Magellan: Toward Automatic Mapping from High-Level SDN Programs to Low-Level, Optimized Datapath Pipelines

ABSTRACT

Despite the emergence of pipelining as a key feature of next-generation SDN data-path models, there are no previous studies on how to automatically design, populate, and update datapath pipelines from completely datapath-oblivious, high-level SDN programs. We refer to this as the pipelining-oblivious programming challenge. This paper presents Magellan, the first systematic investigation of the challenge. Including multiple novel algorithms, including a novel, efficient combination of symbolic map and flow-rule guided execution to compute compact pipeline contents, and an efficient pipeline design algorithm to automatically optimize pipeline resource usage and satisfy hardware constraints, Magellan achieves automatic design, population, and updates of datapath pipelines from completely datapath-oblivious, high-level SDN programs. We implement Magellan and apply it to multiple high-level programs to demonstrate its benefits.

1. INTRODUCTION

The availability of high-level control-plane programming can be a key benefit of Software Defined Networking (SDN). Just as the availability of programmer-efficient, high-level programming, on top of machine-efficient, low-level machine execution, makes a key contribution to the success of general computer software, the availability of easy-to-program, high-level SDN programming, on top of efficient, low-level network datapath forwarding, has the potential to substantially realize the benefits of SDN.

In the last few years, substantial progress has been made in designing and realizing efficient, low-level network datapath forwarding. In particular, recent advances (e.g., P4 [4], RMT [5], recent versions of OpenFlow [10], and Flexpipe [20]) have shown that flow-table based pipelines, or pipelines for short, can provide a highly compact, high-performance low-level network datapath forwarding model. As a result, there have been substantial recent interests in fully realizing datapath pipelining, and substantial progress (e.g., [4, 14, 23, 16, 24, 25]) has been made.

Despite the substantial interests and progress, there are no previous studies on how to automatically design, populate, and update datapath pipelines from general-purpose, completely datapath-oblivious, high-level SDN programs. We refer to this as the pipelining-oblivious programming challenge. Existing work either starts with given pipelines (e.g., [4, 23]), leaving the challenge of pipeline design to programmers, or cannot use pipelines automatically (e.g., [27]), missing the substantial benefits of multi-table pipelining.

The objective of this paper is to conduct the first systematic study on the pipelining-oblivious programming challenge. After all, it is not obvious that it is feasible to achieve high-level, pipelining-oblivious programming. In particular, a high-level, pipelining-oblivious program expressed in a general-purpose programming language can have complex computation (e.g., boolean and arithmetic expressions) and control structures (e.g., conditional statements, loops). On the other hand, to achieve substantial high throughput (e.g., millions of packets per second), low-level pipelines, in general cases, provide only simple lookups. The gap between these two models could be too large to be automatically bridged without programmers’ explicit support.

A key contribution of this paper is to show that, surprisingly, there can be systematic techniques to automatically design, populate and incrementally update low-level pipelines from high-level, completely pipelining-oblivious programs, in quite real settings. We demonstrate the result using a real system called Magellan.

Specifically, Magellan computes and updates practical, optimized pipelines from pipelining-oblivious programs in steps. First, through a basic but important insight that instructions and table lookups are two representations of computation, Magellan turns an arbitrary program into a theoretical pipeline, where each instruction can be considered as a flow table, leading to theoretical feasibility. Then, Magellan computes a largely theoretical pipeline to a real, compact pipeline, through a highly novel, efficient combination of symbolic map and direct execution. In particular, symbolic map takes advantage of instruction recognition and system state constraints to automatically compute flow tables for multiple types of instructions; a network specialized direct execution called FlowExplore then takes these flow tables and conducts flow-rule guided state propagation to compute flow tables for each instruction, leading to a compact pipeline. Further, Magellan applies a novel, efficient pipeline design algorithm called DeepExpansion to automatically optimize the compact pipeline resource usage and satisfy hardware constraints, leading to practical, optimized pipelines.

