Magellan: Automatic Programming of Multi-Table Datapath from Datapath-Oblivious SDN Programs

Andreas Voellmy
Yale University

Y. Richard Yang
Yale University

Abstract

Despite the emergence of multi-table pipelining as a key feature of next-generation SDN data-path models, there is no existing work that addresses the substantial programming challenge of utilizing multi-tables automatically. In this paper, we present Magellan, the first system that addresses the aforementioned challenge. Introducing a set of novel algorithms based on static analysis, actual explorative execution, and incremental computing, Magellan achieves automatic derivation, population, and update of effective multi-table pipelines from a datapath-oblivious, high-level SDN program written in a general-purpose language. We implement a complete prototype of Magellan and apply it to implement practical SDN policies. We show that Magellan generates compact tables comparable with those by experts. Comparing the flow tables generated by Magellan with those produced from standard SDN controllers including OpenDaylight and Floodlight, we show that Magellan uses between 46-68x fewer rules. Using incremental updates, we show Magellan completes all-to-all pings after a link failure 87x faster than full recompilation.

1 Introduction

Multi-table pipelining has emerged as the foundation of the next generation SDN datapath models, such as recent versions of OpenFlow [7], RMT [3], and Flexpipe [18]. Avoiding key issues such as unnecessary combinatorial explosions to substantially reduce datapath table sizes, multi-table pipelining is essential for making SDN practical. At the same time, the introduction of multi-tables also adds additional SDN programming tasks including designing effective layout of pipelines, populating the content of multiple tables, and updating multiple tables consistently when there are changes. These tasks add substantial burdens for SDN programmers, leading to lower programming productivity. Automating these tasks can substantially simplify SDN programming.

In this paper, we investigate how to automatically derive, populate, and update effective multi-table pipelines from datapath-oblivious algorithmic policies (AP) [21]. We choose the algorithmic policies model because it is highly general and flexible; hence it poses minimal constraints on SDN programming. On the other hand, effectively utilizing multi-table pipelines from algorithmic policies can be extremely challenging, because APs are expressed in a general-purpose programming language with arbitrary complex control structures (*e.g.*, conditional statements, loops), and the control structures of APs can be completely oblivious to the existence of multi-tables. Hence, it is not clear at all whether one can effectively program multi-table pipelines from such APs. We refer to this as the *oblivious multi-table programming challenge*. In the original paper [21] which proposed the algorithmic policies model, the authors use an approach called trace trees to generate flow table rules. Their approach, however, is limited to a single table setting, missing the substantial benefits of multi-tables. Although there is previous work on how to use multi-table datapath (*e.g.*, [2, 20]), the setting is that the tables and their pipelining are already given.

The main contribution of this paper is the development of Magellan, the first system that addresses the oblivious multi-table programming challenge. The core of Magellan consists of two novel components: the table-design algorithm, and the table-populate algorithm. The table-design algorithm conducts a novel static analysis of an AP, by considering the specific computational capabilities of flow tables (*e.g.*, only matching, availability of metadata as registers), to decompose the instructions in the AP into subcomputations and to generate a novel graph data structure called a Magellan table graph. Each node in the Magellan table graph contains the block of instructions whose behavior will be emulated by a flow table, and the program flow of control among the subcomputations provides the flow table pipeline structure. Taking the Magellan table graph and system state variables as input, the table-populate algorithm conducts a novel program exploration for each node in the Magellan table graph to generate the content of its flow table. Utilizing the saved intermediate results of the exploration algorithm in a data structure called the explorer graph, Magellan can also achieve incremental computation (*i.e.*, re-mapping or re-exploration), to achieve correct, highly efficient flow table updates after state changes.

We implement a complete prototype of Magellan and apply it to program complex, practical policies. For the practical Group-Based Policies [8], we show that Magellan generates compact tables comparable with those by experts. Comparing the flow tables generated by Magel-

lan with those implemented by standard SDN controllers, for the layer 2 learning and routing benchmark, we show that Magellan uses between 46-68x fewer rules than systems including OpenDaylight and Floodlight, since none of them used multi-tables. Demonstrating the benefits of incremental updates, we show that Magellan completes all-to-all pings after a link failure 87x faster than that of full recompilation approaches such as Pyretic.

We emphasize that despite the substantial progress made by Magellan, there are limitations on what Magellan can achieve. In particular, there are algorithmic policies that are simple to express in a general-purpose language but fundamentally impossible to be expressed *compactly* using flow tables.

The paper is organized as follows. We first illustrate the problems in Section 2. In Section 3, we give a design overview. Sections 4, 5, and 6 give details on three key components of Magellan. We evaluate Magellan in Section 7 and compare with related work in Section 8.

2 Motivation

We start with a simple, but representative example AP called `L2-Route` to illustrate the basic challenges and ideas. The AP performs routing using layer 2 addresses:

```
// Program: L2-Route
 1. Map macTable(key: macAddress, value: sw)

 2. onPacket(p):
 3.   s = p.macSrc
 4.   srcSw = macTable[s]
 5.   d = p.macDst
 6.   dstSw = macTable[d]
 7.   if (srcSw != null && dstSw != null):
 8.     egress = myRouteAlg(srcSw, dstSw)
 9.   else
10.     egress = drop
```

In this example and throughout this paper, we use the following AP abstraction: each packet p, upon entering the network at an ingress point, will be delivered to a user-defined callback function named `onPacket`, also referred to as the function f. This function sets the egress variable to be the path that the packet should take across the network. We refer to this style of returning the whole path as the global policy. A variation on this programming model is to define a local, per-switch `onPacket` function. The results will be similar.

Although `L2-Route` looks simple, it includes key components of a useful algorithmic policy: maintaining a system state variable, and processing each packet according to its attributes and the current state. Specifically, line 1 of `L2-Route` declares its state variable `macTable`: a key-value map data structure that associates each *known* L2 endpoint to its attachment switch. Given a fixed packet, `L2-Route` performs a lookup, using the `macTable` state variable, of the source and destination switches for the packet, and then computes a route between the two switches through the network.

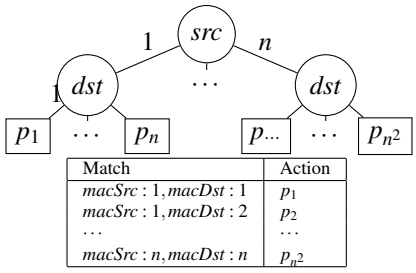

Figure 1: Trace tree and flow table for `L2-Route`.

Result of current tool: The only current work that handles general algorithmic policies is Maple [21], which uses a trace tree approach: a policy is repeatedly invoked within a tracing runtime system that records the sequence of packet attributes read by each invocation, and the recorded execution traces form a trace tree; a trace tree can be compiled to a single flow table, where each leaf of the tree corresponds to a rule in the flow table. Figure 1 shows the resulting trace tree and the flow table required for `L2-Route` to support n hosts with MAC addresses $1 \ldots n$ communicating with each other. For example, the bottom left result p_1 is the execution trace of a packet with *macSrc* 1 and *macDst* 1.

Despite its simplicity, this example illustrates well the issues of the trace tree approach. First, assume the program sees packets between each pair of endhosts stored in `macTable`. Then the trace tree has n^2 leaves, generating a flow table with n^2 rules. This, however, as we show below, is much larger than necessary. Second, even worse, assume a setting where packets with source or destination MAC not stored in `macTable` can appear (*e.g.*, due to attacks). Then, the trace tree approach will still generate flow table rules for such packets. In a worst case where a large number of such packets appear, the trace tree approach may generate well above n^2 rules–in the limit, the trace tree can generate 2^{96} rules, resulting in a not resilient system.

