A Fast Compiler for NetKAT

Steffen Smolka
Cornell University (USA)
smolka@cs.cornell.edu

Spiridon Eliopoulos *
Inhabited Type LLC (USA)
spiros@inhabitedtype.com

Nate Foster
Cornell University (USA)

Arjun Guha
UMass Amherst (USA)

Abstract

High-level programming languages play a key role in a growing number of networking platforms, streamlining application development and enabling precise formal reasoning about network behavior. Unfortunately, current compilers only handle "local" programs that specify behavior in terms of hop-by-hop forwarding behavior, or modest extensions such as simple paths. To encode richer "global" behaviors, programmers must add extra state—something that is tricky to get right and makes programs harder to write and maintain. Making matters worse, existing compilers can take tens of minutes to generate the forwarding state for the network, even on relatively small inputs. This forces programmers to waste time working around performance issues or even revert to using hardware-level APIs.

This paper presents a new compiler for the NetKAT language that handles rich features including regular paths and virtual networks, and yet is several orders of magnitude faster than previous compilers. The compiler uses symbolic automata to calculate the extra state needed to implement "global" programs, and an intermediate representation based on binary decision diagrams to dra-

matically improve performance. We describe the design and implementation of three essential compiler stages: from virtual programs (which specify behavior in terms of virtual topologies) to global programs (which specify network-wide behavior in terms of physical topologies), from global programs to local programs (which specify behavior in terms of single-switch behavior), and from local programs to hardware-level forwarding tables. We present results from experiments on real-world benchmarks that quantify performance in terms of compilation time and forwarding table size.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Software-defined networking, domain-specific languages, NetKAT, Frenetic, Kleene Algebra with tests, virtualization, binary decision diagrams.

1. Introduction

High-level languages are playing a key role in a growing number of networking platforms being developed in academia and industry. There are many examples: VMware uses `nlog`, a declarative language based on Datalog, to implement network virtualization [19]; SDX uses Pyretic to combine programs provided by different participants at Internet exchange points [13, 25]; PANE uses NetCore to allow end-hosts to participate in network management decisions [9, 24]; Flowlog offers tierless abstractions based on Datalog [26]; Maple allows packet-processing functions to be specified directly in Haskell or Java [33]; OpenDaylight's group-based policies describe the state of the network in terms of application-level connectivity requirements [29]; and ONOS provides an "intent framework" that encodes constraints on end-to-end paths [28].

* Work performed at Cornell University.

jnfoster@cs.cornell.edu arjun@cs.umass.edu

The details of these languages differ, but they all offer abstractions that enable thinking about the behavior of a network in terms of high-level constructs such as packet-processing functions rather than low-level switch configurations. To bridge the gap between these abstractions and the underlying hardware, the compilers for these languages map source programs into forwarding rules that can be installed in the hardware tables maintained by software-defined networking (SDN) switches.

Unfortunately, most compilers for SDN languages only handle "local" programs in which the intended behavior of the network is specified in terms of hop-by-hop processing on individual switches. A few support richer features such as end-to-end paths and network virtualization [19, 28, 33], but to the best of our knowledge, no prior work has presented a complete description of the algorithms one would use to generate the forwarding state needed to implement these features. For example, although NetKAT includes primitives that can be used to succinctly specify global behaviors including regular paths, the existing compiler only handles a local fragment [4]. This means that programmers can only use a restricted subset that is strictly less expressive than the full language and must manually manage the state needed to implement network-wide paths, virtual networks, and other similar features.

Another limitation of current compilers is that they are based on algorithms that perform poorly at scale. For example, the NetCore, NetKAT, PANE, and Pyretic compilers use a simple translation to forwarding tables, where primitive constructs are mapped directly

to small tables and other constructs are mapped to algebraic operators on forwarding tables. This approach quickly becomes impractical as the size of the generated tables can grow exponentially with the size of the program! This is a problem for platforms that rely on high-level languages to express control application logic, as a slow compiler can hinder the ability of the platform to effectively monitor and react to changing network state.

Indeed, to work around the performance issues in the current Pyretic compiler, the developers of SDX [13] extended the language in several ways, including adding a new low-cost composition operator that implements the disjoint union of packet-processing functions. The idea was that the implementation of the disjoint union operator could use a linear algorithm that simply concatenates the forwarding tables for each function rather than using the usual quadratic algorithm that does an all-pairs intersection between the entries in each table. However, even with this and other optimizations, the Pyretic compiler still took tens of minutes to generate the forwarding state for inputs of modest size.

Our approach. This paper presents a new compiler pipeline for NetKAT that handles local programs executing on a single switch, global programs that utilize the full expressive power of the language, and even programs written against virtual topologies. The algorithms that make up this pipeline are orders of magnitude faster than previous approaches—e.g., our system takes two seconds to compile the largest SDX benchmarks, versus several minutes in Pyretic, and other benchmarks demonstrate that our compiler is able to handle large inputs far beyond the scope of its competitors.

These results stem from a few key insights. First, to compile local programs, we exploit a novel intermediate representation based on binary decision diagrams (BDDs). This representation avoids the combinatorial explosion inherent in approaches based on forwarding tables and allows our compiler to leverage well-known techniques for representing and transforming BDDs. Second, to compile global programs, we use a generalization of symbolic automata [27] to handle the difficult task of generating the state needed to correctly implement features such as regular forwarding paths. Third, to compile virtual programs, we exploit the additional expressiveness provided by the global compiler to translate programs on a virtual topology into programs on the underlying physical topology.

We have built a full working implementation of our compiler in OCaml, and designed optimizations that reduce compilation time and the size of the generated forwarding tables. These optimizations are based on general insights related to BDDs (sharing common structures, rewriting naive recursive algorithms using dynamic programming, using heuristic field orderings, etc.) as well as domain-specific insights specific to SDN (algebraic optimization of NetKAT programs, per-switch specialization, etc.). To evaluate the performance of our compiler, we present results from experiments run on a variety of benchmarks. These experiments demonstrate that our compiler provides improved performance, scales to networks with tens of thousands of switches, and easily handles complex features such as virtualization.

Overall, this paper makes the following contributions:

- We present the first complete compiler pipeline for NetKAT that translates local, global, and virtual programs into forwarding

tables for SDN switches.

- We develop a generalization of BDDs and show how to implement a local SDN compiler using this data structure as an intermediate representation.
- We describe compilation algorithms for virtual and global programs based on graph algorithms and symbolic automata.
- We discuss an implementation in OCaml and develop optimizations that reduce running time and the size of the generated forwarding tables.
- We conduct experiments that show dramatic improvements over other compilers on a collection of benchmarks and case studies.

The next section briefly reviews the NetKAT language and discusses some challenges related to compiling SDN programs, to set the stage for the results described in the following sections.

2. Overview

NetKAT is a domain-specific language for specifying and reasoning about networks [4, 11]. It offers primitives for matching and modifying packet headers, as well combinators such as union and sequential composition that merge smaller programs into larger ones. NetKAT is based on a solid mathematical foundation, Kleene Algebra with Tests (KAT) [20], and comes equipped with an equational reasoning system that can be used to automatically verify many properties of programs [11].

NetKAT enables programmers to think in terms of functions on packets histories, where a packet (pk) is a record of fields and a his-

tory (h) is a non-empty list of packets. This is a dramatic departure from hardware-level APIs such as OpenFlow, which require thinking about low-level details such as forwarding table rules, matches, priorities, actions, timeouts, etc. NetKAT fields f include standard packet headers such as Ethernet source and destination addresses, VLAN tags, *etc*., as well as special fields to indicate the port (pt) and switch (sw) where the packet is located in the network. For

Syntax

$$
\begin{array}{rrll}
\text{Naturals} & n & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
\text{Fields} & f & ::= & f_1 \mid \cdots \mid f_k \\
\text{Packets} & pk & ::= & \{f_1 = n_1, \cdots, f_k = n_k\} \\
\text{Histories} & h & ::= & \langle pk \rangle \mid pk{::}h \\
\text{Predicates} & a, b & ::= & \mathit{true} && \textit{Identity} \\
 & & \mid & \mathit{false} && \textit{Drop} \\
 & & \mid & f{=}n && \textit{Test} \\
 & & \mid & a + b && \textit{Disjunction} \\
 & & \mid & a \cdot b && \textit{Conjunction} \\
 & & \mid & \neg a && \textit{Negation} \\
\text{Programs} & p, q & ::= & a && \textit{Filter} \\
 & & \mid & f{\leftarrow}n && \textit{Modification} \\
 & & \mid & p + q && \textit{Union} \\
 & & \mid & p \cdot q && \textit{Sequencing} \\
 & & \mid & p^* && \textit{Iteration} \\
 & & \mid & \mathsf{dup} && \textit{Duplication}
\end{array}
$$

Semantics

$$
\begin{aligned}
[\![p]\!] &\in \mathsf{History} \to \mathcal{P}(\mathsf{History}) \\
[\![\mathit{true}]\!]\ h &\triangleq \{h\} \\
[\![\mathit{false}]\!]\ h &\triangleq \{\} \\
[\![f{=}n]\!]\ (pk{::}h) &\triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h &\triangleq \{h\} \setminus ([\![a]\!]\ h)
\end{aligned}
$$

$$
\begin{aligned}
[\![f \leftarrow n]\!]\ (pk::h) &\triangleq \{pk[f := n]::h\} \\
[\![p + q]\!]\ h &\triangleq [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cdot q]\!]\ h &\triangleq ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p^*]\!]\ h &\triangleq \bigcup_i F^i\ h \\
\text{where } F^0\ h &\triangleq \{h\} \text{ and } F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h \\
[\![\mathsf{dup}]\!]\ (pk::h) &\triangleq \{pk::(pk::h)\}
\end{aligned}
$$

Figure 1: NetKAT syntax and semantics.

brevity, we use src and dst fields in examples, though our compiler implements all of the standard fields supported by OpenFlow [23].

Net*KAT *syntax and semantics. Formally, NetKAT is defined by the syntax and semantics given in Figure 1. Predicates a describe logical predicates on packets and include primitive tests $f{=}n$, which check whether field f is equal to n, as well as the standard collection of boolean operators. This paper focuses on tests that match fields exactly, although our implementation supports generalized tests, such as IP prefix matches. Programs p can be understood as packet-processing functions that consume a packet history and produce a set of packet histories. Filters a drop packets that do not satisfy a; modifications $f \leftarrow n$ update the f field to n; unions $p + q$ copy the input packet and process one copy using p, the other copy using q, and take the union of the results; sequences $p \cdot q$ process the input packet using p and then feed each output of p into q (the $\bullet$ operator is Kleisli composition); iterations p^* behave like

the union of p composed with itself zero or more times; and dups extend the trajectory recorded in the packet history by one hop.