We implement Magellan and apply it to multiple high-level programs, in diverse settings including switch.p4, GBP [11] and interdomain. We show that Magellan can identify table resource sharing optimization effectively. Comparing the pipelines generated by Magellan with those 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 com-
pletes all-to-all pings after a link failure 87x faster than that of full recompilation.

We emphasize that despite the substantial progress made by Magellan, there are limitations on what it can achieve and the pipelining-oblivious programming challenge remains. In particular, there are SDN programs that are simple to express in a general-purpose language but fundamentally impossible to be expressed *compactly* using pipelines. Also, we show that Magellan can generate effective multi-table pipelines, but the general problem of generating optimal flow tables is computationally hard.

The rest of the paper is organized as follows. Section 2 illustrates the problems that programmers using pipelines face and Magellan addresses. Section 3 gives the system architecture and an overview of the steps. Sections 4 and 5 give details on two key components of Magellan. We evaluate Magellan in Section 6 and give related work in Section 7.

2. MOTIVATION

We start with a simple example to illustrate the complexities that programmers face and Magellan addresses.

**Example program**: Consider an example high-level SDN program, Example. Expressed in a high-level, algorithmic (also called one-big-switch) programming model ([2, 27]). Example is a simple but representative high-level SDN program. Such a program (1) maintains a set of system states, and (2) defines a general logic (we assume it is a function called `onPacket`) to process, in particular, how to route each packet according to the attributes of the packet and the current state. We emphasize that the key algorithms that we develop in this paper are not specific to a particular programming style. For example, the pipeline-design algorithm can be applied to only schema (e.g., P4), and the pipeline-population algorithm can be used to other settings such as standard OpenFlow packet-in handlers.

```java
// Program: Example
Map condTable(key: macAddress, value: GBP cond)
Map hostTable(key: macAddress, value: sw)
Set openPorts(key: port)

L0: onPacket(p):
L1: if (p.dstPort < 1025) {
L2: dstSw = SW_FW
L3: dstCond = V
L4: } else if (openPorts.contains(p.dstPort)) {
L5: dstSw = hostTable[p.srcMac]
L6: dstCond = condTable[p.dstMac]
L7: } else {
L8: egress = Drop; return
L9: // All pairs paths for all dstConds
L10: allPaths = AllPairsCond.execute( ... );
L11: srcSw = hostTable[p.srcMac]
L12: egress = allPaths[dstCond].get(srcSw,dstSw);
}
```

Specifically, the first three lines of the program declare three state variables: `hostTable` and `condTable` are key-value map data structures that associate each layer 2 (L2) endpoint to its attachment switch, and host condition (e.g., authentication status, priority, respectively); `openPorts` is a set containing allowed TCP ports. Below is an example state for these three state variables for illustration purpose.

```
hostTable:
MAC Switch MAC Condition
11 SW1 V (Verified)
22 SW2 V (Verified)
33 SW2 V (Verified)
*   SW_UNK UV (Unverified)
```

openPorts: (22, 53, 80, 8000, 8080, 9090)

The `onPacket` function specifies, at a high-level logic, how to process each packet according to packet attributes and system state. Line 1 of the function tests whether the packet is to a privileged port (also called well-known ports [22], which are ports 0 to 1024). If so, the packet must go to a special firewall `SW_FW` switch (line 2); if the port is not a privileged port but a port allowed in `openPorts` (line 4), the packet can go to its destination switch, which is found by the `hostTable`; otherwise, the packet is dropped (lines 7-8). Since the route of a packet not dropped depends on the verification condition of the destination switch, lines 3 and 6 assign the conditions correspondingly. The programmer writes `AllPairs`, which is a batch function that computes a path (e.g., shortest path) for each pair of switches under each condition (line 9). Line 10 finds the `srcSw` of the packet and line 10 finds the next hop from `srcSw` to `dstSw` by looking up in the batch computed results (line 11).

Unfortunately, a high-level program such as Example is a logically centralized program, and hence not suitable to be executed on switches. In the current systems, an SDN programmer needs to convert it into the datapath programming model. This, however, can be a quite challenging task.