Suboptimal manual table design by experts: Since there are no existing tools to automatically generate multi-tables, we asked several experienced network professionals with significant SDN knowledge to design tables for `L2-Route`. We allowed experts to take advantage of *datapath registers* (aka *metadata fields*) that can be used to store state across tables, and which are available in several dataplane models, including OpenFlow and P4. We use notation reg_x to denote a register holding values for program variable x.

We found that most experts chose a two-table design, as shown in Figure 2, reasoning that the program performs two classifications, one on *macSrc* and the other on *macDst* and hence two table lookups suffice. The first table matches on *macSrc* to write an appropriate *srcSw* value into reg_{srcSw}. The second table also matches on the outcome of the first table (held in reg_{srcSw}) since this at-

Table 1

Match	Action
$macSrc = a_1$	$reg_{srcSw} = y_1$, Jump 2
...	...
$macSrc = a_n$	$reg_{srcSw} = y_n$, Jump 2
otherwise	$reg_{srcSw} = null$, Jump 2

Table 2

Match	Action
$reg_{srcSw} = y_1, macDst = a_1$	$output = o_{1,1}$
...	...
$reg_{srcSw} = y_1, macDst = a_n$	$output = o_{1,n}$
$reg_{srcSw} = y_2, macDst = a_1$	$output = o_{2,1}$
...	...
$reg_{srcSw} = y_k, macDst = a_n$	$output = o_{k,n}$

Figure 2: Two-table design common by human experts.

Table 1

Match	Action
macSrc: a_1	reg_1=y_1, goto 2
...	...
macSrc: a_n	reg_1=y_n, goto 2
otherwise	reg_1=*null*, drop

Table 2

Match	Action
macDst: a_1	reg_2=z_1, goto 3
...	...
macDst: a_n	reg_2=z_n, goto 3
otherwise	reg_2=*null*, drop

Table 3

Match	Action
reg_1: y_1, reg_2: z_1	$p_{1,1}$
...	...
reg_1: y_n, reg_2: z_n	$p_{n,n}$
...	...
reg_1: *null*, reg_2: *null*	drop

Figure 3: The optimized 3 table pipeline for L2-Route.

tribute also affects the desired outcome. This design successfully avoids the n^2 cross product problem, since the number of reg_{srcSw} values is typically much lower than the number of host interfaces in the network.

While this design improves over the single table design generated by trace trees, it is suboptimal for most networks that have many more hosts than switches. In particular, the three table design shown in Figure 3, which has a final table that matches on combinations of switches, typically requires far fewer rules. Specifically, if n is the number of hosts in the network and k the number of switches output by the *macTable* mapping, then the two-table design requires $n+kn$ rules, while the three table design requires $2n+k^2$ rules. For a network with 4,000 hosts and 100 switches, the two-table design requires $404K$ rules while the three-table design requires $18K$ rules, a 22x difference.

We therefore see that selecting good pipeline designs requires considering details such as the flow of data values through the given program, which are difficult and tedious for humans to consider and easily overlooked.

Burden of populating tables with rules: In addition to designing a pipeline, a human expert is required to define how tables are populated with rules at runtime, which can be a complex task. Consider for example, how to generate new rules for the two-table design when a single, new entry (a', s') is inserted into *macTable*. If a' is a new key and s' is a value not previously occurring in the table, then Table 1 requires a new entry $macSrc : a' \rightarrow reg_{srcSw} : s'$ and Table 2 requires new entries of the form $reg_{srcSw} : s', macDst : a \rightarrow output : o_{a,s'}$ for every key a of *macTable*. This illustrates that a single change to a high-level state may require changes in multiple flow tables.

Moreover, if L2-Route is modified in a minor way, the situation becomes more challenging:

```
2.  onPacket(p):
3.    s = p.macSrc
4.    srcSw = macTable[s]
4a.   if srcSw member [1,2,3,4]:
4b.      egress = drop; return
5.    d = p.macDst
6.    dstSw = macTable[d]
```

In this version of L2-Route, the program drops packets from switches 1 through 4. In this case, it is unnecessary to continue processing packets from switches 1-4. In the two-table design, Table 2 need not match on values 1-4 for the reg_{srcSw} field, which could lead to substantial saving of space when the number of hosts is large. Taking advantage of this in populating entries for Table 2 therefore requires reasoning about the flow of possible values to Table 2, which is a burden for programmers.

Burden of target-specific programming: In addition to the conceptual design of the forwarding pipeline and the runtime processes to populate the pipelines' rules, a programmer is faced with the substantial burden of encoding these designs into target-specific forwarding models. For example, when targeting Open vSwitch, a programmer may use the Nicira-extension registers to implement the datapath registers and populate entries using an OpenFlow protocol. On the other hand, when implementing the design with P4, the programmer would need to declare metadata structures and fields and would need to use a target-specific proprietary runtime protocol to populate rules. Since there is no existing portability layer that spans various OpenFlow and P4 switches, the high-level design and runtime algorithms will need to be coded multiple times for each supported target, leading to duplicated effort and increased likelihood of bugs.

3 Design Overview

The high-level objective of Magellan is to simple to state: alleviate the aforementioned burdens of table design, table population, and target-dependent programming, so that SDN programmers can focus on high-level, general-purpose, and target-independent programming.

There are multiple design decisions when realizing the objective of Magellan. One design decision is whether to design complete compilation, similar to a setting that a modern compiler compiles a high level language completely to a target machine code; the compiler no longer needs to be available after compilation. This, however, is not possible in our settings, revealing fundamental differences between traditional compilation and Magellan compilation. Specifically, one difference between our setting and traditional compilers is that machine code can realize state update (*i.e.*, modifying memory storing the state), but datapath flow tables cannot update states,

except for simple state variables such as some counters. Hence, Magellan must (1) encode flow tables such that datapath packets that trigger state changes (*e.g.*, an AP that checks if a packet containing a new MAC, and if so, adds it to a system state variable) should be forwarded back to the controller; and (2) Magellan must have an online runtime to carry out the update. Even beyond that, for APs that will not update system state, their computations may not be mapped practically. We have

Proposition 1 *Consider* `return p.macDst % n`*, where* $n \neq 2^i$ *for any i. There is no compact flow table representation of this statement, where a compact flow table is one who has far fewer than* 2^{48} *rules.*

This result is not totally surprising, because flow tables in SDN provide a fundamentally limited computational model. Although one possibility to handle such cases is to restrict the language, Magellan makes the design decision of allowing general programming. Hence, although Magellan compiler will warn a programmer about such cases, it still allows the programmer to proceed if chosen by the programmer. Magellan generates flow tables with conditions to detect cases when such computations happen and send them back to the controller for execution, achieving a complete design.

Another key design decision is whether to conduct table design for specific system state. Note that the content of flow tables will depend on system state (see examples in Section 2). The benefit of this approach is that the table designed is optimized for the system state. An issue of this approach, however, is that the system state of a network can change, due to policy state (*e.g.*, ACL rules) or network status state (*e.g.*, link status) changes. Many programmable forwarding devices, specifically those implementing P4, require a possibly time-consuming configuration step to specify table attributes (though not their contents) prior to operating on input traffic, prohibiting frequent runtime reconfiguration of the forwarding pipeline. Hence Magellan relies on a combination of static and dynamic analysis: static analysis based on program structure and stable system state statistics to design multi-table pipeline, and dynamic, runtime analysis based on exact system state to populate rule tables.

