OpenFlow 1.0 specification, including forward, drop (indicated by an empty action list), and modify (the administrator determines which fields can be modified). The administrator also controls whether a controller can query packets and counters from switches and send packets to switches.

Example. In the example in Figure 1, the administrator can restrict the MAC learner to match only on source and destination MAC and inport and the firewall to match only on the five tuple. Also, the administrator can disallow both applications from modifying packets.

2.3 Handling Failures

Controllers, switches, and CoVisor itself can fail during operation. We describe how CoVisor responds to them.

Controller failure: The administrator configures CoVisor with a default policy for each controller to execute in the event of controller failure. The default policy is application-dependent. For example, a logical default for a firewall controller is *drop* (erase all installed rules and install a rule that drops all packets), because a firewall should fail safe. In contrast, the default policy for a monitoring controller can be *id* (identical, i.e., leave all rules in the switch), as monitoring rules are not critical to the operation of a network, and the counters can be reused if the monitoring controller recovers.

Switch failure: If a switch fails, all its rules are removed and CoVisor notifies the relevant controllers. Moreover, in the case of many-to-one virtualization, CoVisor allows the virtual switch to remain functional by rerouting traffic around the failed physical switch (if possible in the physical network).

Hypervisor failure: We currently do not deal with hypervisor failure. Replication techniques in distributed systems may be applied to CoVisor, but a full exploration is beyond the scope of this work.

3 Incremental Policy Compilation

Network management is a dynamic process. Applications update their policies in response to various network events, like a change in the traffic matrix, switch and link failures, and detection of attacks [6, 7, 8, 9, 10, 11, 12, 13, 14]. Therefore, CoVisor receives streams of member policy updates from controllers and has to recompile and update the composed policy frequently. In this section, we first review policy compilation and introduce a strawman solution, and then we describe an efficient solution based on a convenient algebra on rule priorities.

3.1 Background on Policy Compilation

The first stage of policy compilation entails combining member policies into a single composed policy. Controllers implement member policies by sending OpenFlow rules to CoVisor. A rule r is a triple $r = (p;m;a)$ where p is a priority, m is a match pattern, and a is an action list. Given a rule $r = (p;m;a)$, we use the notation $r.priority$ to refer to p, $r.match$ to refer to m, and $r.action$ to refer to a. We denote the set of packets matching $r.match$ as $r.mSet$. Now we describe how to compile each composition operator outlined in §2.1. We assume all policy implementations include only OpenFlow 1.0 rules and that each switch has a single flow table.

Parallel operator (+): To compile $T_1 + T_2$, we first compile T_1 and T_2 into implementations R_1 and R_2. (In practice, each controller communicates its member policy to

Monitoring M_R
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(0;\ *;\ drop)$

Routing Q_R
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Load balancing L_R
$(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1)$
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2)$
$(0;\ *;\ drop)$

Elephant flow routing E_R
$(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$

Parallel composition: $comp_+(M_R, Q_R)$
$(5;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ count, fwd(1))$
$(4;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ count, fwd(2))$
$(3;\ srcip = 1.0.0.0/24;\ count)$
$(2;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Sequential composition: $comp_{\gg}(L_R, Q_R)$
$(2;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1))$
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2))$
$(0;\ *;\ drop)$

Override composition: $comp_{\triangleright}(E_R, Q_R)$
$(3;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$
$(2;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
$(0;\ *;\ drop)$

Figure 3: Example of policy compilation.

CoVisor in an already compiled form. We explicitly include this step because it represents the base case of the recursive process.) Then, we compute $comp_+(R_1, R_2)$ by iterating over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. We produce a rule r in the composed implementation if the intersection of $r_{1i}.mSet$ and $r_{2j}.mSet$ is not empty. *r.match* is the intersection of $r_{1i}.match$ and $r_{2j}.match$, and *r.actions* is the union of $r_{1i}.actions$ and $r_{2j}.actions$. We defer priority assignment to later discussion in this subsection. Consider the example of $comp_+(M_R, Q_R)$ in Figure 3. Let $M_R = m_1, \ldots, m_n$ and $Q_R = q_1, \ldots, q_k$. We begin by considering m_1 and q_1. Since $m_1.mSet \cap q_1.mSet \neq \emptyset$, we produce a first rule r_1 in $comp_+(M_R, Q_R)$ with match pattern $\{srcip = 1.0.0.0/24, dstip = 2.0.0.1\}$ and action list $\{count, fwd(1)\}$. Composing all (m_i, q_j) pairs gives the composed policy implementation $comp_+(M_R, Q_R)$ of the policy composition $M + Q$.

Sequential operator ($\gg$): To compile $T_1 \gg T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\gg}(R_1, R_2)$. As with $comp_+(R_1, R_2)$, we iterate over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. However, now we produce a rule r in the composed policy if the intersection of $r_{2j}.mSet$ and the set of packets produced by applying $r_{1i}.action$ to all packets in $r_{1i}.mSet$ is not empty. Consider the example of $comp_{\gg}(L_R, Q_R)$ in Figure 3. Again, we begin iterating over $(l_i, q_j) \in L_R \times Q_R$ pairs by considering l_1 and q_1. Applying $l_1.action$ to all packets in $l_1.mSet$ gives the set of packets matching pattern $\{srcip = 0.0.0.0/2, dstip = 2.0.0.1\}$. The intersection of this set and $q_1.mSet$ is not empty. Hence, we generate the first rule in the composed policy implementation with match pattern $\{srcip = 0.0.0.0/2, dstip = 3.0.0.0\}$ and action list $\{dstip = 2.0.0.1, fwd(1)\}$. Repeating this process for all (l_i, q_j) pairs yields $comp_{\gg}(L_R, Q_R)$, the implementation of $L \gg Q$.