**Low-level statement encoding complexity**: Consider how to process line 1 on the datapath. A naive, single flow table encoding of it is to create a flow table with each row for one value of `p.dstPort`. This will need 65,536 entries. A more compact encoding that one may realize is that one needs only the first 1025 entries, substantially reducing the number of entries. An even more compact encoding, which most programmers that we have surveyed do not know, uses a much more compact encoding shown below, leveraging TCAM wildcard matching and priority capabilities:

```
priority dstPort outcome
2 000000xxx xxxxxxxx true // 0-1023
2 00001000 00000000 true // 1024
1 xxxxxxxxx xxxxxxxx false // others
```

**Flow table pipeline structure complexity**: Assume that the programmer figures out the preceding compact encoding, and realizes that she can use a conceptually clean design with one table representing one statement, obtaining a pipeline (Pipeline I) shown in Figure 1. During this process, the programmer needs to use current state to compute the content of tables for several statements. For example, table for L6 will have 4 entries according to example state of `hostTable` shown at the beginning of this section. We show them as "valid macs" in Figure 1 to save space.

An issue of Pipeline I, however, is that it has more flow rules than necessary. Consider Pipeline II shown in Figure 2. One can see that Pipeline II does not mirror the program structure any more and hence can be less easy for many programmers to come up. A benefit of Pipeline II is that it has not only fewer tables but also fewer rules than Pipeline I (a patient reader can try to verify that Pipeline II has only 22
Figure 1: Pipeline I: One table per instruction pipeline design for Example.

Figure 2: Pipeline II: Optimal pipeline design for Example.

Figure 3: Pipeline III: Optimal 2 table pipeline for Example. rules, while Pipeline I has 34). As flow tables are scarce resources, it is valuable to design a pipeline with less resource consumption, but such optimization can be quite challenging to typical programmers.

**Hardware constraint complexity:** Some hardware limits the number of flow tables, in particular stages, to achieve high throughput. Consider a setting that the hardware allows only 2 tables (a constraint of a major switch vendor). Then, neither Pipeline I (9 tables) nor Pipeline II (4 tables) can satisfy this constraint. For the system state, Pipeline III (Figure 3) is the optimal pipeline, in terms of the smallest number of flow rules, for any pipeline with less than or equal to 2 tables. Deriving this optimal pipeline, however, will not be obvious to most programmers. For example, this optimal pipeline combines in the first table p.dstPort and p.dstMac, but these two packet attributes are referenced quite apart in the program, in the first and second to last program statements respectively.

**State update complexity:** Example is simple and a real program may contain more complexity. As an example, a more complex program may need to handle MAC learning, which can be achieved by adding an instruction at the beginning of Example:

```java
// Program: Example-MacLearn
onPacket(p):
  hostTable[p.srcMac] = p.ingressSw
  // rest of Example
```

Given such a statement, one might be concerned that now every packet needs to go back (called punt) to the controller. This, however, is not the case. One can avoid unnecessary punts by placing a guard before state updates as shown below, so that the controller is involved only when the state really needs to be updated. Detecting such updates and automatically generating controller code to handle the punts, however, will further increase programmer complexity.

```java
onPacket(p):
  if (hostTable[p.srcMac] == p.ingressSw) {
    hostTable[p.srcMac] = p.ingressSw
  }
  // rest of Example
```

**Summary:** Although a high-level SDN program (or at least the high-level logic) can be quite clear, realization of a high-level program in low level datapath can be challenging due to complexities including low level statement encoding, optimizing flow table pipeline structure, and satisfying switch hardware constraints. This paper develops the first systematic framework and multiple novel algorithms to address the complexities and substantially simplify SDN programming.

### 3. OVERVIEW

Magellan needs to handle some substantial complexities. Its input can be a complex, high-level SDN program. Its output includes multiple non-trivial data structures, including (1) specification of pipeline schema; (2) content of the pipeline; and (3) an intermediate data structure allowing efficient update of the pipeline content. As a result, the data structures and algorithms present may appear complex at a first glance, although the underlying ideas are always simple.