In summary, Magellan introduces four key components to efficiently implement algorithmic policies on both OpenFlow and P4 targets:

- The first is the Analyzer, which converts programs written in a friendly, user-oriented language into a language- and target-independent Intermediate Representation (IR) of simple instructions and analyzes these programs to produce information, such as control- and data-flow graphs and variable liveness.
- The second is the Table Designer, which uses the analyses produced by the Analyzer to produce a forwarding pipeline design consisting of table definitions, metadata information, and control flow among tables.
- The third is the Explorer, the key runtime component that populates the flow tables designed by Table Designer based on actual runtime values, such as the particular values of program state variables.
- The fourth is the Controller, which handles difficult-for-flow-table to handle instructions as well as interacts with specific target switches using their target-specific protocol and which translates the target-independent flow tables produced by Explorer into target-specific rules and commands.

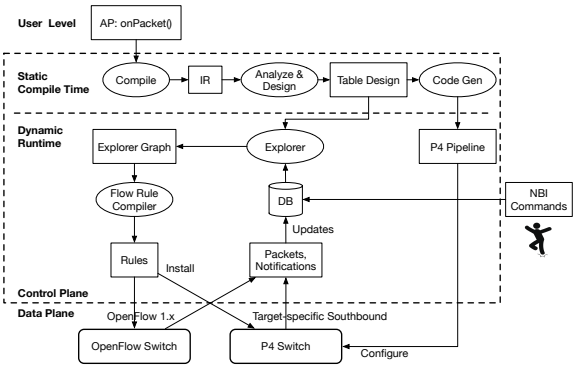

Figure 4: Magellan major components and workflow.

Figure 4 shows main Magellan components and the basic workflow targeting both OpenFlow and P4 switches. The rest of this paper focuses on the Analyzer, Table Designer and Explorer. We omit discussions of the relatively straightforward Controller component.

4 Analyzer: Algorithmic Policy to Intermediate Representation (IR)

Algorithmic policy (AP): Since different languages and different systems using the same language may impose different syntax to specify an AP, we do not give a formal specification of AP. Instead, we use a pseudo-code style to specify APs in this paper. Different from low-level datapath computation models such as Openflow and P4, which typically allow only simple numerical types and limited computation flow control, a good language for APs should allow generic, compound data types (*e.g.*, sets, hash maps) and complex computation flow control such as conditionals and loops.

To appreciate the flexibility, generality, and complexity of APs, consider an AP for *Group-based Policy* (GBP) [8], a substantial framework for specifying application policies in data centers in both OpenDaylight [16] and OpenStack [17]. GBP policy[1] consists of a collection of *endpoint groups* (group). Each endpoint (aka network host) is assigned to one group, and each group

[1]For brevity, we have simplified and removed some GBP details, such as optimized handling of ARP, extended action set, etc.

participates in one or more *contracts*. Given a packet p, GBP checks if there exists a contract that is common to the packet sender's group's contracts and to the packet receiver's group's contracts. A matching contract consists of an ordered sequence of clauses. Whether a clause is enabled depends on conditions on packet source and packet destination, where the condition of an endpoint is a set of labels (*e.g.*, `virus_detected`, `authenticated`, `os_windows`). An enabled clause leads to a set of rules, where each rule consists of both a packet classifier and a set of actions (*e.g.*, permit).

Despite the preceding description complexity, implementing GBP using an AP is relatively straightforward. Figure 5 shows an AP, focusing on the case that GBP checks permit, and including code to do L2 learning and to compute the out port of the packet.

```
Map macTable(key:macAddress, value:port)
Map mac2Cond(key:macAddress, value:StringSet)
Map mac2Group(key:macAddress, value:int)
Map group2Contracts(key:int, value:[int])
...

def gbp(p):
 1. srcCond = mac2Cond[p.macSrc]
 2. dstCond = mac2Cond[p.macDst]
 3. srcGrp  = mac2Group[p.macSrc]
 4. dstGrp  = mac2Group[p.macDst]
 5. if (srcGrp == null || dstGrp == null):
 6.   return false
 7. sctrcts = contracts[srcGrp]
 8. dctrcts = contracts[dstGrp]
 9. if (sctrcts == null || dctrcts == null):
10.   return false
11. for (scontract : sctrcts):
12.   for (dcontract : dctrcts):
13.     if (scontract == dcontract):
14.       // check clauses conditions ...
15.  return false

def onPacket(p):
 1. macTable[p.macSrc] = p.ingressPort
 2. permit = gbp(p)
 3. if (permit == false): egress=drop; return
 4. d = macTable[p.macDst]
...
```

Figure 5: GBP AP.

One can see that the GBP AP uses complex data structures as state variables. For example, *mac2Cond* maps a MAC address representing an endpoint host to a set of string labels representing its conditions. On the other hand, *mac2Group* is a relatively simpler state which maps a MAC address to the ID of a endpoint group that the endpoint belongs to. The GBP AP also uses complex control structures including conditionals at lines 5, 9, 13 of the `gbp` function and for loops at lines 11,12. Note that different SDN programmers may design different data structures and different control structures. Some part of the program may be written inefficiently. What we show is just one possible input to Magellan.

AP Intermediate Representation (IR): To avoid excessive source language dependence and to simplify the design of the system, Magellan maps an input AP into a simple Intermediate Representation (IR) instruction set, shown in Figure 6. The instructions in Magellan IR should be relatively easy to understand. To help with the understanding, we show below the IR for a segment (lines 1-5) of the `gbp` function in Figure 5:

```
⟨instruction⟩ ::= ⟨ident⟩ '=' ⟨expr⟩
                | 'label' ⟨label⟩
                | 'goto' ⟨label⟩
                | 'cond' ⟨booleanexpr⟩ ⟨label⟩ ⟨label⟩
                | 'return' ⟨expr⟩

⟨expr⟩        ::= 'arith' ⟨arithexpr⟩
                | 'boolean' ⟨booleanexpr⟩
                | 'call' ⟨function-name⟩ ⟨expr-list⟩
                | 'readpktfield' ⟨fieldexpr⟩
                | 'lookup' ⟨mapident⟩ ⟨expr-key⟩
                | 'members' ⟨mapident⟩ ⟨expr-list⟩
                | 'update' ⟨mapident⟩ ⟨expr-key⟩ ⟨expr-val⟩
                | ⟨etc⟩
```

Figure 6: Magellan intermediate representation (IR).

```
1. v1 = readpktfield macSrc
2. srcCond = lookup mac2Cond v1
3. v2 = readpktfield macDst
4. dstCond = lookup mac2Cond v2
5. srcGrp = lookup mac2Group v1
6. dstGrp = lookup mac2Group v2
7. cond srcGrp == null || dstGrp == null 8 9
8.  ...
```

5 Table Design

We start with the basic insights of table design. Consider each IR instruction I. It is a function that reads a subset of program variables and assigns new values to a subset of variables. We refer to the subset of program variables whose values are read by the instruction as its *input variables*, denoted *inputs*(I), and the subset of variables assigned to by it as its *output variables*, denoted *outputs*(I). A key insight of our table design is that there are two ways to realize the function: (1) execute the instruction; or (2) implement it as a flow table that matches on the input variables and put the outputs in the output registers. The `update` instruction adds some complexity, which we will discuss in Section 5.5.

