Yale University
Department of Computer Science

Towards Automatic Generation of Multi-table Datapath from Datapath-Oblivious Algorithmic SDN Policies

Andreas Voellmy Y. Richard Yang Xiao Shi

YALEU/DCS/TR-1504
Jan. 30 2015

Abstract

Despite the emergence of multi-table pipelining as a key feature of next-generation SDN datapath models, there is no existing work that addresses the substantial programming challenge of generating effective multi-tables automatically. In this paper, we present Magellan, the first system that tries to address the aforementioned challenge and beyond. Introducing a novel algorithm called `Map-Explore`, Magellan extracts efficient multi-table pipelines from a datapath oblivious, high-level SDN program written in a general-purpose language and maximizes local switch processing. We implement a prototype of Magellan for a Turing-complete programming language. 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.

1 Introduction

Multi-table pipelining has emerged as the foundation of the next generation SDN datapath models, such as recent versions of OpenFlow [9], Protocol-Oblivious Forwarding (POF) [23], RMT [3], and Flexpipe [20]. However, it remains a major issue in SDN programming to fully utilize multi-table pipelining. Without the ability to automatically use multi-table pipelines in a given higher-level programming model, SDN programmers may be forced to switch to a lower level programming model where the powerful, but low level features of multi-tables are directly exposed. This can create substantial programming complexity, leading to lower programming productivity. To date, there is no previous study on automatic generation of effective multi-table pipelines, from multi-table oblivious higher level programs.

This paper investigates how to compute effective multi-table pipelines from algorithmic policies [24]. We choose the algorithmic policy model because it is highly flexible and hence poses minimal constraints on SDN programming. As a result of the generality, if we can compute high-quality multi-table pipelines for algorithmic policies, we can use algorithmic policies as a powerful intermediate language for implementing other high-level SDN programming models.

Computing effective multi-table pipelines from algorithmic policies, however, is challenging. In particular, algorithmic policies are expressed in a general-purpose programming language with arbitrary complex control structures (*e.g.*, conditional statements, loops), and the control structures of the program can be completely oblivious to the existence of multi-tables. Hence, it is not clear at all whether one can extract effective multi-table pipelines from such programs. We refer to this as the *multi-table extraction challenge*. In the original paper [24] which proposed the algorithmic policy model, the authors use an approach called trace trees to generate

flow table rules. Their approach, however, is limited to a single table setting. Even in the single-table setting, one can show that the approach may generate ineffective flow tables, not to say the setting of the more challenging multi-tables. Although there is previous work on how to use multi-table datapath (*e.g.*, [2, 22]), the setting is that the tables and their pipelining are already given.

This paper introduces Magellan, the first system that addresses the multi-table extraction challenge and beyond. The core of Magellan is a novel, substantial algorithm called `Map-Explore`, which conducts a novel, efficient form of hybrid *symbolic* (map) and *direct* (explore) execution of a multi-table oblivious program written in a general-purpose language, resulting in a data-structure called *mapper-explorer graph*. Considering and taking advantage of the computational capabilities (*e.g.*, only matching, availability of metadata as registers) of flow tables when constructing the mapper-explorer graphs, Magellan maps mapper-explorer graphs into flow tables, considering practical switch hardware constraints. Morevoer, Magellan addresses an additional key challenge in SDN programming: efficient flow table utilization and update under system dynamics. Using `Map-Explore`, Magellan automatically generates flow tables that maximize local switch processing and maintains switch-controller consistency despite system updates.

We implement a prototype of Magellan for a Turing-complete programming language. Comparing the flow tables generated by Magellan 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.

Despite the substantial progress made by Magellan, there are limitations on what Magellan can achieve with only flow tables. In particular, there are algorithms that are simple to express in a general-purpose language but fundamentally impossible to express *compactly* using flow tables.

The rest of the paper is organized as follows. We first illustrate the problem and our basic ideas in Section 2. In Section 3, we present the system architecture and details of the `Map-Explore` algorithm. We evaluate Magellan in Section 4 and give related work in Section 5.

2 Basic Ideas

We start with a simple, but representative example to illustrate the basic challenges and ideas. Section 3 presents an architecture to systematically implement the ideas.