To help better present Magellan, we divide its components and workflow of Magellan into a four-step system, labelled 1-4 in Figure 4. The first step, from program to theoretical pipeline, is more conceptual; the next three steps (from theoretical pipeline to compact pipeline; from compact pipeline to practical, optimized pipeline; incremental updates) are more algorithmic. We give an overview of the steps in this section. Sections 4 and 5 present details of the two key steps (the second and third steps).

![Magellan system components and workflow](image-url)
putational model than the instruction-set (or more generally, algorithmic) computation models.

**Approach:** The first (conceptual) step of Magellan turns an arbitrary program into a theoretical pipeline \( P_i \), through a basic but important insight that each instruction can be considered as a flow table, leading to theoretical feasibility.

Specifically, although the detailed syntax of a high-level programming language can be complex, there are essentially only two types of instructions, assignments and control. In programming languages, one may also say two types of statements. For consistency of terminology, we use instructions in this paper. One can construct a flow table for each instruction, achieving conversion of a program to a pipeline:

- **assignment instruction** \( I \) \((y = \text{ins}(x_1, x_2, ..., x_n))\), followed by instruction \( J \), where \( x_i \) has domain \( X_i \). For example, assume that \( I \) has two inputs \( x_1 \) and \( x_2 \), \( x_1 \)'s domain has values \( u_1, u_2, ..., u_{3f} \), and \( x_2 \)'s domain has values \( v_1, v_2, ..., v_N \). Then \( I \) can be encoded as a table matching on \( x_1 \) and \( x_2 \) and the actions consist of (1) assigning a value to \( y \), and (2) jumping to the flow table for the next instruction:

  \[
  \begin{array}{c|c}
  x_1 & \text{actions} \\
  u_1 \ v_1 & y-I(u_1, v_1), \text{jump to } J \\
  \vdots & \vdots \\
  u_M \ v_N & y-I(u_M, v_N), \text{jump to } J \\
  \end{array}
  \]

- **control instruction** \( I \). For example, an unconditional jump instruction can be constructed as a table with a single wildcard entry, with action jumping to flow table of the jump; a conditional instruction if \((g \{ J_1 \ldots \} \text{ else } \{ I_2 \ldots \})\), can be encoded as a table matching on \( g \) and the action is to jump to the flow table of the corresponding instruction:

  \[
  \begin{array}{c|c}
  g & \text{action} \\
  \text{true} & \text{Jump to table for } I_1 \\
  \text{false} & \text{Jump to table for } I_2 \\
  \end{array}
  \]

The preceding conversion can lead to a jumping loop among the flow tables, if there is a loop in the program. Looping, however, is typically not allowed by current datapath pipeline structures. Fortunately, as we focus on structured programming, a loop can be considered as a single block of instructions computing the assignments of a set of variables. Hence, we handle a loop as an extended instruction.

**Outcome:** Hence, the first step converts an arbitrary program to a sequence of instructions, where the jumps among the instructions form a directed acyclic graph (DAG). Instructions which do not depend on packet attributes are called non-free instructions and can be moved out and their results precomputed. \( L_9 \) of Example 1 is an example non-free instruction. Removing such instructions can result in substantial reduction of program length.

### 3.2 From Theoretical to Per-Instruction Compact Pipeline \( (P_I) \)

**Problem:** The table construction used in the preceding step is for theoretical feasibility only, and hence may not be compact. For example, the theoretical construction for if \( \text{p.dstPort} < 2015 \) enumerates the domain of \( \text{p.dstPort} \), which has \( 2^{16} \) values. This clearly is not compact.

**Approach:** The second step, from a theoretical pipeline to a compact pipeline, computes a compact flow table representation for each instruction, resulting in a compact per-instruction-table (PIT) pipeline which we denote \( P_I \).