One flow table per instruction may not be feasible. Hence, we need to consider a block of instructions. When considering execution of an instruction in a block, instruction flow of control will happen. Also, to understand a block of instructions, it helps to also keep track of variables who may not be used by one instruction but may be used by following instructions. Hence, we model the semantics of each instruction I as a *transition function*, $\bar{I} : Store \rightarrow (PC, Store)$, where *Store* is a set of potential execution states, and each element $store \in Store$ consists of an assignment $\{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}$ of values $a_1, \ldots, a_n$ to program variables $v_1, \ldots, v_n$ and PC enumerates the set of instructions occurring in the program. Packet attributes are among the variables $v_1, \ldots v_n$.

Table 1

Match	Action
$macSrc = a_1$	$reg_s = a_1$, Jump 2
...	...
$macSrc = a_n$	$reg_s = a_n$, Jump 2

Table 2

Match	Action
$reg_s = a_1$	$reg_{srcSw} = y_1$, Jump 3
...	...
$reg_s = a_n$	$reg_{srcSw} = y_n$, Jump 3
otherwise	$reg_{srcSw} = null$, Jump 3

Table 3

Match	Action
$macDst = a_1$	$reg_d = a_1$, Jump 4
...	...
$macDst = a_n$	$reg_d = a_n$, Jump 4

Table 4

Match	Action
$reg_d = a_1$	$reg_{dstSw} = y_1$, Jump 5
...	...
$reg_d = a_n$	$reg_{dstSw} = y_n$, Jump 5
otherwise	$reg_{dstSw} = null$, Jump 5

Figure 7: Instruction-level memoization

Given this semantic instruction model, we can develop a first simple approach to modeling subcomputations by modeling each instruction with a flow table that records the input-output semantics of the instruction. Specifically, each rule in the table for I will simulate the effect of executing I on a variable binding *store*: if $\overline{I}(store) = (pc, store')$ then we can use a rule that matches the input variables to I against the values of those variables in *store* and which has an action that first writes to the variables changed by I to the values they take in $store'$ and then jumps to the table for pc. To determine the set of *store* values that I may execute on, we can apply a reactive controller that observes the values of input variables at each table through a packet punt mechanism.

Figure 7 illustrates the instruction-level memoization approach applied to lines 3-6 of `L2-Route`. For example, table 2 models statement `L4` by matching on the register for s and setting the register for *srcSw*. While the approach succeeds in avoiding the n^2 cross-product problem of single flow table implementations, it introduces several problems. In particular, it requires an excessive number of tables for realistic programs, since programmable, multi-table switch chips in the near future are likely to have tens of flow tables while realistic programs will likely have 100s to 1000s of lines of code. Moreover, each instruction's table may require a large number of rules, since this approach uses an exact match against variable values and there is one rule per input store that is observed to occur at the instruction. In this example, tables 1 and 2 will have one rule per MAC address occurring in the input traffic.

5.1 Compact-Mappable Statements

Fortunately, we can improve on the naive, exact match approach described in the previous section by observing that many instructions can have compact encodings using ternary, prioritized matches available in flow tables that require far fewer rules than the preceding generic, naive input-output mapping. For example, the statement `L4: srcSw = macTable[s]` can be encoded with exactly $m+1$ rules, where m is the number of entries in `macTable`: there is one rule for each key in `macTable` and a final rule for the case where s is not found in `macTable`. Similarly a statement such as `x = srcSw > 4` has a compact encoding: if `srcSw` is represented with k bits, then if any of the $k-2$ high order bits are set, x is assigned true, else false. Hence we need $k-1$ rules to implement the instruction on the bits of *macSrc*, whereas a memo table for this instruction would require one rule per observed *macSrc* value.

We therefore identify a large set of *compact-mappable* statements. Each compact-mappable statement is an assignment $v = e$, where we group the expression e into three categories: state variable lookup, boolean expression, and flow-table compatible arithmetic. A lookup expression has the form $t[v_1, \ldots, v_n]$ for some system state table t where at least one of v_i is a packet attribute variable (this can be generalized, but for simplicity we keep this form.) Boolean expressions are a conjunction of non-negated or negated simple conditions, which include $pattr$ `relop` e, $pattr_1 = pattr_2$, where $pattr, pattr_1, pattr_2$ are packet attributes, e is an expression not involving packet attributes, `relop` is one of $<, \leq, =, \geq, >$. Mappable arithmetic expressions include bitmasked fields, *e.g.* expressions of the form $pattr$ & $mask$.

Specifically, `TableMap` (Algorithm 1) maps a subset of compact-mappable state table lookups into flow tables (it can be extended to map other compact-mappable statements similarly). A flow table can simulate the state table lookup by matching the input variables against each of the keys occuring in the table and sets the output according to the value associated with the key (Line 5). However, since some variable-key bindings may not occur at a given lookup instruction (*e.g.* in the statements `x=1; y=nextHop[macDst,x];`, only keys of `nextHop` whose second component is 1 are needed). Therefore, `TableMap` filters the keys against the input *store* (line 4). In the boolean case of testing $v = pattr == e$, we evaluate both the pattern expression and the right-hand side expression to obtain a match condition and then assign *true* to v if the match succeeds. In both state table lookups and compact boolean expressions, we include default rules to handle the cases when the table lookup fails and when a boolean test returns false (lines 6 and 10). We use OpenFlow's priority convention that rules with higher priorities take precedence.

While not all table lookups and boolean expressions in the given input programs are compact-mappable, many of these statements can be transformed into compact-mappable form. Magellan accomplishes this by developing a *packet attribute propagation analysis* (a variation on constant propagation) to compute, for each variable and program point, whether the variable is equal to some

Algorithm 1 TableMap($I, store$)

1: **switch** $(I.type)$ **do**
2: **case** $v = t[v_1, \ldots, v_n]$:
3: **for** $((x_1, \ldots, x_n), y) \in entries(t)$ **do**
4: **if** $(\{v_1 : x_1, \ldots, v_n : x_n\})$ possible in *store* **then**
5: Add rule $prio : 1, v_1 : x_1, \ldots, v_n : x_n \mapsto v : y$
6: Add rule $prio : 0 \mapsto v : null$
7: **case** $v = pattrexp == e$:
8: let $v_1, \ldots, v_n$ be $inputs(I)$
9: Add rule $prio : 1, v_1 : x_1, \ldots, v_n : x_n, eval(pattrexp, store) : eval(e, store) \mapsto v : true$
10: Add rule $prio : 0, v_1 : x_1, \ldots, v_n : x_n \mapsto v : false$
11: **end switch**

packet attribute at the given program point. For example, applying the analysis to `L2-Route` discovers that variable *s* in line `L4` is equivalent to *macSrc* and that *d* at `L6` is equivalent to *macDst* and therefore rewrites these two table lookups to be `srcSw = macTable[macSrc]` and `dstSw =macTable[macDst]`. This transformation therefore allows both lookups in *macTable* in `L2-Route` to be considered as compact-mappable.

5.2 Regions: Instruction Aggregation

Although the program contains a set $C = I_1, \ldots, I_c$ of compact-mappable statements, there will be many other instructions that are not non-compact mappable, whose behavior must also be simulated by the forwarding pipeline. As we observed previously, a naive, instruction-level mapping is impractical.