2.1 State Read-only Policies

Example and programming model:

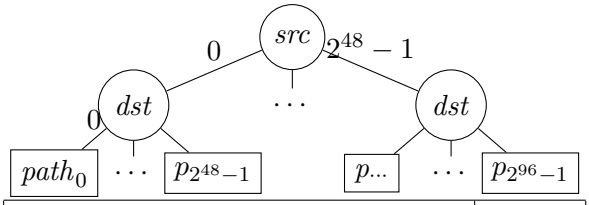
```
// Program: L2-Route
1.  Map macTable(key: macAddress, value: sw)

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

Consider an example algorithmic policy, `L2-Route`, shown above, to implement routing on layer 2 addresses. In this example and throughout this paper, we use the following algorithmic policy 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 returns 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 set of system states, and processing each packet according to packet attributes and current system states. Specifically, line 1 of `L2-Route` declares its state variable `macTable`: a key-value map data structure that associates each 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.

Problem of previous work: The only previous work that handles general algorithmic policies is [24], 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 complete trace tree and the flow table for `L2-Route`. For example, the bottom left result $path_0$ is the execution trace of a packet with $macSrc$ 0 and $macDst$ 0.

This example shows that the trace tree approach is inefficient in two ways. First, it needs 2^{96} invocations, which we call an exploration, to complete the trace tree, due to its blackbox nature. Specifically, packet attributes $p.macSrc$ and $p.macDst$ are free variables, in that they are formal arguments of f and hence do not have actual value bindings until a specific invocation of f. Hence, a minimum of 2^{96} invocations is needed to complete the exploration of all possible bindings.



Match	Action
$macSrc : 0, macDst : 1$	p_0
...	...
$macSrc$: 0xff...ff, $macDst$: 0xff...ff	$p_{2^{96}-1}$

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

Second, with one rule for each leaf, the flow table will have 2^{96} rules; this is clearly impractical.

From blackbox to whitebox exploration: Since the preceding has shown that the largely blackbox based trace-tree approach does not scale, it is intuitive to try a whitebox based approach. A key benefit of a whitebox based approach is that it can see *derived* free variables–variables that depend on packet attributes. The trace-tree approach focuses only on packet attributes, but $srcSw$ and $dstSw$ in `L2-Route` are free variables as well.

Algorithm 1 DirectExplore $(PC, store)$