Routing Q_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))
$(0;\ *;\ drop)$

Parallel composition: $comp_+(M_R, Q_R)$
(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))
$(0;\ *;\ drop)$

Figure 4: Example of updating policy composition. Strawman solution.

Override operator ($\triangleright$): To compile $T_1 \triangleright T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\triangleright}(R_1, R_2)$ by stacking R_1 on top of R_2 with higher priority. For example in Figure 3, to compile $comp_{\triangleright}(E_R, Q_R)$, we put E_R's rules above Q_R's rules. Thus, packets with source IP 1.0.0.0 and destination IP 2.0.0.1 will be forwarded to port 3, and other packets with destination IP 2.0.0.1 will be forwarded to port 1.

Priority assignment and policy update problem: Recall that a rule r is a triple $(r.priority; r.match; r.action)$. Thus far, we have explained how to generate a list of $(match; action)$ pairs, or *pseudo-rules*. Our list of

Monitoring M_R
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(0;\ *;\ drop)$

Routing Q_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_R
$(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1)$
(2; srcip=0.0.0.0/1,dstip=3.0.0.0; dstip=2.0.0.3)
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2)$
$(0;\ *;\ drop)$

Elephant flow routing E_R
$(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$

Parallel composition: $comp_+(M_R, Q_R)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ fwd(1), count)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ fwd(2), count)$
(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: $comp_{\gg}(L_R, Q_R)$
$(25;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1))$
(17; srcip=0.0.0.0/1,dstip=3.0.0.0; dstip=2.0.0.3,fwd(3))
$(9;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2))$
$(0;\ *;\ drop)$

Override composition: $comp_{\rhd}(E_R, Q_R)$
$(9;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(1; dstip=2.0.0.3; fwd(3))
$(0;\ *;\ drop)$

Figure 5: Example of incremental update.

pseudo-rules is prioritized in the sense that each pseudo-rule's position indicates its relative priority, but we have not addressed how to assign a particular priority value to each pseudo-rule. Priority assignment is important for minimizing the overhead of policy update. Ideally, a single rule addition in one member policy implementation should not require recomputing the entire composed policy from scratch, nor should it require clearing the physical switch's flow table and installing thousands of flow-mods. (A flowmod is an OpenFlow message to update a rule in a switch.) In concrete terms, the update problem involves minimizing the following two overheads:

- **Computation overhead:** The number of rule pairs over which the composition function *comp* iterates to recompile the composed policy.
- **Rule update overhead:** The number of flowmods needed to update a switch to the new policy.

Strawman solution: The strawman solution is to assign priorities to rules in the composed implementation from bottom to top starting from 0 by increment of 1. Then, it installs the difference between the old implementation and the new one. For example, the priorities of rules in Figure 3 are assigned in this way. This approach incurs high computation and rule update overhead, because it requires recompiling the whole policy to determine each rule's new relative position and updates rules that only change priorities. For example, when a new rule is inserted to Q_R (in bold in Figure 4), although only the third and the seventh rules in $comp_+(M_R, Q_R)$ are new, five rules change their priorities. We have to update these five existing rules as well as add two new rules. Rules in bold in Figure 4 count toward this rule update overhead.

3.2 Incremental Update

Ideally, the priority of rule r in the composed implementation is a function solely of the rules in the member implementations from which it is generated. In this way, any updates of other rules in member implementations will not affect r. We observe that rule priorities form a convenient algebra which allows us to achieve this goal.

Add for parallel composition: Let R be the composed implementation of $comp_+(R_1, R_2)$. If rule $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the sum of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority + r_{2j}.priority. \qquad (1)$$

We show the example of $comp_+(M_R, Q_R)$ in Figure 5. The first rule in $comp_+(M_R, Q_R)$ is composed from m_1 and q_1. Hence, its priority is $m_1.priority + q_1.priority = 2$. Suppose a new rule (in bold in Figure 5) is inserted to Q_R. We only need to iterate over rule pairs (m_i, q_3) for all $m_i \in M_R$, rather than iterate over all the rule pairs. This generates two new rules (in bold in Figure 5). All existing rules do not change.

Concatenate for sequential composition: Let R be the composed implementation of $comp_{\gg}(R_1, R_2)$. If $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the concatenation of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority \circ r_{2j}.priority. \qquad (2)$$

Symbol $\circ$ in Equation 2 represents the concatenation of two priorities, where each priority is represented as a fixed-width bit string. Concatenation 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 $a_1 > a_2$ if and only if $\big(b_1 > b_2$ or $(b_1 = b_2$ and $c_1 > c_2)\big)$, and $a_1 = a_2$ if and only if $(b_1 = b_2$ and $c_1 = c_2)$. In practice, concatenation is computed as follows. Let r_{2j} be in the range $[0, MAX_{R_2})$ where $MAX_{R_2} - 1$ is the highest priority that R_2 may use[1]. Then $r_k.priority$ is computed by

$$r_k.priority = r_{1i}.priority \times MAX_{R_2} + r_{2j}.priority.$$

We show the example of $comp_{\gg}(L_R, Q_R)$ in Figure 5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\gg}(L_R, Q_R)$ is composed from l_1 and q_1. Thus, its priority is $l_1.priority \times 8 + q_1.priority = 25$. Suppose a new rule is inserted to L_R $\big($in bold in Figure 5$\big)$. We only need to iterate over rule pairs (l_3, q_j) for all $q_j \in Q_R$. This generates a new rule with priority 17 $\big($in bold in Figure 5$\big)$. All existing rules do not change.