Topology encoding. Readers who are familiar with Frenetic [10], Pyretic [25], or NetCore [24], will be familiar with the basic details of this functional packet-processing model. However, unlike these languages, NetKAT can also model the behavior of the entire network, including its topology. For example, a (unidirectional) link from port pt_1 on switch sw_1 to port pt_2 on switch sw_2, can be encoded in NetKAT as follows:

$$\mathsf{dup} \cdot \mathsf{sw}{=}sw_1 \cdot \mathsf{pt}{=}pt_1 \cdot \mathsf{sw}{\leftarrow}sw_2 \cdot \mathsf{pt}{\leftarrow}pt_2 \cdot \mathsf{dup}$$

Applying this pattern, the entire topology can be encoded as a union of links. Throughout this paper, we will use the shorthand $[sw_1{:}pt_1]{\rightarrow}[sw_2{:}pt_2]$ to indicate links, and assume that dup and modifications to the switch field occur only in links.

Local programs. Since NetKAT can encode both the network topology and the behavior of switches, a NetKAT program describes the end-to-end behavior of a network. One simple way to write NetKAT programs is to define predicates that describe where packets enter (in) and exit (out) the network, and interleave steps of processing on switches (p) and topology (t):

$$in \cdot (p \cdot t)^* \cdot p \cdot out$$

To execute the program, only p needs to be specified—the physical topology implements *in*, *t*, and *out*. Because no switch modifications or dups occur in p, it can be directly compiled to a collection of forwarding tables, one for each switch. Provided the physical topology is faithful to the encoding specified by *in*, t, and *out*, a

network of switches populated with these forwarding tables will behave like the above program. We call such a switch program p a *local* program because it describes the behavior of the network in terms of hop-by-hop forwarding steps on individual switches.

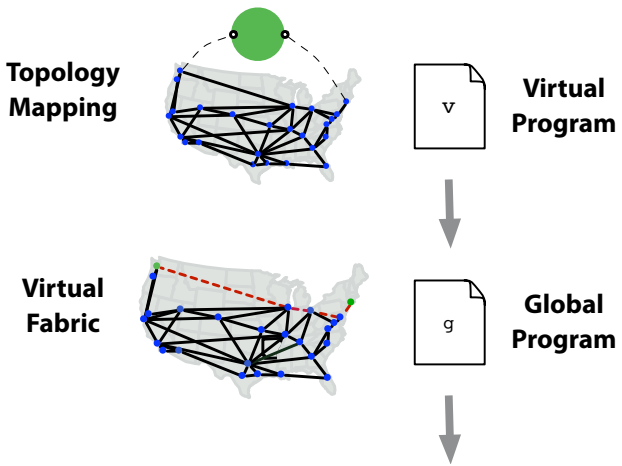
Global programs. Because NetKAT is based on Kleene algebra, it includes regular expressions, which are a natural and expressive formalism for describing paths through a network. Ideally, programmers would be able to use regular expressions to construct forwarding paths directly, without having to worry about how those paths were implemented. For example, a programmer might write the following to forward packets from port 1 on switch sw_1 to port 1 on switch sw_2, and from port 2 on sw_1 to port 2 on sw_2, assuming a link connecting the two switches on port 3:

$$\begin{aligned} &\mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}3 \cdot [sw_1{:}pt_3]{\rightarrow}[sw_2{:}pt_3] \cdot \mathsf{pt}{\leftarrow}1 \\ +\ &\mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}3 \cdot [sw_1{:}pt_3]{\rightarrow}[sw_2{:}pt_3] \cdot \mathsf{pt}{\leftarrow}2 \end{aligned}$$

Note that this is *not* a local program, since is not written in the general form given above and instead combines switch processing and topology processing using a particular combination of union and sequential composition to describe a pair of overlapping forwarding paths. To express the same behavior as a local NetKAT program or in a language such as Pyretic, we would have to somehow write a single program that specifies the processing that should be done at each intermediate step. The challenge is that when sw_2 receives a packet from sw_1, it needs to determine if that packet originated at port 1 or 2 of sw_1, but this can't be done without extra information. For example, the compiler could add a tag to packets at sw_1 to track the original ingress and use this information to determine the processing at sw_2. In general, the expressiveness of *global* programs

creates challenges for the compiler, which must generate explicit code to create and manipulate tags. These challenges have not been met in previous work on NetKAT or other SDN languages.

Virtual programs. Going a step further, NetKAT can also be used to specify behavior in terms of virtual topologies. To see why this is a useful abstraction, suppose that we wish to implement point-to-point connectivity between a given pair of hosts in a network with dozens of switches. One could write a global program that explicitly forwards along the path between these hosts. But this would be tedious for the programmer, since they would have to enumerate all of the intermediate switches along the path. A better approach is to express the program in terms of a virtual "big switch" topology whose ports are directly connected to the hosts, and where



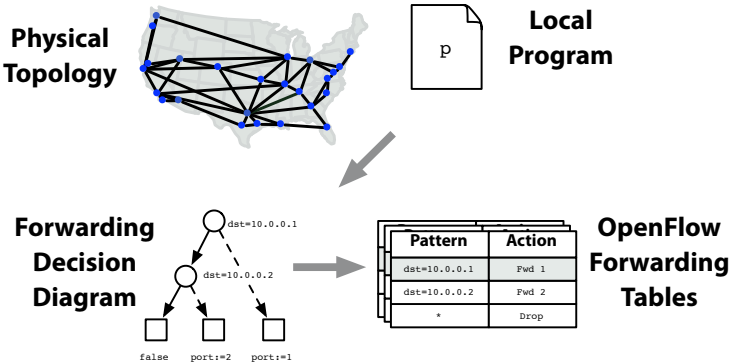


Figure 3: NetKAT compiler pipeline.

the relationship between ports in the virtual and physical networks is specified by an explicit mapping—*e.g.*, the top of Figure 3 depicts a big switch virtual topology. The desired functionality could then be specified using a simple local program that forwards in both directions between ports on the single virtual switch:

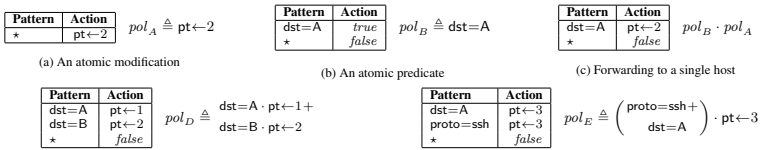
$$p \triangleq (\mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}2) + (\mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}1)$$

This one-switch virtual program is evidently much easier to write than a program that has to reference dozens of switches. In addition, the program is robust to changes in the underlying network. If the operator adds new switches to the network or removes switches for maintenance, the program remains valid and does not need to

be rewritten. In fact, this program could be ported to a completely different physical network too, provided it is able to implement the same virtual topology.

Another feature of virtualization is that the physical-virtual mapping can limit access to certain switches, ports, and even packets that match certain predicates, providing a simple form of language-based isolation [14]. In this example, suppose the physical network has hundreds of connected hosts. Yet, since the virtual-physical mapping only exposes two ports, the abstraction guarantees that the virtual program is isolated from the hosts connected to the other ports. Moreover, we can run several isolated virtual networks on the same physical network, *e.g.*, to provide different services to different customers in multi-tenant datacenters [19].

Of course, while virtual programs are a powerful abstraction, they create additional challenges for the compiler since it must generate physical paths that implement forwarding between virtual ports and also instrument programs with extra bookkeeping information to keep track of the locations of virtual packets traversing the physical network. Although virtualization has been extensively studied in the networking community [3, 7, 19, 25], no previous work fully describes how to compile virtual programs.



(a) An atomic modification

(b) An atomic predicate

(c) Forwarding to a single host

(d) Forwarding traffic to two hosts

(e) Monitoring SSH traffic and traffic to host A

Figure 2: Compiling using forwarding tables.

Compilation pipeline. This paper presents new algorithms for compiling NetKAT that address the key challenges related to expressiveness and performance just discussed. Figure 3 depicts the overall architecture of our compiler, which is structured as a pipeline with several smaller stages: (i) a *virtual compiler* that takes as input a virtual program v, a virtual topology, and a mapping that specifies the relationship between the virtual and physical topology, and emits a global program that uses a fabric to transit between virtual ports using physical paths; (ii) a *global compiler* that takes an arbitrary NetKAT program g as input and emits a local program that has been instrumented with extra state to keep track of the execution of the global program; and a (iii) *local compiler* that takes a local program p as input and generates OpenFlow forwarding tables, using a generalization of binary decision diagrams as an intermediate representation. Overall, our compiler automatically generates the extra state needed to implement virtual and global programs, with performance that is dramatically faster than current SDN compilers.

These three stages are designed to work well together—e.g., the fabric constructed by the virtual compiler is expressed in terms of

regular paths, which are translated to local programs by the global compiler, and the local and global compilers both use FDDs as an intermediate representation. However, the individual compiler stages can also be used independently. For example, the global compiler provides a general mechanism for compiling forwarding paths specified using regular expressions to SDN switches. We have also been working with the developers of Pyretic to improve performance by retargeting its backend to use our local compiler.

The next few sections present these stages in detail, starting with local compilation and building up to global and virtual compilation.

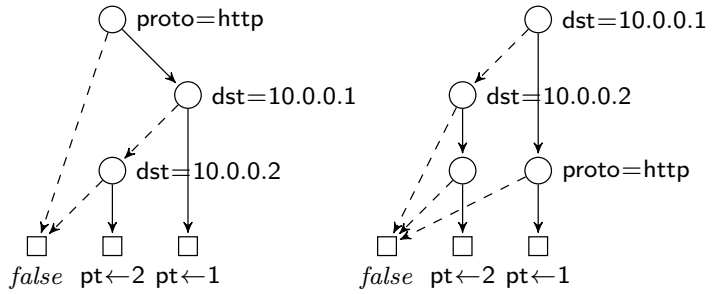
3. Local Compilation

The foundation of our compiler pipeline is a translation that maps local NetKAT programs to OpenFlow forwarding tables. Recall that a local program describes the hop-by-hop behavior of individual switches—i.e. it does not contain dup or switch modifications.

Compilation via forwarding tables. A simple approach to compiling local programs is to define a translation that maps primitive constructs to forwarding tables and operators such as union and sequential composition to functions that implement the analogous operations on tables. For example, the current NetKAT compiler translates the modification $\mathsf{pt}\leftarrow 2$ to a forwarding table with a single rule that sets the port of all packets to 2 (Figure 2 (a)), while it translates the predicate $\mathsf{dst}=\mathsf{A}$ to a flow table with two rules: the first matches packets where $\mathsf{dst}=\mathsf{A}$ and leaves them unchanged and the second matches all other packets and drops them (Figure 2 (b)).

To compile the sequential composition of these programs, the compiler combines each row in the first table with the entire second

table, retaining rules that could apply to packets produced by the



(a) proto ⊏ dst. (b) dst ⊏ proto.

Figure 4: Two ordered FDDs for the same program.

row (Figure 2 (c)). In the example, the second table has a single rule that sends all packets to port 2. The first rule of the first table matches packets with destination A, thus the second table is transformed to only send packets with destination A to port 2. However, the second rule of the first table drops all packets, therefore no packets ever reach the second table from this rule.