Fortunately, we can improve over instruction-level mapping by observing that our semantic model of instructions extends to blocks of instructions as well. For example, we can model a sequence of instructions $I_1; \ldots; I_n$ where each has a transition function $\tau_1, \ldots, \tau_n$ with the composed function $\tau_n \ldots \circ \pi_1 \circ \tau_2 \circ \pi_1 \circ \tau_1$, where we use a helper function $\pi_1(store, pc) = store$ that selects the updated store from the tuple returned by τ_i . We can leverage this to apply the preceding memorization technique to entire *regions* of non-compact instructions where each region has some unique starting instruction, denoted $entry(R)$, and extends to include all instructions reachable from $entry(R)$ without traversing any other region headers. We extend our notation of input and output variables to regions. Specifically, $inputs(R)$ is the set of variables that may potentially be used before being written to by instructions in R (similar, but not identical, to the conventional notion of *variable liveness*[2]). By memorizing regions of instructions, we dramatically reduce the number of tables required. We call the regions and the control flow among them, the *magellan table graph*.

[2]Informally, a variable is *live* at a program execution point if its value may later be read before it is written to.

`DefineRegions` (Alg. 2) demonstrates how we form regions. `DefineRegions` begins by considering each compact-mappable statement to the entry of a region (line 1). We then process the compact-mappable statements in topological order relative to the control flow graph of the program (we address programs with loops in Section 5.3). For each compact-mappable I_j, we consider the instructions reachable from I_j without traversing entering another region. If adding a considered instruction to the region for I_j would not add a new input variable to the input set of the region, we include the instruction. Otherwise, we mark the instruction as beginning a new region (line 8). We do not recursively apply this process to the newly discovered region headers, although in general this could be advantageous. To illustrate the algorithm, consider the following AP:

Algorithm 2 `DefineRegions`()

1: $H = C$
2: **for** $i = 1 \ldots c$ **do**
3: **while** true **do**
4: let I be the topologically next considered instruction reachable from I_i without traversing a region header.
5: **if** there does not exist such an I **then**
6: break
7: **if** including I in region R_i adds new variables to $inputs(R_i)$ **then**
8: $H = H + \{I\}$
return H

```
1.  x = t1[macSrc];
2.  y = x * x;
3.  z = t2[macDst];
4.  egress = [y + z];
```

Figure 8 depicts both the control and dataflow dependencies of this program. `DefineRegions` starts with $H = \{1, 3\}$ (the thick circles), since these are compact mappable instructions. It processes $i = 1$ first and examines instruction 2. Since the variables in $inputs(I_2) = \{x\}$ are produced within the region R_1, including I_2 in R_1 does not increase $inputs(R_1)$ and hence we include I_2 in R_1. The algorithm then exits the inner loop and processes region R_2. The algorithm then examines I_4 and since $inputs(I_4) = \{y, z\}$, where the value of y is produced outside of the region R_2, including I_4 in R_2 would increase its input set. Hence, I_4 is marked as an additional region header (*i.e.* added to H). The boxes in Figure 8 indicate the resulting regions computed by our algorithm.

Note that our definition of regions differs from traditional compiler notion of *basic block*. In particular, regions may include instructions that change control flow (unlike basic blocks), may be smaller or larger than basic blocks, and allow overlapping regions.

5.3 Loops

Many important APs contain loops, for example to perform some complex search (*e.g.* ACLs, contracts). Since

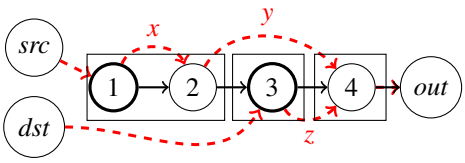

Figure 8: Control and data flow graphs for example program. Each instruction is labeled with its line number; other nodes are program inputs or outputs. Control flow is depicted with solid black edges while data flow is depicted with dashed (red) edges.

forwarding models (*e.g.* OpenFlow and P4) do not allow cyclic control flow among flow tables, we cannot naively apply our prior techniques to programs with loops.

Fortunately, our region abstraction provides a simple approach to supporting such programs. Specifically, we consider each loop that contains one or more compact-mappable statements to be a single region, whose entry is the loop entry instruction. Note that loops that do not contain compact-mappable statements are handled by the prior techniques with no modification, as they will be incorporated into the above-designed regions.

Magellan therefore first computes the strongly connected components of the control flow graph and forms a graph $G = (V, E)$ where the nodes V consist of the strongly connected components (*i.e.* sets of instructions) and where $(u, v) \in E$ is an edge whenever an instruction in u has an edge in the original control flow graph to an instruction in v. We then compute regions using a `DefineRegions` where each strongly connected component is considered as a pseudo-instruction. Since G is acyclic and hence can be topologically ordered.

5.4 Register Allocation & Value Encoding

The number of datapath registers (or packet metadata) is limited in real switching chips and it is important to use few datapath registers. To address this, Magellan adapts a register allocation algorithm from the compiler literature, applying it to a *variable interference graph* based on the input and output variables of the computed regions.

In addition, most forwarding models, including OpenFlow and P4, do not provided compound data types, such as collections. In order to tabulate regions whose input or output sets include variables that hold collections, Magellan implements *runtime value encoding*. In particular, at runtime, each unique collection that must be stored in a given variable v is assigned a unique numeric identifier. To generate a rule that matches on (or writes to) a variable v with a given collection, Magellan instead matches on (or writes to) the collection's assigned identifier.

5.5 State Updating Instructions

Realistic APs often include state-update instructions. In particular, even the most basic SDN programming example, namely MAC learning switch, performs state updates in order to implement correct behavior:

```
1. onPacket(p):
2.   macTable[p.macSrc] = p.ingressPort
3.   p = macTable[p.macDst]
4.   if (p == null) { egress=[allPorts]}
5.   else { egress = [p] }
```

In this example, the program begins in line 2 by updating the mapping of mac addresses to ports. The program then continues to use this same `macTable` in line 3 to look up where the destination of the packet resides.

To handle state updates, we extend the computational model to describe the updates performed by an instruction. In particular, we consider the transition function for any instruction I to be $\bar{I} : Store \rightarrow (PC, Store, Updates)$, where *Updates* denotes the set of sequences of system state variable updates. For example, if I is line 3 of `L2Learn`, then $\bar{I}(\{macSrc : 1, inPort : 2\}) = (\{macSrc : 1, inPort : 2\}, 4, [insert(macTable, [1], 2)]$.

To implement state updates, Magellan modifies a rule that updates state to instead punt the packet to the controller, effectively halting the execution in the pipeline. The controller receives the punted packet, executes it using the given AP, and performs any state updates needed. These updates may lead to updated flow tables so that these packets are not punted again. If asynchronous updates are permitted, Magellan can continue processing the packet, in addition to sending punted packet, allowing the packet to be processed in the current forwarding state, while the controller updates state asynchronously.

6 Runtime: Proactive Table Population

While the table designer determines the forwarding pipeline organization, the Magellan runtime system populates and maintains designed flow tables at runtime. The `TableMap` algorithm in Section 5 described how we can map a compact-mappable instruction into a flow table given the collection of all *store*s which can reach the instruction. While a reactive approach to determining these sets is possible, this can substantially damage performance by causing many packet misses. Therefore, Magellan develops a novel exploration technique that computes reachable sets (and flow tables) proactively in most cases. We also extend table mapping to map entire regions, including those with loops, to flow tables.