Specifically, Magellan computes \( P_I \) in two passes. In the first pass, it conducts a symbolic program analysis of each instruction to identify whether it has a compact map (CM) representation, due to either instruction structure or system state constraints, without actually executing the instruction. See Table 2 for a preview of such instructions. Let \( P(CM) \) be the set of such identified instructions and \( P(CM)^\text{rem} \) the remaining. Consider the Example program. Symbolic analysis will identify that all but \( L_{11} \) in \( P(CM) \). Figure 5 shows the result after the analysis.

![Figure 5: Instruction processed by symbolic map.](image)

Although the symbolic pass makes much progress, it leaves two problems unaddressed, as shown in Figure 5:

- First, \( P(CM) \) may not be empty. Consider Example 1. One can see that \( P(CM) \) contains \( L_{11} \) and hence the analysis cannot determine the flow table of \( L_{11} \). We refer to this as the NCM-table-not-defined problem.

- Second, consider the table for \( L_4 \). One can see that the table contains flow rules for ports 22, 53, and 80, but these rules should be pruned by the preceding instruction \( L_1 \) (dstPort < 1025). Considering each instruction in isolation, the symbolic-map pass cannot solve this problem. We refer to this as the CM-dead-entries problem.

Solving both problems requires essentially computing the set of valid values of inputs to each instruction \( I \), which we denote as \( I.inVals \). With \( I.inVals \) for \( I \) in \( P(CM) \), one can then execute \( I \) on each item in \( I.inVals \) as long \( I.inVals \) is not too large, obtaining the flow table of \( I \); with \( I.inVals \) for \( I \) in \( P(CM) \), one can prune the dead rules, solving the second problem.

Computing \( I.inVals \) for each instruction \( I \), however, is not straightforward and has requirements to address.

- First, the items in \( I.inVals \) can come from multiple execution paths. Consider \( L_{11} \), which uses input variable dstSw. From Figure 5 one can see that one path providing dstSw to \( L_{11} \) is from \( L_2 \) (Sw_ifW) and the other \( L_5 \) (hostTable lookup). We refer to this as the multi-path collection requirement. An algorithm that cannot handle multi-path collection may lead to missing valid inputs.

- Second, consider an instruction \( I \) with multiple inputs. A naive approach is to compute the valid inputs to each input variable individually. This, however, can lead to unnecessary combination explosion. Still consider \( L_{11} \). Suppose...
that both hostTable and condTable have 1000 entries for a given set M of 1000 MAC addresses, mapping to 100 switches and 100 conditions, respectively. Assume that one uses the results of symbolic analysis on $L_5$ and $L_6$ directly to process $L_{11}$. Then the combined inputs for dstSw and dstCond can lead to 100x100 combinations. However, the maximum number of combinations of dstSw and dstCond is only 1000, one for each MAC in M. We refer to this as the dependent-path-detection requirement. An algorithm that cannot handle dependent paths may lead to unnecessary combination explosion, resulting in low scalability.

In the second pass of computing CP, the novel FLOWEXPLORE algorithm (Section 4.2) solves the NCM-table-not-defined and CM-dead-entries problems. The algorithm satisfies both requirements by using two simple, novel ideas.

- First, recognizing that the most compact flow table for I depends on state, which is a valid binding of variables right before the execution of I, the FLOWEXPLORE algorithm computes flow tables for (I, state) pairs, and aggregates them to obtain the final flow table for each I.
- By considering state, the algorithm removes dead-entries of an existing CM table, as they will not be allowed by any valid state. For a non-CM statement, the states provide valid inputs, allowing Magellan to compute its flow table.

- Second, one concern of introducing state is potential state explosion. However, recognizing that the flow tables of instructions in $P(\text{CM})$ already provide high aggregation of states with compact representation, the FLOWEXPLORE algorithm takes advantage of it and uses the flow rules of those instructions to propagate states, substantially reducing the number of states to explore.