To compile a union, the compiler computes the pairwise intersection of all patterns to account for packets that may match both tables. For example, in Figure 2 (d), the two sub-programs forward traffic to hosts A and B based on the dst header. These two

sub-programs do not overlap with each other, which is why the table in the figure appears simple. However, in general, the two programs may overlap. Consider compiling the union of the forwarding program, in Figure 2 (d) and the monitoring program in Figure 2 (e). The monitoring program sends SSH packets and packets with dst=A to port 3. The intersection will need to consider all interactions between pairs of rules—an $\mathcal{O}(n^2)$ operation. Since a NetKAT program may be built out of several nested programs and compilation is quadratic at each step, we can easily get a tower of squares or exponential behavior.

Approaches based on flow tables are attractive for their simplicity, but they suffer several serious limitations. One issue is that tables are not an efficient way to represent packet-processing functions since each rule in a table can only encode positive tests on packet headers. In general, the compiler must emit sequences of prioritized rules to encode operators such as negation or union. Moreover, the algorithms that implement these operators are worst-case quadratic, which can cause the compiler to become a bottleneck on large inputs. Another issue is that there are generally many equivalent ways to encode the same packet-processing function as a forwarding table. This means that a straightforward computation of fixed-points, as is needed to implement Kleene star, is not guaranteed to terminate.

Syntax

Booleans $b ::= \top \mid \bot$
Contexts $\Gamma ::= \cdot \mid \Gamma, (f,n) : b$
Actions $a ::= \{f_1 \leftarrow n_1, \ldots, f_k \leftarrow n_k\}$
Diagrams $d ::= \{a_1, \ldots, a_k\}$ *Constant*
$\mid \ (f = n \,?\, d_1 : d_2)$ *Conditional*

Well Formedness

$\boxed{\Gamma \sqsubset (f,n)}$

$$\frac{}{\cdot \sqsubset (f,n)}\ \textsc{Nil}$$

$$\frac{f' \sqsubset f}{\Gamma, (f',n') : b' \sqsubset (f,n)}\ \textsc{Lt} \qquad \frac{f' = f \qquad n' \sqsubset n}{\Gamma, (f',n') : \bot \sqsubset (f,n)}\ \textsc{Eq}$$

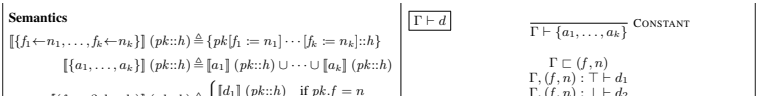


Figure 5: Forwarding decision diagrams: syntax, semantics, and well formedness.

Binary decision diagrams. To avoid these issues, our compiler is based on a novel representation of packet-forwarding functions using a generalization of *binary decision diagrams* (BDDs) [1, 6]. To briefly review, a BDD is a data structure that encodes a boolean function as a directed acyclic graph. The interior nodes encode boolean variables and have two outgoing edges: a true edge drawn as a solid line, and a false edge drawn as a dashed line. The leaf nodes encode constant values true or false. Given an assignment to the variables, we can evaluate the expression by following the appropriate edges in the graph. An *ordered* BDD imposes a total order in which the variables are visited. In general, the choice of variable-order can have a dramatic effect on the size of a BDD and hence on the run-time of BDD-manipulating operations. Picking an optimal variable-order is NP-hard, but efficient heuristics often work well in practice. A *reduced* BDD has no isomorphic subgraphs and every interior node has two distinct successors. A BDD can be reduced by repeatedly applying these two transformations:

- If two subgraphs are isomorphic, delete one by connecting its incoming edges to the isomorphic nodes in the other, thereby *sharing* a single copy of the subgraph.
- If both outgoing edges of an interior node lead to the same successor, eliminate the interior node by connecting its incoming

edges directly to the common successor node.

Logically, an interior node can be thought of as representing an IF-THEN-ELSE expression.[1] For example, the expression:

$$(a\,?\,(c\,?\,1 : (d\,?\,1 : 0)) : (b\,?\,(c\,?\,1 : (d\,?\,1 : 0)) : 0))$$

represents a BDD for the boolean expression $(a \vee b) \wedge (c \vee d)$. This notation makes the logical structure of the BDD clear while abstracting away from the sharing in the underlying graph representation and is convenient for defining BDD-manipulating algorithms.

In principle, we could use BDDs to directly encode NetKAT programs as follows. We would treat packet headers as flat, n-bit vectors and encode NetKAT predicates as n-variable BDDs. Since NetKAT programs produce sets of packets, we could represent them in a relational style using BDDs with $2n$ variables. However, there are two issues with this representation:

- Typical NetKAT programs modify only a few headers and leave the rest unchanged. The BDD that represents such a program would have to encode the identity relation between most of its input-output variables. Encoding the identity relation with BDDs requires a linear amount of space, so even trivial programs, such as the identity program, would require large BDDs.
- The final step of compilation needs to produce a prioritized flow table. It is not clear how to efficiently translate BDDs that represent NetKAT programs as relations into tables that represent packet-processing functions. For example, a table of length one is sufficient to represent the identity program, but to

[1] We write conditionals as $(a\,?\,b : c)$, in the style of the C ternary operator.

generate this table from the BDD sketched above, several paths would have to be compressed into a single rule.

Forwarding Decision Diagrams. To encode NetKAT programs as decision diagrams, we introduce a modest generalization of BDDs called *forwarding decision diagrams* (FDDs). An FDD differs from BDDs in two ways. First, interior nodes match header fields instead of individual bits, which means we need far fewer variables compared to a BDD to represent the same program. Our FDD implementation requires 12 variables (because OpenFlow supports 12 headers), but these headers span over 200 bits. Second, leaf nodes in an FDD directly encode packet modifications instead of boolean values. Hence, FDDs do not encode programs in a relational style.

Figures 4a and 4b show FDDs for a program that forwards HTTP packets to hosts 10.0.0.1 and 10.0.0.2 at ports 1 and 2 respectively. The diagrams have interior nodes that match on headers and leaf nodes corresponding to the actions used in the program.

To generalize ordered BDDs to FDDs, we assume orderings on fields and values, both written $\sqsubset$, and lift them to tests $f{=}n$ lexicographically:

$$f_1{=}n_1 \sqsubset f_2{=}n_2 \triangleq (f_1 \sqsubset f_2) \vee (f_1 = f_2 \wedge n_1 \sqsubset n_2)$$

We require that tests be arranged in ascending order from the root. For reduced FDDs, we stipulate that they must have no isomorphic subgraphs and that each interior node must have two unique successors, as with BDDs, and we also require that the FDD must not contain redundant tests and modifications. For example, if the test $\mathsf{dst}{=}10.0.0.1$ is true, then $\mathsf{dst}{=}10.0.0.2$ must be false. Accordingly, an FDD should not perform the latter test if the former succeeds. Similarly, because NetKAT's union operator $(p + q)$

is associative, commutative, and idempotent, to broadcast packets to both ports 1 and 2 we could either write $\mathsf{pt}{\leftarrow}1 \;+\; \mathsf{pt}{\leftarrow}2$ or $\mathsf{pt}{\leftarrow}2 \;+\; \mathsf{pt}{\leftarrow}1$. Likewise, repeated modifications to the same header are equivalent to just the final modification, and modifications to different headers commute. Hence, updating the dst header to 10.0.0.1 and then immediately re-updating it to 10.0.0.2 is the same as updating it to 10.0.0.2. In our implementation, we enforce the conditions for ordered, reduced FDDs by representing actions as

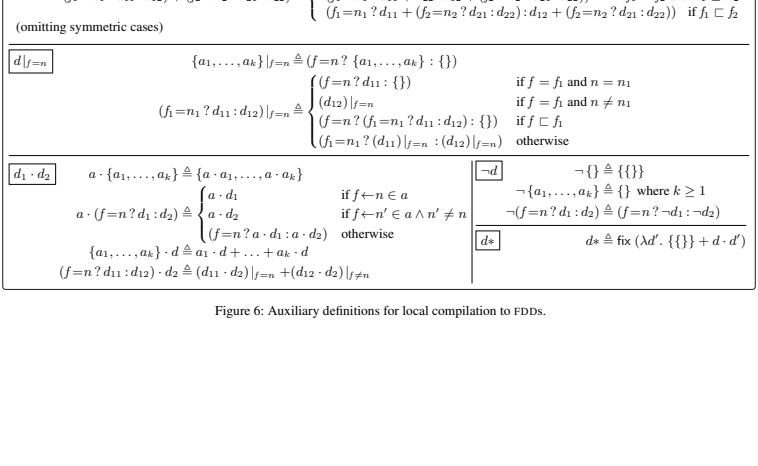


Figure 6: Auxiliary definitions for local compilation to FDDs.

$$\mathcal{L}[\![\mathit{false}]\!] \triangleq \{\} \qquad \mathcal{L}[\![f \leftarrow n]\!] \triangleq \{\{f \leftarrow n\}\}$$
$$\mathcal{L}[\![\mathit{true}]\!] \triangleq \{\{\}\} \qquad \mathcal{L}[\![f = n]\!] \triangleq (f = n\,?\,\{\{\}\} : \{\})$$
$$\mathcal{L}[\![\neg p]\!] \triangleq \neg\mathcal{L}[\![p]\!] \qquad \mathcal{L}[\![p_1 + p_2]\!] \triangleq \mathcal{L}[\![p_1]\!] + \mathcal{L}[\![p_2]\!]$$
$$\mathcal{L}[\![p^*]\!] \triangleq \mathcal{L}[\![p]\!]* \qquad \mathcal{L}[\![p_1 \cdot p_2]\!] \triangleq \mathcal{L}[\![p_1]\!] \cdot \mathcal{L}[\![p_2]\!]$$

Figure 7: Local compilation to FDDs.

sets of sets of modifications, and by using smart constructors that eliminate isomorphic subgraphs and contradictory tests.

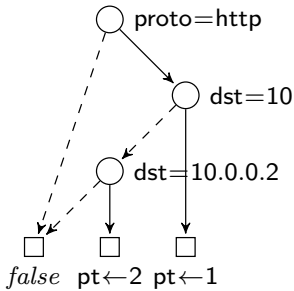
Figure 5 summarizes the syntax, semantics, and well-formedness conditions for FDDS formally. Syntactically, an FDD d is either a constant diagram specified by a set of actions $\{a_1, \ldots, a_k\}$, where an action a is a finite map $\{f_1 \leftarrow n_1, \ldots, f_k \leftarrow n_k\}$ from fields to values such that each field occurs at most once; or a conditional diagram $(f = n\,?\,d_1 : d_2)$ specified by a test $f = n$ and two subdiagrams. Semantically, an action a denotes a sequence of modifications, a constant diagram $\{a_1, \ldots, a_k\}$ denotes the union of the individual actions, and a conditional diagram $(f = n\,?\,d_1 : d_2)$ tests if the packet satisfies the test and evaluates the true branch (d_1) or false branch (d_2) accordingly. The well-formedness judgments $\Gamma \sqsubset (f, n)$ and $\Gamma \vdash d$ ensure that tests appear in ascending order and do not contradict previous tests to the same field. The context Γ keeps track of previous tests and boolean outcomes.