Stack for override composition: Let R be the composed implementation of $comp_{\rhd}(R_1, R_2)$, and let R_2's priority space be $[0, MAX_{R_2})$. To assign priorities in R, we increase the priorities of R_1's rules by MAX_{R_2} and keep the priorities of R_2's rules unchanged. This process essentially stacks R_1's priority space on top of R_2's priority space. Specifically, let $r_k \in R$. By definition of $comp_{\rhd}$, r_k is in either R_1 or R_2. Let $r_k.mPriority$ be r_k's priority in the member implementation from which it comes. We assign priority to r_k as follows.

$$r_k.priority = \begin{cases} r_k.mPriority + MAX_{R_2} & \text{if } r_k \in R_1 \\ r_k.mPriority, & \text{if } r_k \in R_2 \end{cases} \quad (3)$$

We show the example of $E_R \rhd Q_R$ in Figure 5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\rhd}(E_R, Q_R)$ is generated from e_1, so it is assigned priority $e_1.priority + 8 = 9$. The second rule in $comp_{\rhd}(E_R, Q_R)$ is generated from q_1, so it is assigned priority $q_1.priority = 1$. When a new rule q_3 that matches $dstip = 1.0.0.3$ is inserted to Q_R, we simply add a new rule with priority 1 (in bold in Figure 5) to $comp_{\rhd}(E_R, Q_R)$ without affecting existing rules.

Remark 1 *We show the proofs of correctness for parallel and sequential composition in [18]. A similar proof for override composition is straightforward. Also note that we may waste the priority space if we do not have an accurate estimation on MAX_{R_2} in sequential and override composition. This is not a weakness of the algorithm, but a weakness of using OpenFlow as the primary representation of the policy.*

With the algebra on rule priorities above, CoVisor processes the three kinds of rule updates as follows. Let R be the composed policy implementation of R_1 and R_2.

[1]CoVisor limits the priority space of each member policy, because the bits for priority in hardware are limited in practice.

Algorithm 1 Symbolic path generation

```
1: function GENPATHS(pkt)
2:     pkt.children ← {evaluate policy on pkt}
3:     for all child in pkt.children do
4:         if not child.REACHEDEGRESS() then
5:             GENPATHS (child)
```

Rule addition: When a new rule r_1^* is added to R_1 (or r_2^* to R_2), CoVisor composes this rule with each rule in R_2 (or R_1). It assigns priorities to new rules according to Equations 1, 2, and 3 and installs them to switches. All existing rules are untouched.

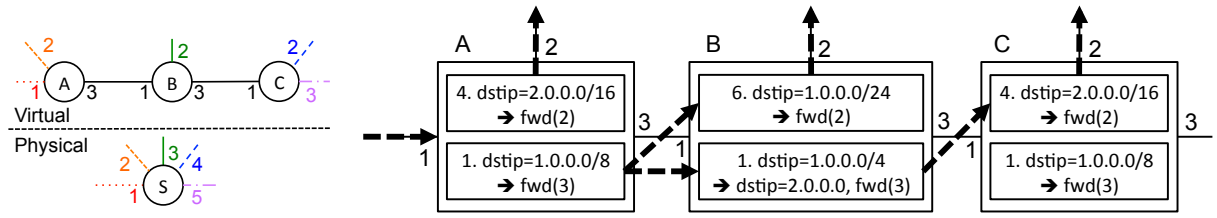
Rule deletion: When an old rule r_1^* is deleted from R_1 (or r_2^* from R_2), CoVisor finds all rules in R that are composed from this rule and deletes them from switches. All other rules are untouched.

Rule modification: Modifying a rule is equivalent to deleting an old rule and then inserting a new rule.

4 Compiling Topology Transformations

The first phase of compilation (§3) generates a composed policy for the virtual network. The second phase, which we describe in this section, compiles a policy for the virtual topology into one for the physical network. It comprises two sub-cases as described in §2.2.1: many-to-one and one-to-many. One-to-one is a degenerate case of these two. While previous work has explored compilation of the many-to-one case [19, 20], there does not exist any compilation algorithm for the one-to-many case. Pyretic [15] offers the one-to-many feature but implements it by sending the first packet of each flow to the controller and then installing micro-flow rules, a strategy which incurs prohibitive overhead. We present the first compilation algorithm for the one-to-many case.

Our algorithm is a novel combination of symbolic analysis [21] and incremental sequential composition. Intuitively, we inject a symbolic packet into the virtual network, follow all possible paths to egress ports, and sequentially compose the rules along each path. In this way, we derive rules for the physical switch to process traffic as intended by the controller's policy for the virtual network. To handle rule updates incrementally, we keep all the symbolic paths computed during this analysis and minimally modify them when the virtual policy changes. We divide our description into three parts: symbolic path generation (§4.1), sequential composition on symbolic paths (§4.2), and incremental update (§4.3).



(a) Topology. Colors and line types indicate physical-virtual mapping.

(b) Flow tables of virtual switches.

Figure 6: One-to-many virtualization.

4.1 Symbolic Path Generation

For each ingress port of the virtual network, we inject a single symbolic packet with wildcards in all fields (except *inport*). At every hop, we evaluate the policy on the packet, which generates zero, one, or more symbolic packets. We follow the generated symbolic packets until they all reach egress ports. Together, these symbolic paths form a tree rooted at the ingress port.