**Outcome:** To summarize, after the symbolic-map and flow-explore passes, for each instruction I, Magellan can compute the following set of properties, shown in Table 1:

<table>
<thead>
<tr>
<th>Property</th>
<th>Computed By</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.inVars</td>
<td>Given</td>
<td>Input variables of I</td>
</tr>
<tr>
<td>I.outVar</td>
<td>Given</td>
<td>Output variable of I</td>
</tr>
<tr>
<td>I.next</td>
<td>Topo. sort</td>
<td>Ins. executed after I</td>
</tr>
<tr>
<td>I.type</td>
<td>Map, exec. guard</td>
<td>Type of instruction, CM, ...</td>
</tr>
<tr>
<td>I.CMTable</td>
<td>Map</td>
<td>Table constructed by CM analysis</td>
</tr>
<tr>
<td>I.table</td>
<td>Explore</td>
<td>Final table</td>
</tr>
<tr>
<td>I.inVals</td>
<td>Explore</td>
<td>Set of valid input values</td>
</tr>
<tr>
<td>I.etRules</td>
<td>Explore</td>
<td>Estimated #rules</td>
</tr>
</tbody>
</table>

Table 1: List of properties for each instruction I.

In addition to the preceding per-instruction outcome, the key outcome of the CP step is the flow-explore graph, which will be used in computing per-instruction I.states and I.table above. More importantly, this data structure will be used in achieving incremental updates (Section 4.3) and also in pipeline design (Section 5.3). Hence, the graph is a key data structure of Magellan.

To aid the design of FLOWEXPLORE, which demands essential dataflow analysis, Magellan converts the program into a new format, with two properties: (1) all control flow dependencies are replaced as data dependencies, by introducing a guard condition for each instruction; (2) all assignments are static-single assignments (SSA) [1], where variables are never assigned to more than once, to ensure that data flow between instructions is nearly trivial to compute.

Figure 6 details the instruction set of this new format.

```
 P ::= {I}  (instruction sequence)
 I ::= I'  (instruction sequence)
 | g : I'  (guarded instruction)
 I' ::= x = y boolop t  (boolean operation instruction)
 | x = y arithop t  (arithmetic operation instruction)
 | x = dbSet.contains(y)  (table contain instruction)
 | x = φ(y1...yn)  (operation parameter)
 | (x1...xn) = f(y1...yn)  (black box instruction)
 | x = ¬x  (datapath instruction)
 t ::= Const | Variable  (operation parameter)
 g ::= x | ¬x  (exec. guard)
```

Figure 6: The IR-CP instruction set.

As an example, Magellan will rewrite Example as the following:

```
// Program: Example-DF (data flow form)
Map condTable(key: macAddress, value: GBP cond)
Map hostTable(key: macAddress, value: sw)
Set openPorts(key: port)
L1: g9 = (p.dstPort < 1025)
L2: if g9: dstSw0 = SW_FW
L3: if g9: dstCond0 = V
L4: if !g9: q1 = openPorts.contains(p.dstPort)
L5: if q1: dstSw1 = hostTable[p.dstMac]
L6: if q1: dstCond1 = condTable[p.dstMac]
L7: if q1: egress0 = Drop
L8: dstSw2 = phi(dstSw0, dstSw1)
L9: dstCond2 = phi(dstCond0, dstCond1)
L10: srcSw0 = hostTable[p.srcMac]
L11: egress1 = f(dstCond2, dstSw2, srcSw0)
L12: return = phi(egress0, egress1)
```

The φ instruction (also known as a pseudo-instruction in SSA) in the instruction set, which is used in L8 in the Example-DF, is required to place the program in SSA form; $x = \phi(y, z)$ indicates that $x$ could get its value from either $y$ or $z$ depending on the execution path of the IR0 program. The φ function ensures that the correct choice of $y$ or $z$ is chosen in a given execution.

One can represent the instructions in the new format as a dataflow graph, which includes a node for each instruction, a node for each packet field used by the program, and a node for the variable holding the egress value (return value), and a link between two nodes when the value produced by one can flow to the other in some executions. Figure 7 is the dataflow representation of Example-CP.