Local compiler. Now we are ready to present the local compiler itself, which goes in two stages. The first stage translates NetKAT source programs into FDDs, using the simple recursive translation given in Figures 6 and 7.

The NetKAT primitives $true$, $false$, and $f \leftarrow n$ all compile to simple constant FDDs. Note that the empty action set $\{\}$ drops all packets while the singleton action set $\{\{\}\}$ containing the identity action $\{\}$ copies packets verbatim. NetKAT tests $f = n$ compile to a conditional whose branches are the constant diagrams for $true$ and $false$ respectively. NetKAT union, sequence, negation, and star all recursively compile their sub-programs and combine the results using corresponding operations on FDDs, which are given in Figure 6.

The FDD union operator $(d_1 + d_2)$ walks down the structure of d_1 and d_2 and takes the union of the action sets at the leaves. However, the definition is a bit involved as some care is needed to preserve well-formedness. In particular, when combining multiple conditional diagrams into one, one must ensure that the ordering on tests is respected and that the final diagram does not contain contradictions. Readers familiar with BDDs may notice that this function is simply the standard "apply" operation (instantiated with union at the leaves). The sequential composition operator $(d_1 \cdot d_2)$ merges two packet-processing functions into a single function. It uses auxiliary operations $d \mid_{f=n}$ and $d \mid_{f \neq n}$ to restrict a diagram d by a positive or negative test respectively. We elide the sequence operator on atomic actions (which behaves like a right-biased merge of finite maps) and the negative restriction operator (which is similar to positive restriction, but not identical due to contradictory tests) to save space. The first few cases of the sequence operator handle situations where a single action on the left is composed with

a diagram on the right. When the diagram on the right is a conditional, $(f{=}n\,?\,d_1 : d_2)$, we partially evaluate the test using the modifications contained in the action on the left. For example, if the left-action contains the modification $f{\leftarrow}n$, we know that the test will be true, whereas if the left-action modifies the field to another value, we know the test will be false. The case that handles sequential composition of a conditional diagram on the left is also interesting. It uses restriction and union to implement the composition, reordering and removing contradictory tests as needed to ensure well formedness. The negation $\neg d$ operator is defined in the obvious way. Note that because negation can only be applied to predicates, the leaves of the diagram d are either $\{\}$ or $\{\{\}\}$. Finally, the FDD Kleene star operator $d*$ is defined using a straightforward fixed-point computation. The well-formedness conditions on FDDs ensures that a fixed point exists.



Pattern	Action
proto=http, dst=10.0.0.1	pt←1
proto=http, dst=10.0.0.2	pt←2
proto=http	*false*
⋆	*false*

Figure 8: Forwarding table generation example.

The soundness of local compilation from NetKAT programs to FDDs is captured by the following theorem:

Theorem 1 (Local Soundness)**.** *If* $\mathcal{L}[\![p]\!] = d$ *then* $[\![p]\!]\ h = [\![d]\!]\ h$.

Proof. Straightforward induction on p. □

The second stage of local compilation converts FDDs to forwarding tables. By design, this transformation is mostly straightforward: we generate a forwarding rule for every path from the root to a leaf, using the conjunction of tests along the path as the pattern and the actions at the leaf. For example, the FDD in Figure 8 has four paths from the root to the leaves so the resulting forwarding table has four rules. The left-most path is the highest-priority rule and the right-most path is the lowest-priority rule. Traversing paths from left to right has the effect of traversing true-branches

before their associated false-branches. This makes sense, since the only way to encode a negative predicate is to partially shadow a negative-rule with a positive-rule. For example, the last rule in the figure cannot encode the test proto≠http. However, since that rule is preceded by a pattern that tests proto=http, we can reason that the proto field is not HTTP in the last rule. If performed naively, this strategy could create a lot of extra forwarding rules—e.g., the table in Figure 8 has two drop rules, even though one of them completely shadows the other. In section 6, we discuss optimizations that eliminate redundant rules, exploiting the FDD representation.

4. Global Compilation

Thus far, we have seen how to compile local NetKAT programs into forwarding tables using FDDs. Now we turn to the global compiler, which translates global programs into equivalent local programs.

In general, the translation from global to local programs requires introducing extra state, since global programs may use regular expressions to describe end-to-end forwarding paths—e.g., recall the example of a global program with two overlapping paths from Section 2. Put another way, because a local program does not contain dup, the compiler can analyze the entire program and generate an equivalent forwarding table that executes on a single switch, whereas the control flow of a global program must be made explicit so execution can be distributed across multiple switches. More formally, a local program encodes a function from packets to sets of packets, whereas a global program encodes a function from packets to sets of packet-histories.

To generate the extra state needed to encode the control flow of a global, distributed execution into a local program, the global

compiler translates programs into finite state automata. To a first approximation, the automaton can be thought of as the one for the regular expression embedded in the global program, and the instrumented local program can be thought of as encoding the states and transitions of that automaton in a special header field. The actual construction is a bit more complex for several reasons. First, we cannot instrument the topology in the same way that we instrument switch terms. Second, we have to be careful not to introduce extra states that may lead to duplicate packet histories being generated. Third, NetKAT programs have more structure than ordinary regular expressions, since they denote functions on packet histories rather than sets of strings, so a more complicated notion of automaton—a symbolic NetKAT automaton—is needed.

At a high-level, the global compiler proceeds in several steps:

- It compiles the input program to an equivalent symbolic automaton. All valid paths through the automaton alternate between switch-processing states and topology-processing states, which enables executing them as local programs.
- It introduces a *program counter* by instrumenting the automaton to keep track of the current automaton state in the *pc* field.
- It determinizes the NetKAT automaton using an analogue of the subset construction for finite automata.
- It uses heuristic optimizations to reduce the number of states.
- It merges all switch-processing states into a single switch state and all topology-processing states into a single topology state.

The final result is a single local program that can be compiled using the local compiler. This program is equivalent to the original global

program, modulo the *pc* field, which records the automaton state.

4.1 NetKAT Automata

In prior work, some of the authors introduced NetKAT automata and proved the analogue of Kleene's theorem: programs and automata have the same expressive power [11]. This allows us to use automata as an intermediate representation for arbitrary NetKAT programs. This section reviews NetKAT automata, which are used in the global compiler, and then presents a function that constructs an automaton from an arbitrary NetKAT program.

Definition 1 (NetKAT Automaton)**.** *A NetKAT automaton is a tuple* $(S, s_0, \epsilon, \delta)$*, where:*

- S *is a finite set of states,*
- $s_0 \in S$ *is the start state,*
- $\epsilon : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk})$ *is the* observation function*, and*
- $\delta : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk} \times S)$ *is the* continuation function*.*

A NetKAT automaton is said to be *deterministic* if δ maps each packet to a unique next state at every state, or more formally if

$$|\left\{s' : S \mid (pk', s') \in \delta\, s\, pk\right\}| \leq 1$$

for all states s and packets pk and pk'.

The inputs to NetKAT automata are guarded strings drawn from the set $\mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^* \cdot \mathsf{Pk}$. That is, the inputs have the form

$$pk_{in} \cdot pk_1 \cdot \mathsf{dup} \cdot pk_2 \cdot \mathsf{dup} \cdots pk_n \cdot \mathsf{dup} \cdot pk_{out}$$

where $n \geq 0$. Intuitively, such strings represent packet-histories through a network: pk_{in} is the input state of a packet, pk_{out} is the output state, and the pk_i are the intermediate states of the packet

that are recorded as it travels through the network.

To process such a string, an automaton in state s can either *accept* the trace if $n = 0$ and $pk_{out} \in \epsilon\, s\, pk_{in}$, or it can consume one packet and dup from the start of the string and transition to state s' if $n > 0$ and $(pk_1, s') \in \delta\, s\, pk_{in}$. In the latter case, the automaton yields a residual trace:

$$pk_1 \cdot pk_2 \cdot \mathsf{dup} \cdots pk_n \cdot \mathsf{dup} \cdot pk_{out}$$

Note that the "output" pk_1 of state s becomes the "input" to the successor state s'. More formally, acceptance is defined as:

$$\begin{aligned} \mathsf{accept}\, s\, (pk_{in} \cdot pk_{out}) &\quad\Leftrightarrow\quad pk_{out} \in \epsilon\, s\, pk_{in} \\ \mathsf{accept}\, s\, (pk_{in} \cdot pk_1 \cdot \mathsf{dup} \cdot w) &\quad\Leftrightarrow\quad \bigvee_{(pk_1, s') \in \delta\, s\, pk_{in}} \mathsf{accept}\, s'(pk_1 \cdot w) \end{aligned}$$

p	$\mathcal{E}[\![p]\!] : \mathsf{Pol}$	$\mathcal{D}[\![p]\!] : \mathcal{P}(\mathsf{Pol} \times L \times \mathsf{Pol})$
a	a	$\varnothing$
$f \leftarrow n$	$f \leftarrow n$	$\varnothing$
dup^{ℓ}	false	$\{\langle \mathit{true}, \ell, \mathit{true} \rangle\}$
$q + r$	$\mathcal{E}[\![q]\!] + \mathcal{E}[\![r]\!]$	$\mathcal{D}[\![q]\!] \cup \mathcal{D}[\![r]\!]$
$q \cdot r$	$\mathcal{E}[\![q]\!] \cdot \mathcal{E}[\![r]\!]$	$\mathcal{D}[\![q]\!] \cdot r \cup \mathcal{E}[\![q]\!] \cdot \mathcal{D}[\![r]\!]$
q^*	$\mathcal{E}[\![q]\!]*$	$\mathcal{E}[\![q^*]\!] \cdot \mathcal{D}[\![q]\!] \cdot q^*$

Figure 9: Auxiliary definitions for NetKAT automata construction.

Next, we define a function that builds an automaton $\mathsf{A}(p)$ from an arbitrary NetKAT program p such that

$$(pk_{out} :: pk_n :: \ldots :: \langle pk_1 \rangle) \in [\![p]\!] \langle pk_{in} \rangle \quad \Leftrightarrow \quad \mathsf{accept}_{A(p)}\, s_0\, (pk_{in} \cdot pk_1 \cdot \mathsf{dup} \cdot \ldots \cdot pk_{out})$$

The construction is based on Antimirov partial derivatives for regular expressions [5]. We fix a set of labels L, and annotate each occurrence of dup in the source program p with a unique label $\ell \in L$. We then define a pair of functions:

- $\mathcal{E}[\![\cdot]\!] : \mathsf{Pol} \to \mathsf{Pol}$ and
- $\mathcal{D}[\![\cdot]\!] : \mathsf{Pol} \to \mathcal{P}(\mathsf{Pol} \times L \times \mathsf{Pol})$

Intuitively, $\mathcal{E}[\![p]\!]$ can be thought of as extracting the local components from p (and will be used to construct ϵ), while $\mathcal{D}[\![p]\!]$ extracts the global components (and will be used to construct δ). A triple

$\langle d, \ell, k\rangle \in \mathcal{D}[\![p]\!]$ represents the derivative of p with respect to dup^ℓ. That is, d is the dup-free component of p up to dup^ℓ, and k is the residual program (or *continuation*) of p after dup^ℓ.