Algorithm 1 shows pseudocode for the path generation algorithm. In Line 2, we create all child packets that can result from evaluating the policy on `pkt`—one child packet for each rule r that `pkt` matches. As we construct the tree, we update `child`'s header, which denotes the subset of traffic represented by the symbolic packet, according to the information encoded in the rule responsible for generating `child` from `pkt`. By doing so, we avoid creating branches for paths that no packet could possibly follow.

We use the example in Figure 6 to illustrate the process. Figure 6(a) shows a physical-virtual topology mapping in which a physical switch S is virtualized to three virtual switches, A, B and C. The mapping between physical and virtual ports is color- and line type-coded. Figure 6(b) shows the policy of each virtual switch.

We inject a symbolic packet with header $*$, denoting wildcards in all fields, into port 1 of A. When we apply A's policy to this packet, we generate two child symbolic packets, p_1 and p_2. p_1 has destination IP 2.0.0.0/16, matches the first rule in A's policy, A_{R1}, and leaves the network at port 2 of A; p_2 has destination IP 1.0.0.0/8, matches A's second rule, A_{R2}, and reaches port 1 of B. We then evaluate B's policy on p_2, again generating two symbolic packets, p_{21} and p_{22}. p_{21} matches B_{R1} and leaves the network at port 2 of B; p_{22} matches B_{R2}, enters C at port 1, matches C_{R1}, and finally leaves the network at port 2 of C. In total, we get the following three symbolic paths: (1) $p_1 : A_{R1}$; (2) $p_{21} : A_{R2} \rightarrow B_{R1}$; and (3) $p_{22} : A_{R2} \rightarrow B_{R2} \rightarrow C_{R1}$.

4.2 Sequential Composition

For each symbolic path, we sequentially compose all the rules along its edges to generate a single rule. Then, we derive a final rule for the physical switch by adding a match on the *inport* value of the symbolic packet at the root of the tree. Returning to our example in Figure 6, the first symbolic path contains only A_{R1}. By adding port 1 to its match, we get the first rule for physical switch S.

$$S_{R1} = \big(4;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2)\big).$$

Adding $inport = 1$ is necessary because traffic that enters port 3 of C (port 5 of S) with destination IP 2.0.0.0/16 will be forwarded to port 2 of C (port 4 of S). Similarly, for the second and third symbolic paths, we evaluate $comp_{\gg}(A_{R2}, B_{R1})$ and $comp_{\gg}\big(comp_{\gg}(A_{R2}, B_{R2}), C_{R1}\big)$, respectively. We assume the priority space for each switch is $[0, 8)$. After adding ingress port, we obtain two more rules.

$$S_{R2} = \big(14; inport = 1, dstip = 1.0.0.0/24; fwd(3)\big)$$
$$S_{R3} = \big(76; inport = 1, dstip = 1.0.0.0/8;\ dstip = 2.0.0.0, fwd(4)\big)$$

Priority assignment: Because symbolic paths may have different lengths, for the devirtualization phase of compilation we need to augment the priority assignment algorithm for sequential composition presented in §3.2. For example, from sequential composition we get priorities 4, 14, and 76 for rules S_{R1}, S_{R2}, and S_{R3}, respectively. But, with these priorities, traffic entering port 1 at S with source IP 1.0.0.0/24 would match S_{R3} rather than S_{R2}, even though S_{R2} should have a higher priority than S_{R3}. This mismatch happens because S_{R2} is calculated from a path with only two hops (its priority is $1 \circ 6 = 14$) and S_{R3} is calculated from one with three hops ($1 \circ 1 \circ 4 = 76$). To address the mismatch, we set a hop length l^*. If a path is fewer than l^* hops, we pad 0s to the concatenation of the rule priorities. In practice, we use the number of switches in the virtual topology as l^*, as a path will have more than that number of hops only if the virtual policy contains a loop. This modified algorithm correctly orders S_{R2} and S_{R3}, assigning them respective priorities of $1 \circ 6 \circ 0 = 112$ and $4 \circ 0 \circ 0 = 256$. Figure 7 shows the rules for S with

Flow table of S
$(256;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2))$
$(112;\ inport = 1, dstip = 1.0.0.0/24;\ fwd(3))$
$(76;\ inport = 1, dstip = 1.0.0.0/8;\ dstip = 2.0.0.0, fwd(4))$

Figure 7: Flow table of switch S in Figure 6.

priorities calculated in this manner. We repeat the above procedure for all ingress ports of the virtual topology to get the final policy for S.

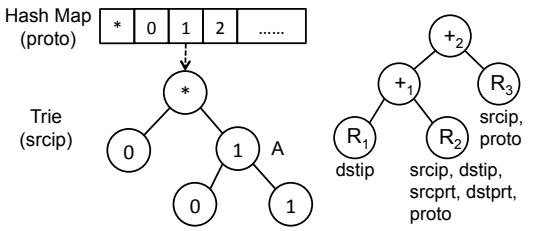
4.3 Incremental Update

By storing all the symbolic paths we generate when compiling a policy and partially modifying them upon a rule insertion or deletion, we can incrementally update a policy. This strategy obviates the need to compile the whole policy from scratch upon every rule update. In particular, when virtual switch V receives a rule update, we reevaluate V's policy on all symbolic packets that enter V. As a result, we may generate new symbolic packets, which we then follow until they reach egress ports. V's policy update may also modify the headers of or eliminate existing symbolic packets. Accordingly, we update the paths of modified symbolic packets and remove the paths of deleted packets. Then, we add and remove rules from the physical switch as described in §4.2. Our priority assignment algorithm ensures that these rule additions and deletions do not affect existing rules generated from symbolic paths that have not changed.