```
1: if (PC, store) has been explored then
2:     return
3: Mark (PC, store) as explored
4: ins = prog[PC]
5: switch (ins.type) do
6: case RETURN:
7:     Save return value
8:     return
9: default:                                              ▷ exploration
10:    (x_1, x_2, ..., x_k) = variables used in ins
11:    for (v_1, ..., v_k) ∈ (store[x_1], ..., store[x_k]) do
12:        store' = store[x_1 ↦ {v_1}, ..., x_k ↦ {v_k}]
13:        (nextPC, store') = EXECUTE(ins, store')
14:        Remove vars from store' if not live at nextPC
15:        DIRECTEXPLORE(nextPC, store')
16: end switch
```

An algorithm that can convert an algorithmic policy f to flow table rules must have computed the output that f returns on each possible value of packet attribute.

For this goal, given the structure of function f, we can design a naive white-box exploration algorithm, `DirectExplore` to compute all outputs. The algorithm models the execution state of f with a pair $(PC, store)$, where PC is the program counter, and $store$ is the execution state, which is a symbolic store that consists of a collection of variable-range bindings. Specifically, `DirectExplore` takes the instruction at the current program counter (line 4) and identifies the variables used in the instruction (line 10). Then it explores (line 11) all bindings allowed by the current ranges of the variables. The initial range of each packet attribute is its allowed set of values. For each binding, the algorithm can execute the instruction (line 13), because each free variable has a single value and hence can be executed. The execution of an instruction (`DirectExplore`) returns the next PC, which will be the following instruction for a sequential instruction and jumped instruction if conditional or jump instruction. `DirectExplore` recurses on this process. Note that the variable $store$ keeps the bindings (*i.e.*, ranges of variables) along each exploration path, recording the execution state. Using the memorization technique (lines 1-2), `DirectExplore` avoids recursion when the bindings (*i.e.*, the $store$) are the same at the same PC. Conceptually, this happens when the execution paths of different inputs merge. To increase merging, line 14 removes all variables from the store that are not *live* at $nextPC$. This is safe, since non-live variables can not affect further computation. One can verify that this algorithm computes all returns of a given f.

Although simple, `DirectExplore` is not practical. For example, consider applying this algorithm to `L2-Route`. It will repeat Line 10 2^{48} times because $p.macSrc$ is a free variable and its range is 0 to $2^{48} - 1$. We define such statements as non-direct executable statements:

Definition 1 *A non-direct executable statement is one which has large fan out if using* `DirectExplore`*.*

Exploration at only low fan-out cut: Instead of giving up the simplicity of white-box exploration using `DirectExplore`, we introduce a key novel idea: we revise `DirectExplore` so that it conducts exploration (i.e., executes line 11) for only statements whose free variables have small ranges. Consider line 5 of `L2-Route`. At this statement, only $srcSw$ and $dstSw$ are accessed and each may have a low range. Assume a network of 999 switches, and the lookup returns `null` if look up fails. One can compute that exploring line 5 needs only $1M (= 1000 \times 1000)$ combinations. We say that line 5 has a "cut" with a small enumeration fan out.

Table mapping, not exploration: To generate flow table rules, we need to obtain the mapping from packet attributes to return results. Since smart exploration may start at the middle of a program (*e.g.*, line 5 of `L2-Route`), the missing piece is to

obtain the mapping from input packet attributes to derived variables (*e.g.*, $srcSw$ and $dstSw$ respectively for `L2-Route`). A naive approach of obtaining these flow table mappings is exploration (*e.g.*, explores lines 3 and 4 of `L2-Route`), but this is not possible.

Our second insight is that in the multi-table pipelines, this naive mapping exploration is unnecessary. Instead, one can recognize program statements with compact flow table mappings which can perform these mappings in flow tables at runtime. For example, the mapping from packet attributes to $srcSw$ is simple: it is just a simple flow table that one can construct given $macTable$, matching on keys of $macTable$.

Definition 2 *A compact mappable statement is a programming statement satisfying the following two conditions: (1) its effect can be represented by a compact flow table; and (2) its outcome has a small range.*

Note that both compact and small are relative terms measured by quantity. Hence, in real implementation, we define a predicate to determine the threshold.

Since a smart explorer obtains the effects of lines 3 and 4 of `L2-Route` without real execution of them, we say that the smart explorer executes lines 3 and 4 *symbolically*.

Note that there exist statements that are simple but are not compact mappable statements.

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.*

Such fundamentally challenging statements can be implemented through a reactive, on-demand approach or through instructions provided by more expressive forwarding models, such as POF [23].

Table linking using registers: The remaining issue is how to integrate the two packet attribute mapping tables and the results of the exploration at line 5. This turns out to be achievable through a novel use of standard datapath registers: the system consists of three tables, where the first table matches on $macSrc$ and encodes the outcome (srcSw) in a *register*, the second matches on $macDst$ and encodes the outcome (dstSw) in a different register, and third (and final) flow table matches on the two stored register values and sets the appropriate action to take on the packet. This collection of flow tables, depicted in Figure 2, consists of $M + M + O^2$ rules, where M denotes the number of hosts listed in `macTable` and O denotes the number of outcomes (*e.g.*, 1000) in `macTable`. Hence, we obtained a efficient multi-table pipeline with compact flow tables.

The mapper-explorer graph: Figure 3(A) illustrates the preceding insights. We refer to the graph shown in the figure as a *mapper-explorer graph*. The first inverted

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*, goto 2

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*, goto 3

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 2: The optimized multi-flow-table pipeline for the `L2-Route` policy.

trapezoid (representing its reduction effect) shows the effect of looking up the free variable $p.macSrc$ in $macTable$. The second inverted trapezoid represents the effect of looking up the free variable $p.dstMac$ in $macTable$. The third node of the figure represents the exploration in the middle of f.

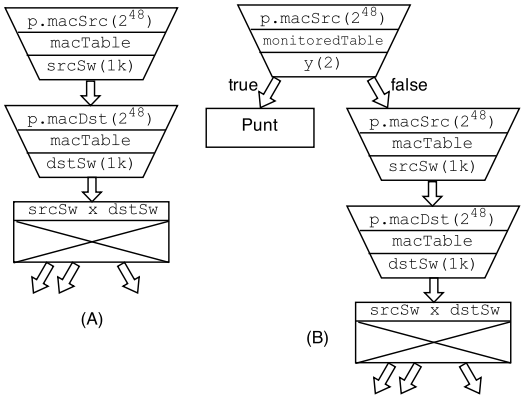


Figure 3: The mapper-explorer graph of `L2-Route`.

2.2 State-updating Policies

An algorithmic policy may not only read states but also update them. Consider an extension to `L2-Route` which saves the last packet sending time of a set of monitored (say stolen) MACs:

```
1. Map macTable(key: macAddress, value: sw)
2. Map monitoredTable(key: macAddress, value: time)
3. onPacket(p):
4.   if (p.macSrc in keys(monitoredTable)):
5.     monitorTable[p.macSrc] = time()
     ... as before ...
```

Program rewriting and punt: This updated program introduces two features: the if statement with free variable and the system update. These new features, however, are relatively easy to handle by simply rewriting the program as follows:

```
y = p.macSrc in keys(monitoredTable)
if (y):
   monitorTable[p.macSrc] = time()
```

With this rewriting, the table mapping technique is still applicable, and hence, leading to a table with match based on the keys of monitorTable. It is possible to explore the `if (y)` statement, since `y` has only two values. Furthermore, it is straightforward to detect that the true case needs controller action (Punt) instead of local switch action. The mapper-explorer graph is shown in Figure 3(B).

The preceding example has touched on many key ideas of our design: exploration only on low fan-out statements, auto table from mapping statements, using registers for method results passing, and automatic punt for system updates. The key contribution of this paper is that we develop them into systematic techniques for real, complex programs.

3 Magellan

The objective of Magellan is to implement the preceding basic ideas in general settings, for real, complex policy programs. Figure 4 shows the key components and work flow.

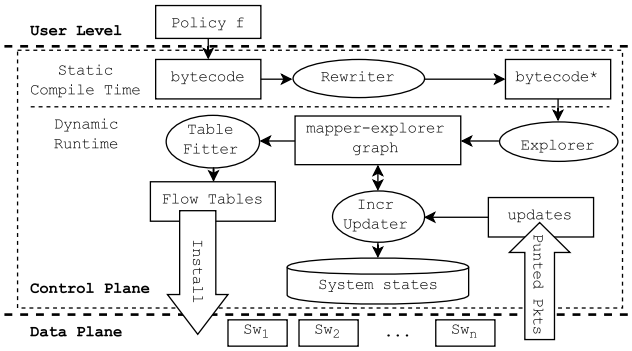


Figure 4: Magellan system components and workflow.

The design of Magellan faces two key challenges: (1) the complexity of a general-purpose programming language, and (2) the complexity of handling realistic datapath constraints. Magellan uses a modular design to decouple the handling of the complexities, with mapper-explorer graphs being the central coordination data structure.

Specifically, Magellan conducts static bytecode rewrite (Rewriter) and dynamic, smart exploration (Explorer), to convert a policy in a general-purpose language to the simple, language-independent mapper-explorer graph data structure. Guided

by the mapper-explorer graph and also considering key datapath constraints (*e.g.*, number of tables), the table fitter performs a set of transformations on the mapper-explorer graph to produce the final multi-table pipeline.

3.1 Compile-time Analysis

The Rewriter performs analysis on a language-independent, general-purpose bytecode and outputs an equivalent program in an extended bytecode, called bytecode*. The extended bytecode includes new `BRK` instructions for compact-mappable instructions, system state accesses (read/write) and packet field accesses. The Rewriter applies data flow analysis to replace non-compact mappable statements with these new instructions whenever possible. In addition, the analysis phase produces information, such as variable liveness, which is leveraged by runtime exploration.

3.2 The Map-Explore Algorithm

The basic structure of `Map-Explore` is easy to describe: it extends `DirectExplore`, by detecting compact mappable statements and processing them in a new `MAP` case. Hence, mappable statements are processed, essentially symbolically, compared with a conventional executor. Another difference between `DirectExplore` and `Map-Explore` is to process native instruction as fast as possible. This is achieved by introducing the `XStep` method, that executes bytecode* instructions up to the next `BRK` instruction. We omit the detailed code of `XStep` due to its simplicity. Moreover, the execution of `Map-Explore` creates the mapper-explorer graph. In this graph, each node has three attributes: an unique ID (line 2), which is a combination of the PC and the state of the store (*e.g.*, using a hash); a type (RET, MAP, EXEC); and the parameters specific to each type. The constructor Node() takes a node id, a type, and the parameters of the type. One may use inheritance for a strong typed language to define different types of nodes.

Only controller actions: The function f may include behaviors, in particular, sending messages and/or updating state variables, that only the controller can do. To support such behaviors, `Map-Explore` will detect such statements and send packets back to the controller. In `Map-Explore`, the detection is in the `UPDATE` case.

It is important to note that `Map-Explore` will only send packets to the controller if they would execute update or send statements according to the program, *in the current system state*. Hence as system state changes, further packets may no longer be punted to the controller, even if the program contains update statements.

Algorithm 2 `Map-Explore` $(PC, store)$: build mapper-explorer graph $G_{mx} = (V_{mx}, E_{mx})$