We calculate $\mathcal{E}[\![p]\!]$ and $\mathcal{D}[\![p]\!]$ simultaneously using a simple recursive algorithm defined in Figure 9. The definition makes use of the following abbreviations,

$$\mathcal{D}[\![p]\!] \cdot q \triangleq \{\langle d, \ell, k \cdot q\rangle \mid \langle d, \ell, k\rangle \in \mathcal{D}[\![p]\!]\}$$
$$q \cdot \mathcal{D}[\![p]\!] \triangleq \{\langle q \cdot d, \ell, k\rangle \mid \langle d, \ell, k\rangle \in \mathcal{D}[\![p]\!]\}$$

which lift sequencing to sets of triples in the obvious way.

The next lemma characterizes $\mathcal{E}[\![p]\!]$ and $\mathcal{D}[\![p]\!]$, using the following notation to reconstruct programs from sets of triples:

$$\sum \mathcal{D}[\![p]\!] \triangleq \sum_{\langle d,\ell,k\rangle \in \mathcal{D}[\![p]\!]} d \cdot \mathsf{dup} \cdot k$$

Lemma 1 (Characterization of $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{D}[\![\cdot]\!]$)**.** *For all programs p, we have the following:*

(a) $p \equiv \mathcal{E}[\![p]\!] + \sum \mathcal{D}[\![p]\!]$.
(b) $\mathcal{E}[\![p]\!]$ *is a local program.*
(c) For all $\langle d, \ell, k\rangle \in \mathcal{D}[\![p]\!]$, d *is a local program.*
(d) For all labels ℓ *in* p*, there exist unique programs* d *and* k *such that* $\langle d, \ell, k\rangle \in \mathcal{D}[\![p]\!]$.

Proof. By structural induction on p. Claims $(b - d)$ are trivial. Claim (a) can be proved purely equationally using only the NetKAT axioms and the KAT-DENESTING rule from [4]. □

Lemma 1 (d) allows us to write k_ℓ to refer to the unique continua-

tion of dup^{ℓ}. By convention, we let k_0 denote the "initial continuation," namely p.

Definition 2 (Program Automaton)**.** *The NetKAT automaton* $\mathsf{A}(p)$ *for a program* p *is defined as* $(S, s_0, \epsilon, \delta)$ *where*

- S *is the set of labels occurring in* p*, plus the initial label* 0*.*
- $s_0 \triangleq 0$
- $\epsilon\, \ell\, pk \triangleq \{pk' \mid \langle pk' \rangle \in [\![\mathcal{E}[\![k_\ell]\!]]\!] \langle pk \rangle\}$
- $\delta\, \ell\, pk \triangleq \{(pk', \ell') \mid \langle d, \ell', k \rangle \in \mathcal{D}[\![k_\ell]\!] \wedge \langle pk' \rangle \in [\![d]\!] \langle pk \rangle\}$

Theorem 2 (Program Automaton Soundness)**.** *For all programs* p*, packets* pk *and histories* h*, we have*

$$h \in [\![p]\!]\langle pk_{in} \rangle \Leftrightarrow \mathsf{accept}\, s_0\, (pk_{in} \cdot pk_1 \cdot \mathsf{dup} \cdot \cdots \cdot pk_n \cdot \mathsf{dup} \cdot pk_{out})$$

where $h = pk_{out} :: pk_n :: \cdots :: \langle pk_1 \rangle$*.*

Proof. We first strengthen the claim, replacing $\langle pk_{in} \rangle$ with an arbitrary history $pk_{in} :: h'$, s_0 with an arbitrary label $\ell \in S$, and p with k_ℓ. We then proceed by induction on the length of the history, using Lemma 1 for the base case and induction step. □

4.2 Local Program Generation

With a NetKAT automaton $\mathsf{A}(p)$ for the global program p in hand, we are now ready to construct a local program. The main idea is to make the state of the global automaton explicit in the local program by introducing a new header field pc (represented concretely using VLANs, MPLS tags, or any other unused header field) that keeps track of the state as the packet traverses the network. This encoding enables simulating the automaton for the global program using a

single local program (along with the physical topology). We also discuss determinization and optimization, which are important for correctness and performance.

Program counter. The first step in local program generation is to encode the state of the automaton into its observation and transition functions using the pc field. To do this, we use the same structures as are used by the local compiler, FDDs. Recall that the observation function ϵ maps input packets to output packets according to $\mathcal{E}[\![k_\ell]\!]$, which is a dup-free NetKAT program. Hence, we can encode the observation function for a given state ℓ as a conditional FDD that tests whether pc is ℓ and either behaves like the FDD for $\mathcal{E}[\![k_\ell]\!]$ or $false$. We can encode the continuation function δ as an FDD in a similar fashion, although we also have to set the pc to each successor state s'. This symbolic representation of automata using FDDs allows us to efficiently manipulate automata despite the large size of their "input alphabet", namely $|\mathsf{Pk} \times \mathsf{Pk}|$. In our implementation we introduce the pc field and FDDs on the fly as automata are constructed, rather than adding them as a post-processing step, as is described here for ease of exposition.

Determinization. The next step in local program generation is to determinize the NetKAT automaton. This step turns out to be critical for correctness—it eliminates extra outputs that would be produced if we attempted to directly implement a nondeterministic NetKAT automaton. To see why, consider a program of the form $p + p$. Intuitively, because union is an idempotent operation, we expect that this program will behave the same as just a single copy of p. However, this will not be the case when p contains a dup: each occurrence of dup will be annotated with a different label. There-

fore, when we instrument the program to track automaton states, it will create two packets that are identical expect for the pc field, instead of one packet as required by the semantics. The solution to this problem is simply to determinize the automaton before converting it to a local program. Determinization ensures that every packet trace induces a unique path through the automaton and prevents duplicate packets from being produced. Using FDDs to represent the automaton symbolically is crucial for this step: it allows us to implement a NetKAT analogue of the subset construction efficiently.

Optimization. One practical issue with building automata using the algorithms described so far is that they can use a large number of states—one for each occurrence of dup in the program—and determinization can increase the number of states by an exponential factor. Although these automata are not wrong, attempting to compile them can lead to practical problems since extra states will

trigger a proliferation of forwarding rules that must be installed on switches. Because switches today often have limited amounts of memory—often only a few thousand forwarding rules—reducing the number of states is an important optimization. An obvious idea is to optimize the automaton using (generalizations of) textbook minimization algorithms. Unfortunately this would be prohibitively expensive since deciding whether two states are equal is a costly operation in the case of NetKAT automata. Instead, we adopt a simple heuristic that works well in practice and simply merge states that are identical. In particular, by representing the observation and transition functions as FDDs, which are hash consed, testing equality is cheap—simple pointer comparisons.

Local Program Extraction. The final step is to extract a local program from the automaton. Recall from Section 2 that, by definition, links are enclosed by dups on either side, and links are the only NetKAT terms that contain dups or modify the switch field. It follows that every global program gives rise to a bipartite NetKAT automaton in which all accepting paths alternate between "switch states" (which do not modify the switch field) and "link states" (which forward across links and do modify the switch field), beginning with a switch state. Intuitively, the local program we want to extract is simply the union of of the ϵ and δ FDDs of all switch states (recall Lemma 1 (a)), with the link states implemented by the physical network. Note however, that the physical network will neither match on the pc nor advance the pc to the next state (while the link states in our automaton do). To fix the latter, we observe that any link state has a unique successor state. We can thus simply advance the pc by two states instead of one at every switch state, anticipating the missing pc modification in link states. To address

the former, we employ the equivalence

$$[sw_1{:}pt_1] \twoheadrightarrow [sw_2{:}pt_2] \equiv sw{=}1 \cdot pt{=}1 \cdot t \cdot sw{=}2 \cdot pt{=}2$$

It allows us to replace links with the entire topology if we modify switch states to match on the appropriate source and destination locations immediately before and after transitioning across a link. After modifying the ϵ and δ FDDs accordingly and taking the union of all switch states as described above, the resulting FDD can be passed to the local compiler to generate forwarding tables.

The tables will correctly implement the global program provided the physical topology (in, t, out) satisfies the following:

- $p \equiv in \cdot p \cdot out$, i.e. the global program specifies end-to-end forwarding paths
- t implements at least the links used in p.
- $t \cdot in \equiv false \equiv out \cdot t$, *i.e.* the in and out predicates should not include locations that are internal to the network.

5. Virtual Compilation

The third and final stage of our compiler pipeline translates virtual programs to physical programs. Recall that a virtual program is one that is defined over a virtual topology. Network virtualization can make programs easier to write by abstracting complex physical topologies to simpler topologies and also makes programs portable across different physical topologies. It can even be used to multiplex several virtual networks onto a single physical network—*e.g.*, in multi-tenant datacenters [19].

To compile a virtual program, the compiler needs to know the

mapping between virtual switches, ports, and links and their counterparts at the physical level. The programmer supplies a virtual program v, a virtual topology t, sets of ingress and egress locations for t, and a relation $\mathcal{R}$ between virtual and physical ports. The relation $\mathcal{R}$ must map each physical ingress to a virtual ingress, and conversely for egresses, but is otherwise unconstrained—e.g., it need not be injective or even a function.[2] The constraints on ingresses and egresses ensures that each packet entering the physical network lifts uniquely to a packet in the virtual network, and similarly for packets editing the virtual network. During execution of the virtual program, each packet can be thought of as having two locations, one in the virtual network and one in the physical network; $\mathcal{R}$ defines which pairs of locations are consistent with each other. For simplicity, we assume the virtual program is a local program. If it is not, the programmer can use the global compiler to put it into local form.

Overview. To execute a virtual program on a physical network, possibly with a different underlying topology, the compiler must (i) instrument the program to keep track of packet locations in the virtual topology and (ii) implement forwarding between locations that are adjacent in the virtual topology using physical paths. To achieve this, the virtual compiler proceeds as follows:

1. It instruments the program to use the virtual switch (`vsw`) and virtual port (`vpt`) fields that track of the location of the packet in the virtual topology.
2. It constructs a *fabric*: a NetKAT program that updates the physical location of a packet when its virtual location changes and vice versa, after each step of processing to restore consistency

with respect to the virtual-physical relation, $\mathcal{R}$.

3. It assembles the final program by combining v with the fabric, eliminating the vsw and vpt fields, and compiling the result using the global compiler.

Most of the complexity arises in the second step because there may be many valid fabrics (or there may be none). However, this step is independent of the virtual program. The fabric can be computed once and for all and then be reused as the program changes. Fabrics can be generated in several ways—*e.g.*, to minimize a costs such as path length or latency, maximize disjointness, etc.