3.3 From $P_I$ to Practical, Optimized Pipeline ($P_O$)

**Problem:** Although the $P_I$ step has made substantial progress, for example, it has converted a program with diverse instructions into a uniform format (a pipeline with a compact flow
identifies an optimized, hardware constraint conforming pipeline, however, can have two problems.

- First, per-instruction tables may not satisfy hardware constraints. For example, a typical hardware limits the number of flow tables to a relatively small number to achieve high speed. The number of per-instruction tables can well exceed this threshold, resulting in hardware infeasibility. This is referred to as the hardware constraint problem.

- Second, even without hardware constraints, per-instruction tables may miss resource usage optimization opportunities and hence can be suboptimal. Consider instructions $L_5$ and $L_6$ of Example 1. Both instructions lookup on p.dstMac. These separate instructions are natural in structured (modular) programming or modular data structures. Implementing these two instructions as two different pipeline tables, however, is suboptimal, because they then use their own copies of matching entries, while a single pipeline table uses shared match resources, and returns both outcomes, reducing the number of lookup times from 2 to 1 and cutting the matching resource use to half. We refer to this as the resource sharing optimization problem.

**Approach:** Solving both problems leads to aggregation of per-instruction tables: for the hardware constraint problem, aggregation may increase hardware resource consumption but is needed to satisfy hardware constraints; for the pipeline resource sharing optimization problem, aggregation of per-instruction tables reuses hardware matching resources, leading to a better design.

Conducting a systematic aggregation evaluation to identify an optimized, constraint conforming pipeline, however, can be expensive. A naive approach is to simply enumerate all aggregations and find the best. Let $N$ be the number of tables in $P_I$, and assume a simple hardware constraint that the number of hardware tables cannot exceed $M$. Then the naive approach will enumerate $M^N$ cases (where each per-instruction table is assigned to one of the $M$ pipeline tables) to pick the best. This can be expensive.

The novel DeepExpansion pipeline-design algorithm (Section 5.1) avoids naive, non-structured pipeline design by taking advantage of a domain insight and program structure to achieve systematic, efficient exploration of pipeline design. Specifically, the domain insight is that although $P_I$ can have a large number of instructions, the number of raw inputs will be few. In particular, a typical program may use only a few packet attributes as inputs. After all, the total number of packet attributes is limited to a small number. We refer to this as the thin input domain insight. Next, consider a set of input variables and assume that they are the match fields of an already-determined hardware pipeline table. Assume that we know the exact value of each variable in this set, then it helps to identify as many determined data as possible. We refer to this as the deep expansion strategy.

**Outcome:** The DeepExpansion pipeline-design algorithm identifies an optimized, hardware constraint conforming pipeline, conducts a simple, novel cross-product merge algorithm to aggregate each set of per-instruction tables into a hardware pipeline table. By adjusting the cost metric of the pipeline design process, DeepExpansion achieves optimized, stable, practical pipeline design and implementation.

4. FROM THEORETICAL PIPELINE TO COMPACT PIPELINE

We now give the details of computing the per-instruction compact pipeline. As we discussed in Section 3, it consists of two passes: the first symbolic-map pass and the second flow-explore pass, which we present in Section 4.1 and 4.2 respectively. Then, we show how the resulting data structure (the explorer graph) is used in table generation and incremental updates in Section 4.3.

4.1 Symbolic Map

Given a program $P$, Magellan conducts symbolic analysis to identify a wide range of instructions that can be mapped into compact forms. For each such identified instruction $I$, $I.type$ is marked as CM, and its flow table representation is computed. Table 2 shows the details of the identified instructions, their compact flow table representations, and size estimation of its table size. One can see that the CM instructions are clarified into multiple types. For example, the first type is named BoolConst. These types will be used in Section 5 for optimization.

Note that compact representation may not always be possible, as we have the following negative result:

**Proposition 1.** Computing the minimum size of a table representing an instruction sequence is NP-hard.

For arithmetic operations, we have the following proposition stating a negative result limiting the capabilities of flow tables:

**Proposition 2.** Consider instruction $\text{return } p.macDst \% n$, where $n \neq 2^i$ for any $i$. Any compact TCAM representation of this instruction uses $O\left(\frac{n-1}{n}2^{48}\right)$ flow rules.