6.1 Region Exploration

Given a fixed, current value of each system state table, an AP is a function with finite inputs and outputs. In principle, we can simply execute the program on all possible input values while instrumenting the execution to observe the stores that can reach each instruction. Unfortunately, the input space is too large, even if we restrict to consider only packet fields used in the program, to make this approach feasible.

Fortunately, we can exploit compact-mappable statements to practically explore all executions of the program

without enumerating all inputs. The key observation is that each compact-mappable statement maps some input packet attributes with large ranges (*e.g. macSrc*) into a small collection of possible outcomes. For boolean compact-mappable instructions, two outcomes are possible, while for system state table lookups, the number of outcomes is the $m+1$ where m is the number of values in the key-value state table. We consider m to be *small*, because it is bounded by the size of the system state, rather than the cardinality of packet header space. We say compact-mappable statements have *low fan-out*.

We can exploit these low fan-out instructions to systematically explore the program by repeatedly executing instructions, starting at the program entry point with an empty store and continuing as follows: if we are executing an instruction whose inputs are derived variables (*e.g.* not input packet attributes), simply compute the result, update the store and then continue executing at the next instruction. If we are executing a low fan-out instruction, then, for each of the outcomes, we calculate a new store updated with that outcome and then continue exploration at the next instruction with that updated store. Otherwise, we are executing a high fan-out instruction (*e.g.* read the *macSrc* attribute) and we retrieve a set of sampled values that have occurred for this instruction in the past, with which to continue executing. In this way, we obtain the set of stores that can reach each instruction.

`ExploreRegions` (Alg. 6; see appendix) applies this exploration idea to each region produced by the Table Designer. It explores each region in topological order, propagating reachability information to subsequent regions as they are reached. For efficiency, each region R is explored once for every distinct set of variable bindings for the variables in $inputs(R)$.

`ExploreRegion` (Alg. 3) explores an individual region R. When the exploration reaches $entry(R')$ of a new region R', it updates the stores reaching R'. Specifically, it extends the current *store* in all possible ways with bindings for variables that are not in $inputs(R)$ but which are present in stores reaching R, and then removes non-live variables at $entry(R')$ (lines 16). In addition, for each region, it generates a *Explorer Graph (EGraph)* which records the packet access operations performed for executions in the region in the form of a directed, acyclic graph (DAG). *EGraph* is conceptually similar to a trace tree [21], in that it records packet accesses to enable flow table compilation. Unlike trace trees, *EGraph* allows sharing of sub-trees, which can occur whenever distinct executions reach the same store and program counter; in this case, subsequent program behaviors are identical and can therefore by represented by a single graph node.

In particular, `ExploreRegion` (Alg. 3) constructs an *EGraph* by identifying nodes by $(pc, store)$ pairs and by recording the outgoing edges from a node in that node's $outEdges[]$ array. An *EGraph* has 5 types of nodes: (1) return nodes, that record the store at a return instruction, (2) map nodes, for compact-mappable instructions, (3) region jump nodes, which indicate that execution jumps to a distinct region, (4) sampling nodes, which correspond to non-compact mappable statements that access some packet field, and (5) execution nodes, which correspond to the execution of non-compact mappable, non-packet access instructions. The algorithm repeatedly executes up to the next break point, where a break point is either a return instruction, a compact-mappable instruction, the first instruction of a distinct region, or an non-compact access to a packet attribute.

Algorithm 3 `ExploreRegion` $(R, pc, store, others)$:

1: $ins = prog[PC]$, $nid = (PC, store)$
2: **if** *nid* explored already **then return**
3: mark *nid* explored
4: **switch** (instruction type) **do**
5: **case** `Return`:
6: add $ReturnNode(nid, restrict(store, outputs))$
7: **case** `Map` of the form $x = e$ with next pc pc':
8: add $node = MapNode(nid, store)$
9: **for** each outcome of y of e possible in *store* **do**
10: $store' = restrict(store + \{x : y\}, pc')$
11: $nid' =$ execute at $(pc', store')$ to next break
12: $node.outEdge[y] = nid'$
13: `ExploreRegion`$(R, pc', store', others)$
14: **case** `branch` to different region R':
15: add $RegionJumpNode(nid, R')$
16: $reach[R'].insert(\{restrict(extend(store, o), entry(R')) : o \in others\})$
17: **case** unconstrained access to packet field $x = fld$:
18: add $node = SampleNode(nid, fld)$
19: pc' is the next PC
20: **for** each sampled value y of *fld* reaching *pc* **do**
21: $store' = restrict(store + \{x : y\}, pc')$
22: $nid' =$ execute at $(pc', store')$ to next break
23: $node.outEdge[y] = nid'$
24: EXPLOREREGION$(R, pc', store', others)$
25: **default**:
26: add $node = ExecNode(nid, store)$
27: $nid' =$ execute at $(pc, store)$ to next break
28: let $(pc', store') = nid'$
29: $node.outEdge[] = nid'$
30: EXPLOREREGION$(R, pc', store', others)$
31: **end switch**

6.2 Non-Compact Packet Access

As described above, some instructions may read packet attributes whose range of values may be extremely large (*e.g.* 2^{48}). For such instructions, we rely on a reactive, sampling approach. For each such instruction, Magellan

runtime records the set of values for the packet attribute seen at this program point. To ensure observation, Magellan generates rules for a *SampleNode* of the EGraph in a way that ensures that any packets having values not observed before are punted (*i.e.* have packet miss) to the controller, where the sample database is updated. After this set is updated, further flow table modifications may be triggered which suppress further packet misses for packets with this value of the given attribute.

The Magellan analysis phase performs packet attribute propagation and often removes packet attribute access instructions after rewriting. In practice, we observe few programs with non-compact packet accesses. However, as noted in Section 3, there will always be statements and programs that cannot be compactly mapped to flow tables regardless of how sophisticated our analyses are.

6.3 Region Table Mapping

Given the reachable stores $reach[R]$ and the EGraph G_R for every region R, `RegionMap` (Alg. 4) maps the overall program into flow tables. For each region R, the algorithm determines each of the R-relevant stores, then compiles G_R at each such store (line 3), and then adds the resulting rules to the table for region R (line 4).

Algorithm 4 `RegionMap`(R):

```
1: let pc = entry(R)
2: for s ∈ {restrict(s',pc) : s' ∈ reach[R]} do
3:     rules = CompileGraph(G_R, pc, s)
4:     Table[R].add(rules)
```

`CompileGraph` (Alg. 5) then compiles each G_R into a single table of rules. The algorithm traverses G_R recursively starting with the root, mapping each EGraph-node into a logical flow table using the `TableMap`. If a resulting rule terminates flow processing or if it jumps to another region (line 5), the recursion terminates and the rule is added to the output collection of $rules'$ (line 6). Otherwise, if a resulting rule jumps to another node within G_R (line 7), the algorithm determines the $store'$ that results from performing the action of the rule (line 8), continues to compile the target of the jump (line 9) and then *inlines* the resulting logical flow table into the current rule. This inlining is required because `CompileGraph` must generate a single logical flow table for the entire region. Note that in combining parent and child rules `Inline` eliminates all combinations that have empty intersections and therefore would never be activated. The `Inline` procedure is straightforward and is omitted for space.