Instrumentation. To keep track of a packet's location in the virtual network, we introduce new packet fields vsw and vpt for the virtual switch and the virtual port, respectively. We replace all occurrences of the sw or pt field in the program v and the virtual topology t with vsw and vpt respectively using a simple textual substitution. Packets entering the physical network must be lifted to the virtual network. Hence, we replace in with a program that matches on all physical ingress locations $\mathbb{I}$ and initializes vsw and vpt in accordance with $\mathcal{R}$:

$$in' \triangleq \sum_{\substack{(sw,pt)\in\mathbb{I} \\ (vsw,vpt)\ \mathcal{R}\ (sw,pt)}} \mathsf{sw}{=}sw \cdot \mathsf{pt}{=}pt \cdot \mathsf{vsw}{\leftarrow}vsw \cdot \mathsf{vpt}{\leftarrow}vpt$$

Recall that we require $\mathcal{R}$ to relate each location in $\mathbb{I}$ to at most one virtual ingress, so the program lifts each packet to at most one ingress location in the virtual network. The vsw and vpt fields are only used to track locations during the early stages of virtual compilation. They are completely eliminated in the final assembly.

Hence, we will not need to introduce additional tags to implement the resulting physical program.

Fabric construction. Each packet can be thought of as having two locations: one in the virtual topology and one in the underlying physical topology. After executing in', the locations are consistent according to the virtual-physical relation $\mathcal{R}$. However, consistency can be broken after each step of processing using the virtual program v or virtual topology t. To restore consistency, we construct

[2] Actually, we can relax this condition slightly and allow physical ingresses to map to zero or one virtual ingresses—if a physical ingress has no corresponding representative in the virtual network, then packets arriving at that ingress will not be admitted to the virtual network.

$$\frac{(vsw, vpt, \mathtt{I}) \rightarrow_v (vsw, vpt', \mathtt{O})}{\begin{bmatrix}(vsw, vpt, \mathtt{I})\\(sw, pt, \mathtt{I})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw, vpt', \mathtt{O})\\(sw, pt, \mathtt{I})\end{bmatrix}}\ \mathcal{V}\text{-POL}$$

$$\frac{(vsw, vpt, \mathtt{O}) \rightarrow_v (vsw', vpt', \mathtt{I})}{\begin{bmatrix}(vsw, vpt, \mathtt{O})\\(sw, pt, \mathtt{O})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw', vpt', \mathtt{I})\\(sw, pt, \mathtt{O})\end{bmatrix}}\ \mathcal{V}\text{-TOPO}$$

$$\frac{\begin{array}{c}(sw, pt, \mathtt{I}) \rightarrow_p^+ (sw', pt', \mathtt{O})\\(vsw, vpt)\ \mathcal{R}\ (sw', pt')\end{array}}{\begin{bmatrix}(vsw, vpt, \mathtt{O})\\(sw, pt, \mathtt{I})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw, vpt, \mathtt{O})\\(sw', pt', \mathtt{O})\end{bmatrix}}\ \mathcal{F}\text{-OUT}$$

$$\frac{\begin{array}{c}(sw, pt, \mathtt{O}) \rightarrow_p^+ (sw', pt', \mathtt{I})\\(vsw, vpt)\ \mathcal{R}\ (sw', pt')\end{array}}{\begin{bmatrix}(vsw, vpt, \mathtt{I})\\(sw, pt, \mathtt{O})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw, vpt, \mathtt{I})\\(sw', pt', \mathtt{I})\end{bmatrix}}\ \mathcal{F}\text{-IN}$$

$$\frac{(vsw, vpt)\ \mathcal{R}\ (sw, pt)}{\begin{bmatrix}(vsw, vpt, \mathtt{I})\\(sw, pt, \mathtt{O})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw, vpt, \mathtt{I})\\(sw, \mathsf{Loop}\ pt, \mathtt{I})\end{bmatrix}}\ \mathcal{F}\text{-LOOP-IN}$$

$$(sw, pt, \mathtt{O}) \rightarrow_p^* (sw', pt', \mathtt{O})$$

$$(vsw, vpt)\ \mathcal{R}\ (sw', pt')$$

$$\frac{(vsw, vpt) \mathrel{\mathcal{R}} (sw', pt')}{\begin{bmatrix}(vsw, vpt, \texttt{0})\\(sw, \texttt{Loop}\ pt, \texttt{I})\end{bmatrix} \rightarrow \begin{bmatrix}(vsw, vpt, \texttt{0})\\(sw', pt', \texttt{0})\end{bmatrix}}\ \mathcal{F}\text{-LOOP-OUT}$$

Figure 10: Fabric game graph edges.

a *fabric* comprising programs f_{in} and f_{out} from the virtual and physical topologies and $\mathcal{R}$, and insert it into the program:

$$q \triangleq in' \cdot (v \cdot f_{out}) \cdot (t \cdot f_{in} \cdot v \cdot f_{out})^* \cdot out$$

In this program, v and t alternate with f_{out} and f_{in} in processing packets, thereby breaking and restoring consistency repeatedly. Intuitively, it is the job of the fabric to keep the virtual and physical locations in sync.

This process can be viewed as a two-player game between a virtual player $\mathcal{V}$ (embodied by v and t) and a fabric player $\mathcal{F}$ (embodied by f_{out} and f_{in}). The players take turns moving a packet across the virtual and the physical topology, respectively. Player $\mathcal{V}$ wins if the fabric player $\mathcal{F}$ fails to restore consistency after a finite number of steps; player $\mathcal{F}$ wins otherwise. Constructing a fabric now amounts to finding a winning strategy for $\mathcal{F}$.

We start by building the game graph $G = (V, E)$ modeling all possible ways that consistency can be broken by $\mathcal{V}$ or restored by $\mathcal{F}$. Nodes are pairs of virtual and physical locations, $[l_v, l_p]$, where a location is a 3-tuple comprising a switch, a port, and a direction that indicates if the packet entering the port (`I`) leaving the port (`O`). The rules in Figure 10 determine the edges of the game graph:

- The edge $[l_v, l_p] \rightarrow [l'_v, l_p]$ exists if $\mathcal{V}$ can move packets from l_v to l'_v. There are two ways to do so: either $\mathcal{V}$ moves packets across a virtual switch ($\mathcal{V}$-POL) or across a virtual link ($\mathcal{V}$-TOPO). In the inference rules, we write $\rightarrow_v$ to denote a single hop in the virtual topology:

$$(vsw, vpt, d) \rightarrow_v (vsw', vpt', d')$$

Reachable Nodes

$$\frac{(sw, pt) \in \mathbb{I} \qquad (vsw, vpt)\ \mathcal{R}\ (sw, pt)}{\begin{bmatrix}(vsw, vpt, \mathtt{I}) \\ (sw, pt, \mathtt{I})\end{bmatrix} \in V}\ \textsc{Ing}$$

$$\frac{u \in V \qquad u \rightarrow v}{v \in V}\ \textsc{Trans}$$

Fatal Nodes

$$\frac{v = \begin{bmatrix}(vsw, vpt, d_1) \\ (sw, pt, d_2)\end{bmatrix} \in V \qquad d_1 \neq d_2 \qquad \forall u.\ v \rightarrow u \Rightarrow u \text{ is fatal}}{v \text{ is fatal}}\ \mathcal{F}\text{-FATAL}$$

$$\frac{v = \begin{bmatrix}(vsw, vpt, d_1) \\ (sw, pt, d_2)\end{bmatrix} \in V \qquad d_1 = d_2 \qquad \exists u.\ v \rightarrow u \wedge u \text{ is fatal}}{v \text{ is fatal}}\ \mathcal{V}\text{-FATAL}$$

Figure 11: Reachable and fatal nodes.

if $d = \texttt{I}$ and $d' = \texttt{O}$ then the hop is across one switch, but if $d = \texttt{O}$ and $d' = \texttt{I}$ then the hop is across a link.

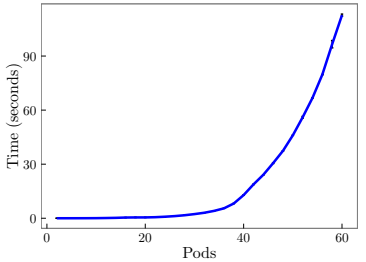
- The edge $[l_v, l_p] \rightarrow [l_v, l'_p]$ exists if $\mathcal{F}$ can move packets from l_p to l'_p. When $\mathcal{F}$ makes a move, it must restore physical-virtual consistency (the $\mathcal{R}$ relation in the premise of $\mathcal{F}$-POL and $\mathcal{F}$-TOPO). To do so, it may need to take several hops through the physical network (written as $\rightarrow_p^+$).
- In addition, $\mathcal{F}$ may leave a packet at their current location, if the location is already consistent ($\mathcal{F}$-LOOP-IN and $\mathcal{F}$-LOOP-OUT). Note that these force a packet located at physical location $(sw, pt, \texttt{O})$ to leave through port pt eventually. Intuitively, once the fabric has committed to emitting the packet through a given port, it can only delay but not withdraw that commitment.

Although these rules determine the complete game graph, all packets enter the network at an ingress location (determined by the in' predicate). Therefore, we can restrict our attention to only those nodes that are reachable from the ingress (reachable nodes in Figure 11). In the resulting graph $G = (V, E)$, every path represents a possible trajectory that a packet processed by q may take through the virtual and physical topology.

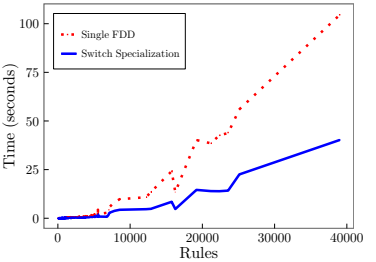
In addition to removing unreachable nodes, we must remove *fatal nodes*, which are the nodes where $\mathcal{F}$ is unable to restore consistency and thus loses the game. $\mathcal{F}$-FATAL says that any state from which $\mathcal{F}$ is unable to move to a non-fatal state is fatal. In particular, this includes states in which $\mathcal{F}$ cannot move to any other state at all. $\mathcal{V}$-FATAL says that any state in which $\mathcal{V}$ can move to a

fatal state is fatal. Intuitively, we define such states to be fatal since we want the fabric to work for any virtual program the programmer may write. Fatal states can be removed using a simple backwards traversal of the graph starting from nodes without outgoing edges. This process may remove ingress nodes if they turn out to be fatal. This happens if and only if there exists no fabric that can always restore consistency for arbitrary virtual programs. Of course, this case can only arise if the physical topology is not bidirectional.