5 Exploiting Policy Structures

CoVisor imposes fine-grained access control on how each controller can match and modify packets. These restrictions both enhance security and provide hints that allow CoVisor to further optimize the compilation process. First, by knowing which fields individual policies match on and modify, we can build custom data structures to index rules, instead of resorting to general R-tree-based data structures for multi-dimensional classifiers as in [22, 23, 24]. Second, by correlating the matched or modified fields of two policies being composed, we can simplify their indexing data structures by only considering the fields they *both* care about.

We first describe the optimization problem, and then we show how to use the above two insights to solve it. For ease of explanation, we first assume that member policies are connected by the parallel operator. Later, we'll describe how to handle the sequential and override operators. Now suppose we have a parallel composition $T_1 + T_2$ with implementation $comp_+(R_1, R_2)$, and a new rule, r_1^*, is inserted into R_1. With our incremental update algorithm (§3.2), we need to iterate over all (r_1^*, r_{2j}) pairs



(a) Example rule index. (b) Example syntax tree.

Figure 8: Example of exploiting policy structures.

where $r_{2j} \in R_2$. The iteration processes $|R_2|$ pairs in total, where $|R_2|$ denotes the number of rules in R_2. However, if we know the structure of R_2, we can index its rules in a way that allows us to skip the rules that don't intersect with r_1^*, thereby further reducing computation overhead.

Index policies based on structure hints: Our goal is to reduce the number of rule pairs to iterate in compilation. A policy's structure indicates which fields should be indexed and how. For example, if R_2 is permitted only to do exact-match on destination MAC, then we can store its rules in a hash map keyed on destination MAC. If r_1^* also does exact-match on destination MAC, we simply use the destination MAC as key to search for rules in R_2's hash map. No rules in R_2 besides those stored under this key can intersect with r_1^*, because they differ on destination MAC. If r_1^* wildcards destination MAC, we return all rules in R_2, as they all intersect with r_1^*.

The preceding example is a simple case in which R_2 matches on one field. In general, a policy may match on multiple fields. We use single-field indexes (hash table for exact-match, trie for prefix-match, list for arbitrary wildcard-match) as building blocks to build a *multi-layer index* for multiple fields. Specifically, we first choose one field f_1 the policy can match and index the policy on this field. We store all rules with the same value in f_1 in the same bucket of the index. This forms the first layer of the index. Then we choose the second field f_2 and index rules in each f_1 bucket on f_2. We repeat this process for all the fields on which the policy can match. We choose the order of fields according to simple heuristics like preferring exact-match fields to prefix-match fields. In practice, a policy normally matches on a small number of fields, which means the number of layers is small.

Consider a policy that does exact-match on *proto* (protocol number) and prefix-match on *srcip*. We first index the policy based on *proto*. All rules with the same value in *proto* go to the same bucket, as shown in Figure 8(a). Note that the hash map contains a bucket keyed on $*$ for rules that do not match on *proto*. Then, we index all the rules that contain the same *proto* value on *srcip*. Because our example policy does prefix match on *srcip*, the second level of our multi-layer index comprises a trie for each bucket in the hash map. Figure 8(a) shows this sec-

ond level for rules with $proto = 1$; bucket A contains all the rules with $proto = 1$ and $srcip = 128.0.0.0/1$.

Correlate policy structures to reduce indexing fields: When composing policies, we can leverage the information we know about both to reduce the work we do to index each. Suppose R_1 matches on $dstip$ and R_2 matches on the five tuple ($srcip$, $dstip$, $srcport$, $dstport$, $proto$). Instead of storing R_2 in a five-layer index, we need only index the $dstip$. Because $dstip$ is the only field on which any rule r_1^* added to R_1 can match, r_1^* will intersect with a rule in R_2 as long as they intersect on $dstip$. Formally, let $R_i.fields$ be the set of fields on which R_i matches and $R_i.index$ be the set of fields R_i indexes. Given R_i and R_j in a composition, we have

$$R_i.index = R_j.index = R_i.fields \cap R_j.fields. \quad (4)$$

Back to our example, we have $R_1.index = R_2.index = R_1.fields \cap R_2.fields = \{dstip\}$.

A policy R_i itself may be composed from other policies R_j and R_k. Unlike in the previous example, we do not *a priori* know $R_i.fields$ and instead rely on the observation that a rule in a composed policy can match on a field f if and only if at least one of its component member policies can match on f. Hence, we get

$$R_i.fields = R_j.fields \cup R_k.fields. \quad (5)$$

Let's look at an example $(R_1 + R_2) + R_3$, which we show as a syntax tree in Figure 8(b). Initially, we know the match fields only for the leaf nodes. Then we calculate the match fields for node $+_1$ with $R_1.fields \cup R_2.fields = \{srcip, dstip, srcprt, dstprt, proto\}$. Then, we use Equation (4) to index $+_1$ and R_3 with $+_1.fields \cap R_3.fields = \{srcip, proto\}$.

Sequential and override composition: Suppose we have sequential composition $T_1 \gg T_2$ with implementation $comp_{\gg}(R_1, R_2)$. Then $R_1.fields$ not only contains the fields R_1 matches but also the fields it modifies in its action set. This is because, for $r_1 \in R_1$ and $r_2 \in R_2$, the pair (r_1, r_2) generates a rule for the composed policy if the intersection of $r_2.mSet$ and the set of packets resulting from applying $r_1.action$ to $r_1.mSet$ is not empty. Similarly, when we index R_1, the key for any rule r_{1i} is the value resulting from applying $r_1.action$ to $r_1.match$. We do not need to index policies for override composition, since we directly stack their rules.