7 Evaluations

In this section, we demonstrate that Magellan (a) can match the designs of human SDN experts on a real world example; (b) improves end-to-end performance over existing systems; (c) quickly repairs forwarding rules after

Algorithm 5 `CompileGraph`$(G_R, pc, store)$:

```
1: let rules = TableMap(pc, store)
2: let rules' = ∅
3: for r ∈ rules do
4:     pc' = r.jumpsTo
5:     if (pc' = entry(R'), R ≠ R') ∨ (pc' = null) then
6:         rules'.add(r)
7:     else
8:         store' = restrict(perform(store, r.regWrites), pc')
9:         childRules = CompileGraph(G_R, pc', store')
10:        for r' ∈ Inline(r, childRules) do
11:            rules'.add(r')
12: return rules'
```

change in system state; and (d) scales rule compilation to networks with hundreds of thousands of endpoints. Our evaluations use our prototype, which consists of 9500 lines of Haskell and includes an OpenFlow 1.3 message layer and controller and a P4 code generator and runtime interface to a reference P4 switch target.

7.1 Real World Policy Compilation Quality

We apply Magellan to *Group-based policy* (GBP) [8] We use the AP specified in Section 4 and compare the GBP authors' published multi-table design with automatically derived tables from Magellan.

GBP flow tables: Figure 9 illustrates the key flow tables used by GBP. The first table matches on *macSrc* and (1) writes the source group to register 1 and (2) ensures mac learning by only having entries for learned (host, port) associations and otherwise punting. Table 2 matches on *macDst* and writes the destination group into register 2 and the outgoing port in register 3. Any destinations which are either not part of a group or whose out port is unknown are dropped by the final rule in table 2. Finally, table 3 matches on all combinations of source and destination groups using matches on registers 1 and 2. In addition, each rule matches on transport ports in order to constrain the forwarding behavior to the specific traffic class of GBP rule. If a packet is permitted at this point, it is forwarded to the appropriate port using the value stored in register 3; otherwise it is dropped.

Table 1 (source group)

Match	Action
inPort:p_1,macSrc:a_1	reg_1=sg_1, goto 2
...	...
inPort:p_1,macSrc:a_n	reg_1=sg_n, goto 2
otherwise	punt

Table 2 (dst group and output port)

Match	Action
macDst:a_1	reg_2=dg_1, reg_3=q_1, goto 3
...	...
macDst:a_n	reg_2=dg_n, reg_3=q_n, goto 3
otherwise	drop

Table 3 (unicast output)

Match	Action
...	...
reg_1:sg_i,reg_2:dg_i,tcpSrc:w_i	drop or out:reg_3
reg_1:sg_i,reg_2:dg_i,tcpDst:w_i	drop or out:reg_3
...	...

Figure 9: Flow table design for GBP. The action in table 3 depends on the matching contract (see Section 7.1).

Magellan flow tables The flow tables produced by Mag-

ellan are identical with the following exceptions: (1) table 2 does not write to a register 3 for the output port, (2) table 3 jumps to an extra final table 4 that matches on *macDst* and forwards to a particular port. While Magellan uses one extra table, Magellan achieves linear scaling of rules by hosts and groups as does GBP's design.

7.2 End-to-End Performance Benefit

Magellan proactively generates compact forwarding rule sets and thereby eliminates many flow table cache misses and allows switches to handle more traffic locally.

Control Systems: We compare Magellan with a range of state-of-the-art commercial and academic SDN systems, including OpenDaylight (ODL) (Helium release), Floodlight (version 1.0), POX [19] (`forwarding.l2_learning` module from 0.2.0), Pyretic (latest version), and Maple (version 0.10.0). POX, Pyretic and Maple are academic systems supporting novel policy languages and compilers, while ODL and Floodlight are open source controllers that form the basis of several commercial control systems. We run controllers on a 2.9 GHz Intel dual core processor with 16 GB 1600 MHz DDR3 memory with Darwin Kernel Version 14.0.0, Java version `1.7.0_51` with Oracle HotSpot 64-Bit Server VM, and Python 2.7.6.

Network: We evaluate all systems using Open vSwitch (OVS) version 2.0.2, which supports both OpenFlow 1.0 (required by many controllers) and OpenFlow 1.3.4, used by Magellan. We vary the number of hosts attached to a switch, with each host attached to a distinct port.

Workload: We evaluate a policy that is available in each system from the system's authors (with minor variations), namely L2 learning and routing. After allowing appropriate initialization of hosts and controller, we then perform an all-to-all ping among the hosts, recording the RTT of each ping and measure the time for all hosts to complete this task. After completing the task, we retrieve and count all Openflow rules installed in the switch.

Results: Figure 10 lists the number of rules, task completion time, and median ping RTT for each system with $H = 70$ and $H = 140$ hosts and Figure 11 charts the median ping RTTs[3]. We observe that for 70 hosts, Magellan uses 33x fewer rules than Maple, ODL and Floodlight, while for 140 hosts, Magellan uses between 46-68x fewer rules than other systems. This rule compression is due to leveraging multi-table pipelines. Other systems generate rules into a single table, and therefore generate approximately H^2 rules, while Magellan generates approximately $2 * H$ rules.

We also observe that Magellan completes the all-to-all ping task 1.2x faster than ODL and 1.4-1.6x faster than Floodlight. Moreover, the median RTT is substantially improved, with Magellan reducing RTT experienced by hosts by 2x versus ODL and between 7x and 10x for Floodlight. This improvement is due to Magellan's proactive rule compilation which generates all rules early in the task - as soon as host locations are learned. In contrast, all other controllers (except Pyretic) generate rules only when a sender sends a first packet to a receiver, and hence other systems continually incur flow table misses throughout the task.

[3]Tests of Maple at 140 hosts and of Pyretic at both 70 and 140 hosts failed and these measurements are therefore omitted.

System	Hosts	Rules	Time (s)	Med RTT(ms)
Maple	70	4767	51	2.0
POX	70	18787	96	9.7
Floodlight	70	4699	37	2.1
OpenDaylight	70	4769	32	0.6
Pyretic	70	-	> 1500	
Magellan	70	142	25	0.3
Maple	140	-	-	
POX	140	13107	389	11.9
Floodlight	140	16451	200	6.1
OpenDaylight	140	19349	150	1.2
Pyretic	140	-	-	
Magellan	140	282	123	0.6

Figure 10: End-to-end performance comparison.

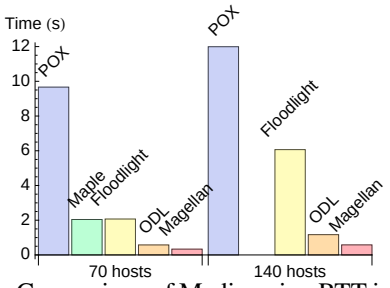

Figure 11: Comparison of Median ping RTT in SDN systems when performing an all-to-all ping task.

7.3 Fast Repair of Forwarding State

We now evaluate the effectiveness of Magellan in repairing forwarding state after a change to system state.

Network topology: We evaluate controllers using four topologies, named "Linear", "Square", "Internet2" and "FatTree". The Linear topology consists of three switches. The Square topology is a small cyclic topology with 4 switches. The Internet2 topology is based on the topology of Internet2 [11] exchanges and includes 14 routers. For Linear, Square and Internet2 we locate one host at each switch. The FatTree topology is a Fat Tree [1] topology with $k = 4$, consisting of 20 switches and two hosts per edge switch.