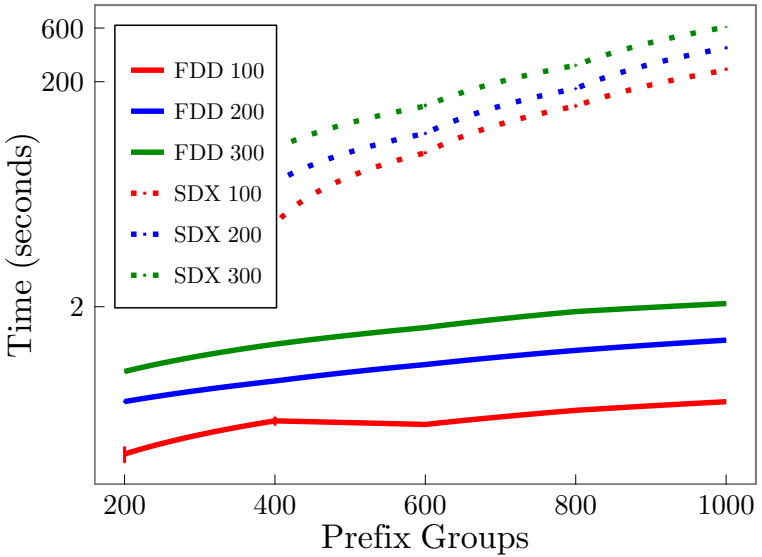
Fabric selection. If all ingress nodes withstand pruning, the resulting graph encodes exactly the set of all winning strategies for $\mathcal{F}$, *i.e.* the set of all possible fabrics. A fabric is a subgraph of G that contains the ingress, is closed under all possible moves by the virtual program, and contains exactly one edge out of every state in



(a) Routing on k-pod fat-trees.



(b) Destination-based routing on topology zoo.



(c) Time needed to compile SDX benchmarks.

Figure 12: Experimental results: compilation time.

which $\mathcal{F}$ has to restore consistency. The $\mathcal{F}$-edges must be labeled with concrete paths through the physical topology, as there may exist several paths implementing the necessary multi-step transportation from the source node to the target node.

In general, there may be many fabrics possible and the choice of different $\mathcal{F}$-edges correspond to fabrics with different characteristics, such as minimizing hop counts, maximizing disjoint paths, and so on. Our compiler implements several simple strategies. For example, given a metric ϕ on paths (such as hop count), our greedy strategy starts at the ingresses and adds a node whenever it is reachable through an edge e rooted at a node u already selected, and e is (i) any $\mathcal{V}$-player edge or (ii) the $\mathcal{F}$-player edge with path π minimizing ϕ among all edges and their paths rooted at u.

After a fabric is selected, it is straightforward to encode it as a NetKAT term. Every $\mathcal{F}$-edge $[l_v, l_p] \rightarrow [l_v, l'_p]$ in the graph is encoded as a NetKAT term that matches on the locations l_v and l_p, forwards along the corresponding physical path from l_p to l'_p, and then resets the virtual location to l_v. Resetting the virtual location is semantically redundant but will make it easy to eliminating the vsw and vpt fields. We then take f_{in} to be the union of all $\mathcal{F}$-IN-edges, and f_{out} to be the union of all $\mathcal{F}$-OUT-edges. NetKAT's global abstractions play a key role, providing the building blocks for composing multiple overlapping paths into a unified fabric.

End-to-end Compilation. After the programs in', f_{in}, and f_{out}, are calculated from $\mathcal{R}$, we assemble the physical program q, defined above. However, one last potential problem remains: although the virtual compiler adds instrumentation to update the physical switch and port fields, the program still matches and updates the virtual switch (vsw) and virtual port (vpt). However, note that by construction of q, any match on the vsw or vpt field is preceded by a modification of those fields on the same physical switch. Therefore, all matches are automatically eliminated during FDD generation, and only modifications of the vsw and vpt fields remain. These

can be safely erased before generating flow tables as the global compiler inserts a program counter into q that plays double-duty to track both the physical location and the virtual location of a packet. Hence, we only need a single tag to compile virtual programs!

6. Evaluation

To evaluate our compiler, we conducted experiments on a diverse set of real-world topologies and benchmarks. In practice, our compiler is a module that is used by the Frenetic SDN controller to map NetKAT programs to flow tables. Whenever network events occur, *e.g.*, a host connects, a link fails, traffic patterns change, and so on, the controller may react by generating a new NetKAT program. Since network events may occur rapidly, a slow compiler can easily be a bottleneck that prevents the controller from reacting quickly to network events. In addition, the flow tables that the compiler generates must be small enough to fit on the available switches. Moreover, as small tables can be updated faster than large tables, table size affects the controller's reaction time too.

Therefore, in all the following experiments we measure flow-table compilation time and flow-table size. We apply the compiler to programs for a variety of topologies, from topology designs for very large datacenters to a dataset of real-world topologies. We highlight the effect of important optimizations to the fundamental FDD-based algorithms. We perform all experiments on 32-core, 2.6 GHz Intel Xeon E5-2650 machines with 64GB RAM.[3] We repeat all timing experiments ten times and plot their average.

Fat trees. A fat-tree [2] is a modern datacenter network design that uses commodity switches to minimize cost. It provides sev-

eral redundant paths between hosts that can be used to maximize available bandwidth, provide backup paths, and so on. A fat-tree is organized into pods, where a k-pod fat-tree topology can support up to $\frac{k^3}{4}$ hosts. A real-world datacenter might have up to 48 pods [2]. Therefore, our compiler should be able to generate forwarding programs for a 48-pod fat tree relatively quickly.

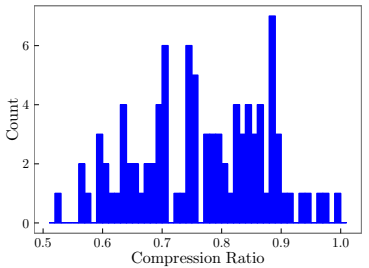
Figure 12a shows how the time needed to generate all flow tables varies with the number of pods in a fat-tree.[4] The graph shows that we take approximately 30 seconds to produce tables for 48-pod fat trees (*i.e.*, 27,000 hosts) and less than 120 seconds to generate programs for 60-pod fat trees (*i.e.*, 54,000 hosts).

This experiment shows that the compiler can generate tables for large datacenters. But, this is partly because the fat-tree forwarding algorithm is topology-dependent and leverages symmetries to minimize the amount of forwarding rules needed. Many real-world topologies are not regular and require topology-independent forwarding programs. In the next section, we demonstrate that our compiler scales well with these topologies too.

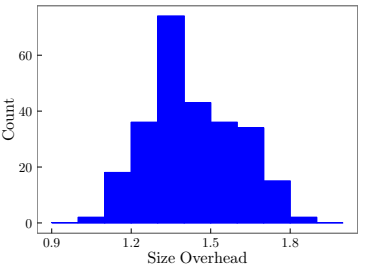
Topology Zoo. The *Topology Zoo* [18] is a dataset of a few hundred real-world network topologies of varying size and structure. For every topology in this dataset, we use *destination-based routing* to connect all nodes to each other. In destination-based routing, each switch filters packets by their destination address and forwards them along a spanning-tree rooted at the destination. Since each switch must be able to forward to any destination, the total number of rules must be $\mathcal{O}(n^2)$ for an n-node network.

[3] Our compiler is single-threaded and doesn't leverage multicore.

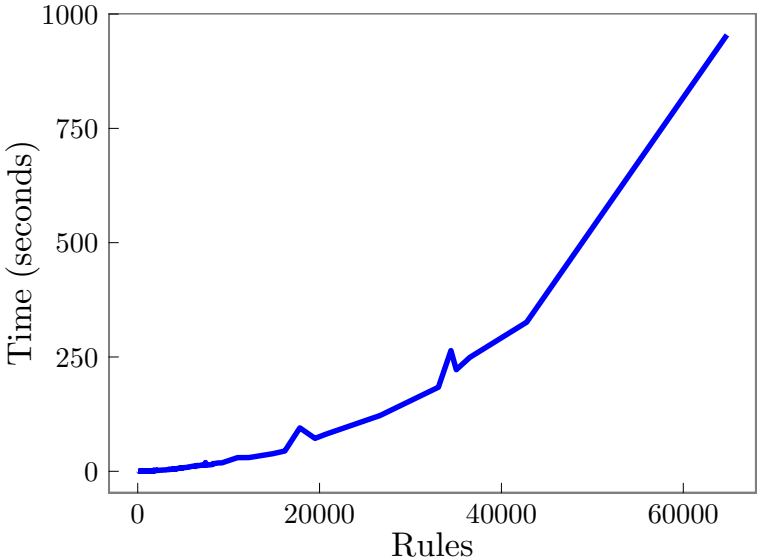
[4] This benchmark uses the switch-specialization optimization, which we describe in the next section.



(a) Compressing Classbench ACLs.



(b) Table size overhead for global programs.



(c) Compilation time for global programs.

Figure 13: Experimental results: forwarding table compression and global compilation.

Figure 12b shows how the running time of the compiler varies across the topology zoo benchmarks. The curves are not as smooth as the curve for fat-trees, since the complexity of forwarding depends on features of network topology. Since the topology zoo is

so diverse, this is a good suite to exercise the *switch specialization* optimization that dramatically reduces compile time.

A direct implementation builds of the local compiler builds one FDD for the entire network and uses it to generate flow tables for each switch. However, since several FDD (and BDD) algorithms are fundamentally quadratic, it helps to first specialize the program for each switch and then generate a small FDD for each switch in the network (*switch specialization*). Building FDDs for several smaller programs is typically much faster than building a single FDD for the entire network. As the graph shows, this optimization has a dramatic effect on all but the smallest topologies.

SDX. Our experiments thus far have considered some quite large forwarding programs, but none of them leverage software-defined networking in any interesting way. In this section, we report on our performance on benchmarks from a recent SIGCOMM paper [13] that proposes a new application of SDN.

An Internet exchange point (IXP) is a physical location where networks from several ISPs connect to each other to exchange traffic. Legal contracts between networks are often implemented by routing programs at IXPs. However, today's IXPs use baroque protocols the needlessly limit the kinds of programs that can be implemented. A Software-defined IXP (an "SDX" [13]) gives participants fine-grained control over packet-processing and peering using a high-level network programming language. The SDX prototype uses Pyretic [25] to encode policies and presents several examples that demonstrate the power of an expressive network programming language.

We build a translator from Pyretic to NetKAT and use it to eval-

uate our compiler on SDXs own benchmarks. These benchmarks simulate a large IXP where a few hundred peers apply programs to several hundred prefix groups. The dashed lines in Figure 12c reproduce a graph from the SDX paper, which shows how compilation time varies with the number of prefix groups and the number of participants in the SDX.[5] The solid lines show that our compiler is orders of magnitude faster. Pyretic takes over 10 minutes to compile the largest benchmark, but our compiler only takes two seconds.

Although Pyretic is written in Python, which is a lot slower than OCaml, the main problem is that Pyretic has a simple table-based compiler that does not scale (Section 2). In fact, the authors of SDX had to add several optimizations to get the graph depicted. Despite these optimizations, our FDD-based approach is substantially faster.

[5] We get nearly the same numbers as the SDX paper on our hardware.