6 Implementation

We implemented a prototype of CoVisor with 4000+ lines of Java code added to and modifying OpenVirteX [3]. We replaced the core logic of OpenVirteX, which isolates multiple controllers, with our composition and incremental update logic (§3). To OpenVirteX's built-in many-to-one virtualization, we added support for the one-to-many abstraction and our proactive compilation algorithm (§4). We further optimized compilation by exploiting the structure of policies as described in §5. We used `HashMap` in the Java standard library [25] to index rules with exact-match fields and `RadixTree` in the Concurrent-trees library [26] to index rules with prefix-match fields. Given a key (e.g., 1.0.0.0/16), `RadixTree` in the Concurrent-trees library only returns values for keys starting with this key (e.g., 1.0.0.0/24 and 1.0.0.0/30). We modified it to also give values for keys included by this key (e.g., 1.0.0.0/8). CoVisor currently supports the OpenFlow flowmod message; other commands, such as barrier messages and querying counters, will be supported in later versions.

7 Evaluation

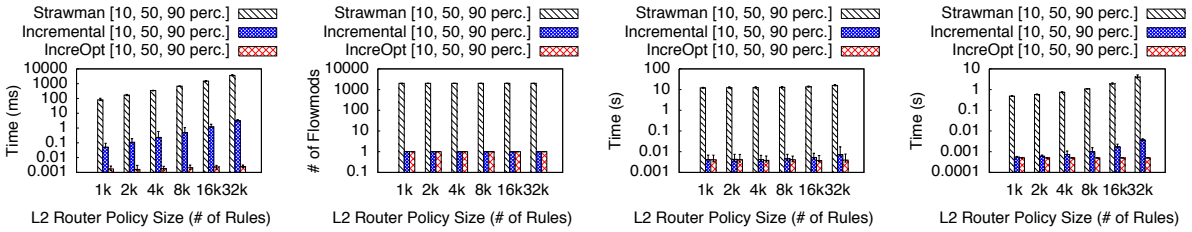
7.1 Methodology

Experiment Setup: We evaluate CoVisor under three scenarios, the first two of which evaluate composition efficiency and the third of which evaluates devirtualization efficiency. In each scenario, we stress CoVisor with a wide range of policy sizes. Since compiling policies to individual physical switches is independent in these scenarios, we show the results for a single physical switch. We run CoVisor on Mininet [27] and use Floodlight controllers [2]. The server is equipped with an Intel XEON W5580 processor with 8 cores and 6GB RAM. We describe each scenario in more detail below.

- **L2 Monitor + L2 Router:** L2 Monitor counts packets for source MAC and destination MAC pairs; L2 Router forwards packets based on destination MAC. The MAC addresses are randomly generated.
- **L3-L4 Firewall ≫ L3 Router:** L3-L4 Firewall filters packets based on the five tuple; L3 Router forwards packets according to destination IP prefix. The firewall policy is generated from ClassBench [28], a tool for benchmarking firewalls. The L3 router policy is generated with IP prefixes extracted from the firewall policy.
- **Gateway virtualization:** This is the topology virtualization discussed in §2.2.1. A switch that connects an Ethernet island to the IP core is abstracted to three virtual switches, which operate as a MAC learner, gateway, and IP router.

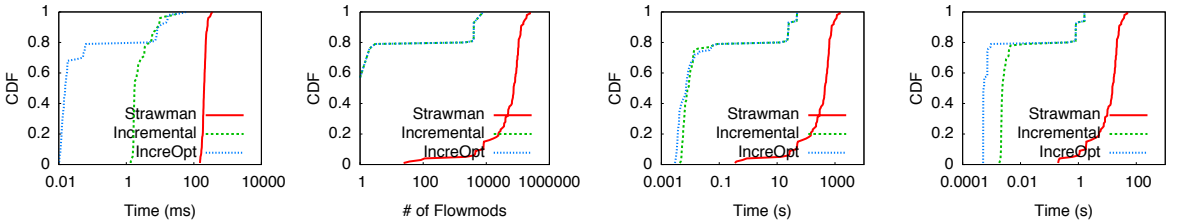
Metrics: We use the following metrics to measure efficiency. The thick bars in Figures 9 and 11 indicate the median, and the error bars show the 10th and 90th percentiles.

- **Compilation time:** The time to compile the policy composition or topology devirtualization.
- **Rule update overhead:** The number of flowmods to update the switch to the new flow table.



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 9: Per-rule update overhead of L2 Monitor + L2 Router (log-log scale).



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 10: Per-rule update overhead of L3-L4 Firewall ≫ L3 Router (log-log scale).

- **Total update time:** The sum of compilation time, rule update time, and additional system overhead like OpenFlow message (un)marshalling. Since hardware switches and software switches takes very different time in rule updates, we show both of them. As the software switches in Mininet do not mimic the rule update latency of hardware switches and do not give accurate timing on the actual rule installation in software switches, we use the rule update latency in [29] for hardware switches and that in [30] for software switches when calculating rule update times.

Comparison: We compare the following approaches.

- **Strawman:** Recompile the new policy from scratch for every policy update.
- **Incremental:** Incrementally compile the new policy using our algebra of rule priorities for policy composition (§3) and keeping symbolic path information for topology devirtualization (§4).
- **IncreOpt:** Further optimize Incremental by exploiting the structures of policies (§5).