$ins = prog[PC]$, $nid = (PC, store)$
if $nid \in V$ **then return**
switch $(ins.type)$ **do**
case `BRK.RETURN`:
    $V = V \cup \{Node(nid, \texttt{RET}, ins.value)\}$
    return
case `BRK.MAP`:
    $x =$ the free variable that ins maps.
    $outcomes =$ possible assignments to x by ins.
    if $|outcomes| < LIM$ **then**
        $V = V \cup \{Node(nid, \texttt{MAP})\}$
        $store' = store[x \mapsto outcomes]$
        Remove vars from $store'$ if not live in $nextPC$.
        $nid' = (nextPC, store')$
        $E = E \cup \{(nid, nid')\}$
        MAPEXPLORE$(nextPC, store')$
        return
case `BRK.UPDATE`:
    $V = V \cup \{Node(nid, \texttt{PUNT})\}$
    return
default: ▷ Begin explore
    $(x_1, x_2, \ldots, x_k) =$ vars used in paths to next `BRK`s
    $V = V \cup \{Node(nid, \texttt{EXEC})\}$
    for $(v_1, \ldots, v_k) \in (store[x_1] \times \ldots \times store[x_k])$ **do**
        $store' = store[x_1 \mapsto \{v_1\}, \ldots, x_k \mapsto \{v_k\}]$
        Remove vars of $store'$ not live on exit of PC.
        $(nextPC, store') =$XSTEP$(ins, store')$
        $nid' = (nextPC, store')$
        $E = E \cup \{(nid, nid', [v_1, \ldots, v_k])\}$
        MAPEXPLORE$(nextPC, store')$
end switch

Proposition 2 *Programs whose* `BRK` *instructions are all compact-mappable will only punt backs that* must *be punted to perform state updates.*

3.3 Table Generation

With a given mapper-explorer graph, we can generate flow tables using a table generation algorithm. The table generation proceeds in three steps: (1) `Graph2Tables`, which converts a mapper-explorer graph G_{mx} to a flow table graph G_{ft} with the same structure; (2) `TableAggregation`, which aggregates flow tables in G_{ft} to reduce the number of tables used; and (3) `RegisterAllocation` which encodes program variables into dataplane-matchable registers.

Graph2Tables: Since mapper-explorer graphs are direct acyclic graphs, we design `Graph2Table` to process the nodes in a given mapper-explorer graph G_{mx} in an inverse topological order, ending with without outgoing edges. Table IDs are assigned in processing order, and therefore no table will have a rule that jumps to a table with a lower or the same table id, satisfying OF specification [9].

The generation of the flow table for each node is mostly straightforward. A RET or PUNT node generates a table with a single entry. The flow table for a MAP node depends on the instruction type. For example, the table for a lookup instruction $t[pattr_1, \ldots, pattr_n]$ consists of one rule per entry in t, where each rule has a match condition that matches each packet attribute $pattr_i$ against the corresponding entry's key, and has an action that writes the entry's value into a *logical register* corresponding to the target variable of the instruction and then jumps to the successor table. In addition there is one final rule at a lower priority with no match condition (unrestricted match) and setting the logical register to $null$ prior to jumping. Other instruction types are mapped to tables in appropriate ways.

The entries of the flow table created for an exploration (EXEC) node match entirely on the logical registers. Each rule jumps to its successor in the graph and prior to jumping writes values to all logical registers whose values have changed between the node's store and the successor's store.

Table Aggregation: We apply simple heuristics to reduce the number of tables used. In particular, we inline any tables for leafs (RET and Updates) and inline any tables that are jumped to by exactly one rule. Inlining these cases can not increase the number of rules used, and may reduce the number of tables required.

Register Allocation: The flow tables produced by `Graph2Table` use one *logical register* per program variable. For switches supporting several actual registers, such as Open vSwitch [19], we adapt graph coloring register allocation from compilers, to map the potentially large number of logical registers to a small number of actual registers. Different register allocation algorithms may be developed for

switches that provide only one wide, maskeable metadata register.

4 Preliminary Evaluation

In this section, we demonstrate that Magellan improves end-to-end performance over existing systems by proactively generating compact forwarding rules and thereby eliminating many flow table misses. Our prototype consists of 5500 lines of code and includes an OpenFlow 1.3 controller.