The SDX paper also reports flow-table sizes for the same benchmark. At first, our compiler appeared to produce tables that were twice as large as Pyretic. Naturally, we were unhappy with this result and investigated. Our investigation revealed a bug in the Pyretic compiler, which would produce incorrect tables that were artificially small. The authors of SDX have confirmed this bug and it has been fixed in later versions of Pyretic. We are actively working with them to port SDX to NetKAT to help SDX scale further.

Classbench. Lastly, we compile ACLs generated using *Classbench* [32]. These are realistic firewall rules that showcase another optimization: it is often possible to significantly compress tables by combining and eliminating redundant rules.

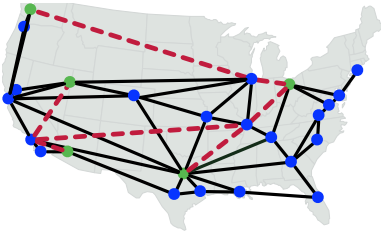
We build an optimizer for the flow-table generation algorithm

in Figure 8. Recall that that we generate flow-tables by converting every complete path in the FDD into a rule. Once a path has been traversed, we can remove it from the FDD without harm. However, naively removing a path may produce an FDD that is not reduced. Our optimization is simple: we remove paths from the FDD as they are turned into rules and ensure that the FDD is reduced at each step. When the last path is turned into a rule, we are left with a trivial FDD. This iterative procedure prevents several unnecessary rules from being generated. It is possible to implement other canonical optimizations. But, this optimization is unique because it leverages properties of reduced FDDs. Figure 13a shows that this approach can produce 30% fewer rules on average than a direct implementation of flow-table generation. We do not report running times for the optimizer, but it is negligible in all our experiments.

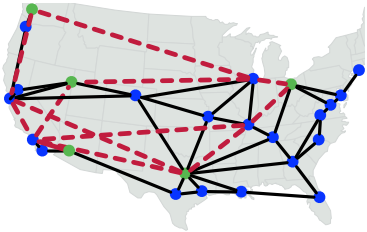
Global compiler. The benchmarks discussed so far only use the local compiler. In this section, we focus on the global compiler. Since the global compiler introduces new abstractions, we can't apply it to existing benchmarks, such as SDX, which use local programs. Instead, we need to build our own benchmark suite of global programs. To do so, we build a generator that produces global programs that describe paths between hosts. Again, an n-node topology has $O(n^2)$ paths. We apply this generator to the Topology Zoo, measuring compilation time and table size:

- *Compilation time:* since the global compiler leverages FDDs, we can expect automaton generation to be fast. However, global compilation involves other steps such as determinization and localization and their effects on compilation time may matter. Figure 13c shows how compilation time varies with the total

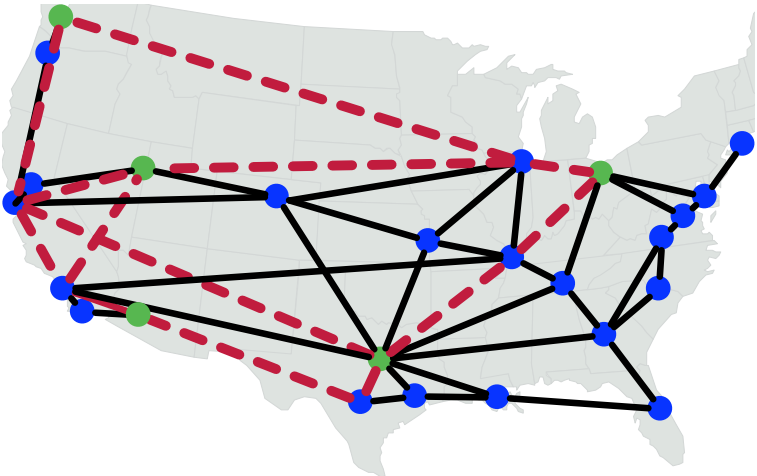
number of rules generated. This graph does grow faster than local compilation time on the same benchmark (the red, dashed



(a) minimum total number of links



(b) minimum number of hops



(c) minimum distance

Figure 14: Three fabrics optimizing different metrics

line in Figure 12b). Switch-specialization, which dramatically reduces the size of FDDs and hence compilation time, does not work on global programs. Therefore, it makes most sense to compare this graph to local compilation with a single FDD.

- *Table size:* The global compiler has some optimizations to eliminate unnecessary states, which produces fewer rules. However, it it does not fully minimize NetKAT automata thus it may produce more rules than equivalent local programs. Figure 13b shows that on the topology zoo, global routing produces tables that are no more than twice as large as local routing.

We belive these results are promising: we spent a lot of time tuning the local compiler, but the global compiler is an early prototype with much room for improvement.

Virtualization case study. Finally, we present a small case study that showcases the virtual compiler on a snapshot of the AT&T backbone network circa 2007–2008. This network is part of the Topology Zoo and shown in Figure 14. We construct a "one big switch" virtual network and use it to connect five nodes (highlighted in green) to each other:

$$\sum_{n=1}^{5} \mathsf{dst}{=}10.0.0.n \cdot \mathsf{pt}{\leftarrow}n$$

To map the virtual network to the physical network, we generate

three different fabrics: (a) a fabric that minimizes the total number of links used across the network, (b) a fabric that minimizes the number of hops between hosts, and (c) a fabric that minimizes the physical length of the path between hosts. In the figure, the links utilized by each of these fabrics is highlighted in red.

The three fabrics give rise to three very different implementations of the same virtual program. Note that the program and the fabric are completely independent of each other and can be updated independently. For example, the operator managing the physical network could change the fabric to implement a new SLA, *e.g.* move from minimum-utilization to shortest-paths. This change requires no update to the virtual program; the network would witness performance improvement for free. Similarly, the virtual network operator could decide to implement a new firewall policy in the virtual network or change the forwarding behavior. The old fabric would work seamlessly with this new virtual program without intervention by the physical network operator. In principle, our compiler could even be used repeatedly to virtualize virtual networks.

7. Related Work

A large body of work has explored the design of high-level languages for SDN programming [8, 19, 24, 25, 28, 29, 33]. Our work is unique in its focus on the task of engineering efficient compilers that scale up to large topologies as well as expressive global and virtual programs.

An early paper by Monsanto et al. proposed the NetCore language and presented an algorithm for compiling programs based on forwarding tables [24]. Subsequent work by Guha *et al*. developed a verified implementation of NetCore in the Coq proof as-

sistant [12]. Anderson et al. developed NetKAT as an extension to NetCore and proposed a compilation algorithm based on manipulating nested conditionals, which are essentially equivalent to forwarding tables. The correctness of the algorithm was justified using NetKAT's equational axioms, but didn't handle global programs or Kleene star. Concurrent NetCore [30] grows NetCore with features that target next-generation SDN-switches. The original Pyretic paper implemented an "reactive microflow interpreter" and not a compiler [25]. However later work developed a compiler in the style of NetCore. SDX uses Pyretic to program Internet exchange points [13]. CoVisor develops incremental algorithms for maintaining forwarding table in the presence of changes to programs composed using NetCore-like operators [15]. Recent work by Jose *et al*. developed a compiler based on integer linear programming for next-generation switches, each with multiple, programmable forwarding tables [16].

A number of papers in the systems community have proposed mechanisms for implementing virtual network programs. An early workshop paper by Casado proposed the idea of network virtualization and sketched an implementation strategy based on a hypervisor [7]. Our virtual compiler extends this basic strategy by introducing a generalized notion of a fabric, developing concrete algorithms for computing and selecting fabrics, and showing how to compose fabrics with virtual programs in the context of a high-level language. Subsequent work by Koponen et al. described VMware's NVP platform, which implements hypervisor-based virtualization in multi-tenant datacenters [19]. Pyretic [25], CoVisor [15], and OpenVirteX [3] all support virtualization—the latter at three different levels of abstraction: topology, address, and control application. However, none of these papers present a complete description of al-

gorithms for computing the forwarding state needed to implement virtual networks.

The FDDs used in our local compiler as well as our algorithms for constructing NetKAT automata are inspired by Pous's work on symbolic KAT automata [27] and work by some of the authors on a verification tool for NetKAT [11]. The key differences between this work and ours is that they focus on verification of programs whereas we develop compilation algorithms. BDDs have been used for verification for several decades [1, 6]. In the context of networks, BDDs and BDD-like structures have been used to optimize access control policies [21], TCAMs [22], and to verify [17] data plane configurations, but our work is the first to use BDDs to compile network programs.

8. Conclusion

This paper describes the first complete compiler for the NetKAT language. It presents a suite of tools that leverage BDDs, graph algorithms, and symbolic automata to efficiently compile programs in the NetKAT language down to compact forwarding tables for SDN switches. In the future, we plan to investigate whether richer constructs such as stateful and probabilistic programs can be implemented using our techniques, how classic algorithms from the automata theory literature can be adapted to optimize global programs, how incremental algorithms can be incorporated into our compiler, and how the compiler can assist in performing graceful dynamic updates to network state.

Acknowledgments. The authors wish to thank the anonymous ICFP '15 reviewers, Dexter Kozen, Shriram Krishnamurthi, Konstantinos Mamouras, Mark Reitblatt, Alexandra Silva, and members of the Cornell PLDG and DIKU COPLAS seminars for insightful comments and helpful suggestions. We also wish to thank the developers of GNU Parallel [31] for developing tools used in our experiments. Our work is supported by the National Science Foundation under grants CNS-1111698, CNS-1413972, CNS-1413985, CCF-1408745, CCF-1422046, and CCF-1253165; the Office of Naval Research under grants N00014-12-1-0757 and N00014-15-1-2177; and a gift from Fujitsu Labs.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*,

27(6):509–516, June 1978.

[2] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, 2008.

[3] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN*, 2014.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.

[5] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[7] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.

[8] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.

[9] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An api for application control of sdns. In *SIGCOMM*, 2013.

[10] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[11] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *POPL*, 2015.

[12] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, 2013.

[13] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. In *SIGCOMM*, 2014.

[14] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[15] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.

[16] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[18] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[19] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Jesse Gross Igor Ganichev, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, , Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[20] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[21] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. XEngine: A fast

and scalable XACML policy evaluation engine. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[22] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *TON*, 18(2):490–500, April 2010.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[24] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.

[25] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, 2013.

[26] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[27] Damien Pous. Symbolic algorithms for language equivalence and Kleene Algebra with Tests. In *POPL*, 2015.

[28] ONOS Project. Intent framework, November 2014. Available at `http://onos.wpengine.com/wp-content/uploads/2014/11/ONOS-Intent-Framework.pdf`.

[29] Open Daylight Project. Group policy, January 2014. Available at `https://wiki.opendaylight.org/view/Group_Policy:Main`.

[30] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *ICFP*, 2014.

[31] O. Tange. GNU parallel - the command-line power tool. *;login: The*

USENIX Magazine, 36(1):42–47, Feb 2011.

[32] David E. Taylor and Jonathan S. Turner. ClassBench: A packet classification benchmark. *TON*, 15:499–511, June 2007.

[33] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.