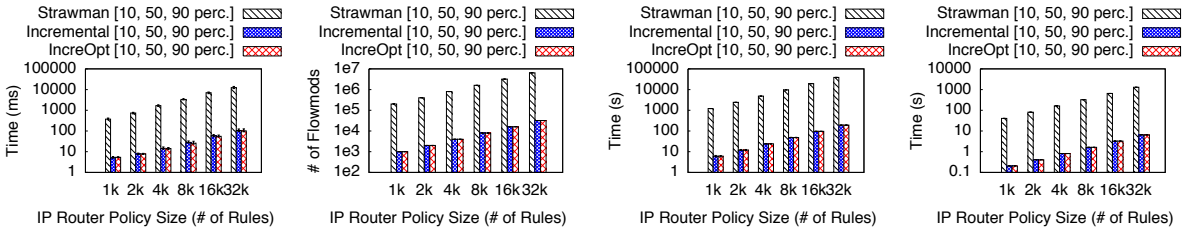
7.2 Composition Efficiency

Figure 9 shows the result of L2 Monitor + L2 Router. In this experiment, we initialize the L2 Monitor policy with 1000 rules, and then add 10 rules to measure the overhead for each. We repeat this process 10 times. We vary the size N of L2 Router policy from 1000 to 32,000 to show how overhead increases with larger policies. Figure 9(a) shows the compilation time. As expected, the compilation time of Strawman and Incremental increases with the policy size, because larger policies force our algorithm to consider more rule pairs. Since Strawman recompiles the whole policy, it is by far the slowest. On the other hand, IncreOpt has almost constant compilation time, because it indexes L2 Router's rules in a hash table keyed on destination MAC. When a rule is inserted to L2 Monitor's policy, the algorithm simply uses the rule's destination MAC to look up rules in the hash table.

Figure 9(b) shows the rule update overhead in terms of number of rules (same for hardware and software switches). Because of its naive priority assignment scheme, Strawman unnecessarily changes priorities of many existing rules and thus generates more flowmods than Incremental and IncreOpt. Incremental and IncreOpt generate the same policy, and therefore they have the same rule update overhead. We also observe that the rule update overhead does not increase with the size of L2 Router's policy. This is because the size of L2 Monitor's policy is fixed, and each monitor rule only intersects with one rule in L2 Router, since they both do exact-match on destination MAC.

Finally, Figures 9(c) and 9(d) show the total time. Notably, Incremental and IncreOpt are significantly faster than Strawman, and the gap between Incremental and IncreOpt is larger when using software switches. This is because software switches update rules faster than hardware switches, and therefore the compilation time accounts for a larger fraction of the total time for software switches.

Figure 10 shows the result of L3-L4 Firewall ≫ L3 Router. As before, we initialize L3-L4 Firewall's policy with 1000 rules and add 10 rules. Since the trend is similar to Figure 9 when we vary the size N of L3 Router, we instead show the CDF when L3 Router policy has 8,000



(a) Compilation Time (b) Rule Update Overhead (c) Total Update Time (Hardware) (d) Total Update Time (Software)

Figure 11: The switch connecting an Ethernet island to the IP core is virtualized to switches that operate as MAC learner, gateway, and IP router. Figures show the overhead of adding a host to the Ethernet island (log-log scale).

rules. Figure 10(a) shows the compilation time. Again, Strawman is several orders of magnitude slower than Incremental and IncreOpt. However, unlike in our previous experiment, we see a stepwise behavior of Incremental, and the difference between Incremental and IncreOpt also disappears after 80th percentile. This is an artifact of the content of L3-L4 Firewall from ClassBench. The firewall policy comprises approximately 80% rules matching on very specific destination IP prefix (/31, /32) and around 20% rules matching very general destination IP prefix (/1, /0). A firewall rule with a very specific destination IP prefix only composes with a few router rules, in which case IncreOpt processes fewer rule pairs in compilation than Incremental. On the other hand, a firewall rule with a very general destination IP prefix like /1 or /0 composes with half or all rules in the router policy, in which case Incremental and IncreOpt process a similar number of rule pairs and have similar compilation time. This reasoning also explains the shape of Incremental and IncreOpt in Figures 10(b), 10(c) and 10(d).

7.3 Devirtualization Efficiency

We use the gateway scenario to evaluate the efficiency of the devirtualization phase of compilation. In this experiment, we have 100 hosts in the Ethernet island. The MAC learner installs forwarding rules for connections between host pairs. To the Ethernet island, switch G simply appears as another host; hosts use G's MAC as destination MAC when they want to reach hosts across the IP core. We initialize the MAC learner policy with 1000 rules in switch E. Then, we add a new host to the Ethernet island. When the new host tries to talk to another host across the IP core, the MAC learner adds two rules to establish a bidirectional connection between the host and switch G. To compile this update, we compose the two new rules with the existing rules in switches G and I. The gateway policy at G is simply a MAC-rewriting repeater and ARP server. The IP router forwards packets based on destination IP prefix. We vary the size of the IP router policy at I from 1000 to 32,000 to evaluate how the overhead increases with larger policies.

Figure 11 shows the overhead. Strawman exhibits a long compilation time, as it has to recompile the policy from scratch. Strawman also generates more flowmods than necessary, because its priority assignment scheme may change the priorities of existing rules. In contrast, Incremental and IncreOpt incur significantly less overhead, because they keep all the symbolic paths and only need to change a few upon receiving the new rules. Finally, we notice that Incremental and IncreOpt do not show much difference in this experiment and the absolute values of total update time are high. This is because the MAC learner policy in switch E and the IP router policy in switch I match on different fields. Thus, when we do sequential composition on virtual paths, Incremental and IncreOpt iterate over a similar number of rule pairs and the result policy is almost a cross-product of the two policies at E and I. The cross-product is inevitable when compiling to a single flow table as the two policies match on different fields. Finally, we note that the multi-table support in OpenFlow 1.3 and newer hardware platforms like P4 [31] can make devirtualization more efficient. If multiple tables in a switch can be configured to in a pipeline to mirror the virtual network topology, then updating virtual switch tables can be directly mapped to updating physical tables. This can dramatically reduce compilation and rule update overhead. A complete exploration of this direction is part of our future work.