Control Systems: We compare Magellan with a range of SDN systems, including OpenDaylight (ODL) (Helium), Floodlight (1.0), POX [21] (`forwarding.l2_learning` module from 0.2.0), Pyretic (latest version), and Maple (0.10.0). POX, Pyretic and Maple are academic systems supporting novel policy languages and compilers, while ODL and Floodlight are industry-developed open source controllers that form the basis of commercial systems. We run controllers on a 2.9 GHz Intel dual core processor with 16 GB 1600 MHz DDR3 memory with Darwin 14.0.0, Java `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 and OpenFlow 1.3.4. We vary the number of hosts attached to a switch, with each host attached to a distinct port.

Workload: We evaluate a policy available in each system from the system's authors (with minor variations), L2 learning. 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 5 lists the number of rules, task completion time, and median ping RTT for each system with $H = 70$ and $H = 140$ hosts and Figure 6 charts the median ping RTTs[1]. 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.

Magellan completes the all-to-all ping 1.2x faster than ODL and 1.4-1.6x faster than Floodlight. The median RTT is substantially improved, with Magellan reducing RTT experienced by hosts by 2x versus ODL and between 7x and 10x

[1]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 5: End-to-end performance comparison.

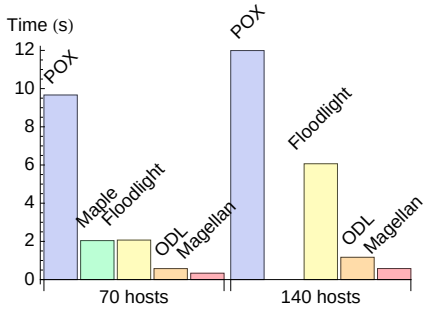


Figure 6: Comparison of Median ping RTT in SDN systems when performing an all-to-all ping task.

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 during the task.

5 Related Work

High-level programming models: We classify existing models into two categories: "tierless" and split-level. In a split-level model (*e.g.*the Frenetic languages [8, 15, 16, 1]), a controller program outputs a new stateless network policy after each relevant event. Magellan provides a tierless programming model, in which programmers specify forwarding behaviors as a packet handling function which has access to system (*i.e.*, control) state. FlowLog [17] and FML [11]) provide tierless programming with limited expressivity (non-recursive Datalog variants).

All the preceding programming systems focus on compiling to a single 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.*, [6, 7, 10, 13, 18]). In particular, NOX [10] offers C++ and Python APIs for raw event handling and switch control, while Beacon [6], Floodlight [7] and OpenDaylight [18] offers a similar API for Java. These APIs require the programmer to manage low-level OpenFlow state explicitly, such as switch-level rule patterns, priorities, and timeouts, and hence add substantial SDN programming complexity. In Magellan, such low level details are transparent to the programmers.

Configuration languages for multi-table pipelines: Several researchers investigate how to make multi-table hardware pipelines hardware easier to use. Concurrent NetCore [22] provides a typed programming language for specifying flow tables. P4 [2] provides languages for specifying parsers, flow tables, and processing order through flow tables. The objective of both systems is for users to specify the tables. In contrast, Magellan does not require that programmers specify flow tables; instead, tables are automatically generated.

Symbolic execution and program analysis: A key idea of our design is to avoid direct executions. A related technique is symbolic execution (*e.g.*, [12, 14, 5]), which executes a program using symbolic expressions to model inputs and system state. Symbolic execution has been used ([4, 5]) to provide systematic analysis and testing of programs. Our system is a novel application of symbolic execution for program compilation in the context of SDN.

6 Acknowledgements

We thank Jose Faleiro for engaging discussions. We thank Xin Wang, Yichen Qian and Huaming Guo for suggestions.

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. 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] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] M. Dobrescu and K. Argyraki. Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114, Seattle, WA, Apr. 2014. USENIX Association.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[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] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 217–230, New York, NY, USA, 2012. ACM.

[16] 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.

[17] 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.

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

[19] Production quality, multilayer open virtual switch. `http://openvswitch.org/`.

[20] 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`.

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

[22] 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.

[23] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[24] 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.