Workload: In particular, we evaluate Magellan, Maple with trace trees, and Floodlight controllers that forward along shortest paths and a Pyretic controller that forwards along a spanning tree. After initializing the system with all-to-all ping traffic, we remove one link from the network. We then measure the time to complete pings between all hosts. Floodlight relies on rule idle timeouts to remove rules and trigger reactive updates. Some tests trigger a pathological condition when the all-to-all ping following a link restoration prevents idle timeouts

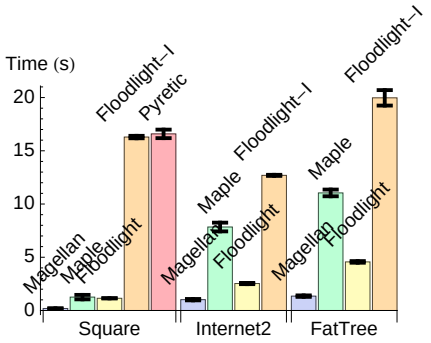

Figure 12: Time restore all-to-all connectivity after a single link failure in three topologies.

from occuring. We therefore measure Floodlight when this condition does and does not occur.

Results: Figure 12 shows that Magellan provides substantial improvement in all topologies. In particular, the mean time to complete an all-to-all ping after a link failure for Magellan is 189 msec, 1.02 sec, and 1.35 sec, for the three topologies respectively. Maple with trace trees, requires from 6x to 8x longer to restore connectivity, and Floodlight requires up to 87x longer when rule idle timeouts fail to trigger and from 2.5x to 6x longer when rule idle timeouts remove rules before the link down event. Pyretic requires 87x longer to restore connectivity due to applying a complex compilation process after each state change. Pyretic did not complete for larger topologies and we therefore omit those results.

7.4 Scalability

The benefits of proactive, compact multi-table compilation may be lost if Magellan's algorithms take too much time. We now evaluate scalability and performance.

Workload: We apply Magellan to `L2-Route`. We vary the number of hosts, H, from 10,000 to 100,000 and the number of switches from 25 to 100. We measure number of rules generated and time to generate these rules when starting from *cold start*: the system state is initialized and then program analysis and rule compilation is started.

Results: Figure 13 shows the number of rules generated as a function of the number of hosts in the system for 25, 50, 100, and 200 switches. We observe that the number of rules grows linearly with the number of hosts, due to the three table compilation result that leverages two tables to extract location information from source and destination addresses into registers and then a final table based on registers, which remains invariant (for a particular number of switches) as the number of hosts varies. Figure 14 shows the amount of time required to compile rules. We observe that Magellan requires under 4 seconds to compile rules for a network of 10,000 hosts and 25 switches and 6.5 minutes to compile rules for a network of 100,000 hosts and 200 switches.

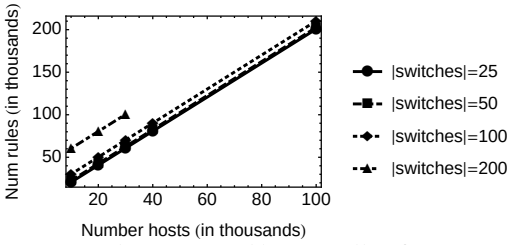

Figure 13: #rules generated by Magellan for L2-route for different numbers of hosts and attachment points.

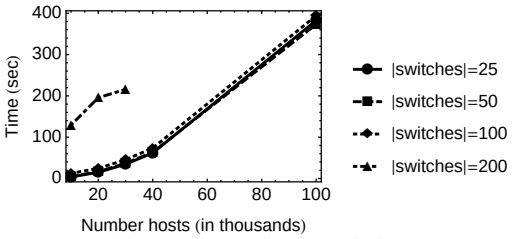

Figure 14: Cold start time: time needed to generate rules for L2-route for different #hosts and attachment points.

8 Related Work

High-level programming models: So-called *Tierless* programming models are most closely related to Magellan's AP abstraction. Of these, only Maple [21] implements general-purpose programs, while FlowLog [15] and FML [10] provide restricted logic-programming. Other languages, such as those in the Frenetic family (Frenetic, Pyretic [6, 14]), provide languages for stateless forwarding policy and require programmers to use a split-level model where a controller compiles a new policy after handling an event. All of these languages and systems only compile to a single OpenFlow flow table.

Low-level SDN control systems: The main mechanism currently available to SDN programmers to handle multi-table pipelines is to use lower-level SDN control systems and APIs (*e.g.*, [4, 5, 9, 13, 16]). Among others, NOX [9], Beacon [4], Floodlight [5] and OpenDaylight [16] offer APIs in C++, Python and Java. These APIs require the programmer to manage low-level OpenFlow state explicitly and hence add substantial SDN programming complexity. In contrast, Magellan automatically handles these low-level details transparently.

Configuration languages for multi-table pipelines: The importance of multi-table pipelines has motivated researchers to to make such hardware easier to use. In [20], a typed programming language called Concurrent NetCore is proposed to specify flow tables. P4 [2] provides a forwarding model with configurable multi-table pipeline and programmable parsers. These languages allow and require users to specify the tables, whereas Magellan automatically derives them. Jose et al [12] study algorithms for mapping table designs to particular target switches, namely RMT and FlexPipe.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[3] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[4] D. Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

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

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[7] O. N. Foundation. Openflow switch specification 1.4.0. Open Networking Foundation (on-line), Oct. 2013.

[8] Group-based Policy (GBP). `https://wiki.opendaylight.org/view/Group_Policy:Architecture/OVS_Overlay#Packet_Processing_Pipeline`.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[10] T. Hinrichs, J. Mitchell, N. Gude, S. Shenker, and M. Casado. Practical declarative network management. In *in ACM Workshop: Research on Enterprise Networking*, 2009.

[11] Internet2. Internet2, 2014. [Online; accessed 29-January-2014].

[12] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, Oakland, CA, May 2015. USENIX Association.

[13] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, Apr. 2014. USENIX Association.

[14] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[15] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 519–531, Berkeley, CA, USA, 2014. USENIX Association.

[16] OpenDaylight. `http://www.opendaylight.org`.

[17] OpenStack. `http://www.openstack.org`.

[18] R. Ozdag. Intel Ethernet Switch FM6000 Series - Software Defined Networking. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf`.

[19] POX. `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[20] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 11–24, New York, NY, USA, 2014. ACM.

[21] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98. ACM, 2013.

A Appendix

`ExploreRegions` (Alg. 6) assumes regions $R_1, \ldots, R_n$ are sorted in topological order of control flow. `ExploreRegions` initializes the reachable set of the first region to consist of a single, empty store and that of other regions are initialized to be empty (lines 1-3). The algorithm then explores each region in all the possible input states (lines 9-10), as computed by `ExploreRegions` calls on earlier regions. To avoid redundantly exploring a region, we first remove all region-irrelevant variables from each store (line 6) to obtain a restricted store s_{in} and only explore a region once per restricted store. Since the region-irrelevant bindings may be relevant in later, the algorithm passes the collection of region-irrelevant bindings to the region exploration procedure (lines 7-8).

Algorithm 6 `ExploreRegions` ():

```
1:  reach[1] = { empty store}
2:  for i = 2...n do
3:      reach[i] = ∅
4:  for i = 1...n do
5:      for s ∈ reach[i] do
6:          s_in = restrict(s, inputs(R_i))
7:          s_other = s − s_in
8:          othermap[s_in].insert(s_other)
9:      for (s_in, others) ∈ entries(othermap) do
10:         ExploreRegion(R_i, entryPC(R_i), s_in, others)
```