8 Related Work

Slicing: Existing network hypervisors mostly focus on slicing; they target multi-tenancy scenarios in which each tenant operates on a disjoint subset, or *slice*, of the traffic [3, 4, 32, 33]. In contrast, CoVisor allows multiple controllers to collaborate on processing the same traffic.

Topology abstraction: Many projects studied the many-to-one case [15, 19, 20, 34]. Pyretic [15] explored the one-to-many case, but its implementation reactively installs micro-flow rules. CoVisor provides the first proactive compilation algorithm by leveraging symbolic analysis to build symbolic paths [21] and applying incremental sequential composition to generate the rules.

Composition: The parallel and sequential operators are proposed in the Frenetic project [5, 15], and the override operator is described in [16, 17]. An incremental compilation algorithm for Frenetic policies is introduced in [17]. CoVisor is novel in using these operators to compose policies written on a variety of controller platforms, rather than just Frenetic. Furthermore, CoVisor takes advantage of the OpenFlow rules' explicit priorities; it uses a convenient algebra to calculate priorities for composed rules, thereby eliminating the need to build dependency graphs for rules and maintain scattered priority distributions [17]. Moreover, [17] only optimizes priority assignment for Frenetic policies; it is not a hypervisor to compose controllers, and does not have algorithms to compile topology virtualizations and optimizations by exploiting policy structures. Finally, it is an open problem to design a good interface for Frenetic to aid incremental update.

CoVisor is built upon our previous workshop paper [18], which presents the rule priority algebra for the parallel and sequential operators. This work includes new results on the algebra for the override operator, a proactive and incremental compilation algorithm for one-to-many virtualization, further optimization of compilation algorithms by exploiting policy structures, and a prototype implementation and evaluation.

Switch table type patterns: Table Type Patterns [35] and P4 [36] provide a syntax for describing flow table capabilities (e.g., fields that can be matched and modified). CoVisor uses this kind of information to build a customized data structure to optimize compilation. CoVisor's optimization technique differs from existing ways to index and accelerate multi-dimensional classifiers that don't know policy structures *a priori* [22, 23, 24, 37].

9 Conclusion

We present CoVisor, a compositional hypervisor that allows administrators to combine multiple controllers to collaboratively process a network's traffic. CoVisor uses a combination of novel algorithms and data structures to efficiently compile policies in an incremental manner. Evaluations on our prototype show that it is several orders of magnitude faster than a naive implementation, and we believe this is just the start of research on compositional hypervisors. There are many interesting future directions. In particular, extending existing and exploring new compilation techniques for multi-table support in compositional hypervisor setting is a very promising direction [31, 38]. First, this allows us to make efficient use of hardware capabilities and reduce the size of final policies for composition and devirtualization. Second, it introduces an incremental deployment path for hardware with OpenFlow 1.3 support as legacy applications written in OpenFlow 1.0 can run on top of CoVisor with CoVisor compiling them to OpenFlow 1.3.

References

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

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

[3] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make your virtual SDNs programmable," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[4] 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*, October 2010.

[5] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker, "Languages for software-defined networks," *IEEE Communications*, vol. 51, pp. 128–134, February 2013.

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM*, August 2013.

[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, August 2013.

[8] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *ACM SIGCOMM*, August 2013.

[9] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM*, August 2014.

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

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

[12] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *USENIX NSDI*, April 2010.

[13] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *ACM CoNEXT*, December 2011.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, August 2011.

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

[16] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined internet exchange," in *ACM SIGCOMM*, August 2014.

[17] X. Wen, L. Li, C. Diao, X. Zhao, and Y. Chen, "Compiling minimum incremental update for modular SDN languages," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[18] X. Jin, J. Rexford, and D. Walker, "Incremental update for a compositional SDN hypervisor," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[19] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *ACM CoNEXT*, December 2013.

[20] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM*, April 2013.

[21] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, April 2013.

[22] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects*, August 1999.

[23] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *ACM SIGCOMM*, August 2003.

[24] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *ACM SIGCOMM*, August 2010.

[25] "Java Standard Library." `http://docs.oracle.com/javase/7/docs/api/`.

[26] "Concurrent-trees Library." `https://code.google.com/p/concurrent-trees/`.

[27] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *ACM CoNEXT*, December 2012.

[28] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *IEEE INFOCOM*, March 2005.

[29] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *ACM SIGCOMM*, August 2014.

[30] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Passive and Active Measurement Conference*, March 2012.

[31] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX NSDI*, May 2015.

[32] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, *et al.*, "Network virtualization in multi-tenant datacenters," in *USENIX NSDI*, April 2014.

[33] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "VeRTIGO: Network virtualization and beyond," in *European Workshop on Software Defined Networks (EWSDN)*, October 2012.

[34] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Workshop on Programmable Routers for Extensible Services of Tomorrow*, November 2010.

[35] "OpenFlow Table Type Patterns 1.0." `http://tinyurl.com/oebjbsf`.

[36] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, vol. 44, pp. 87–95, July 2014.

[37] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," in *ACM SIGCOMM HotSDN Workshop*, August 2014.

[38] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From policies to pipelines," in *ACM ICFP*, September 2014.