For such fundamentally challenging statements, Magellan falls back to reactive handling of punted packets to determine input variable values and rules for the given statement. This will become clear in Section 4.2.

4.2 Flow Explore (FlowExplore)

As we discussed in Section 3.2, the symbolic-map pass leaves two problems for the FlowExplore algorithm to solve: (1) compute flow tables for non-CM type instructions and (2) remove dead-entries for flow tables constructed for CM type instructions in isolation.

**Basic ideas:** The FlowExplore algorithm solves both problems by computing valid state reaching each instruction $I$, where a valid state is a valid binding of variables right before the execution of $I$. For a non-CM instruction, state provides valid inputs (instead of all potential values of the
Table 2: Compact (flow table) representations of statements. \(w\) is bit-width of involved variables.

domain of each input variable of \(J\); for an CM instruction already with a table \(\text{I.CMTable}\), the \(\text{state}\) provides checking on whether a given flow rule entry is possible. We use \text{Example-2} below, which is a shortened version of \text{Example-1}, to illustrate the basic ideas of \text{FLOWEXPLORE}. Hence, in this section, all example instructions are from \text{Example-2}.

// Program: \text{Example-2}
Map hostTable(key: macAddress, value: sw)
L0: onPacket(p):
L1: dstSw = hostTable[p.dstMac]
L2: dstCond = condTable[p.dstMac]
L3: egress = myRouteAlg(dstSw, dstCond)

Valid states for instructions early in the computation can be computed relatively easily. One can see that there can be only one state (empty binding) reaching \(L_1\), which we represent as \((L_1; \emptyset)\).

Computing valid states reaching instructions later in the computation, however, can become more complex. Consider \(L_2\). If Magellan execute instructions in a data-flow based order, \(L_2\) can be executed in parallel with \(L_1\), and only one state (empty binding) reaches \(L_2\). If, on the other hand, execution proceeds in program instruction order, \(L_2\)’s state depends on \(L_1\)’s execution result. To simplify understanding, we present our algorithm using serial execution.

Consider the execution of \((L_1; \emptyset)\) when calculating valid states at \(L_2\). A key issue is that the state of \(L_1\) does not provide any binding to \(p\text{.dstMac}\). \(L_1\)’s input variable. If we treat \(L_1\) as a black box, enumerating \(p\text{.dstMac}\) requires us to enumerate all \(2^4\) possible bindings - clearly too many. However, recognize that \(L_1\text{.type} = \text{CM}\), which implies that the symbolic map pass has computed \(L_1\text{.CMTable}\) with a small number of bindings (we can consider each rule as a binding) to \(L_1\)’s input and output variable. Hence we can apply the flow rule in \(L_1\text{.CMTable}\) one by one to derive one new state reaching \(L_2\) one by one. For example, see \((L_1; \emptyset)\), shown at the top of Figure 8. It has 4 outgoing edges, each derived by one flow rule of \(L_1\text{.CMTable}\).

A naive construction of \(L_1\text{.next state}\), for a given flow rule, is to only add the flow rule’s outcome to the current state. We refer to this as the outcome branching approach. This, however, does not fully utilize all information. Consider how Magellan applies the first rule of \(L_1\text{.CMTable}\) in Figure 8 to \((L_1; \emptyset)\). As shown, it adds both its outcome \(g_0\) and its input binding \(p\text{.dstMac}\), further constraining \(\text{state}\). We refer to this as the full-flow-rule edge approach. Note that \(L_1\)’s last rule aggregates \(p\text{.dstMac}\) remaining bindings (*), avoiding full enumeration.

Using only flow-rules without using the binding in \(\text{state}\) leaves valid information not used. Consider the leftmost \(L_2\) node, labeled \(A\) in Figure 8. To derive the states for \(L_3\) from \(A\). Magellan applies the flow rules in \(L_2\text{.CMTable}\). Consider applying the second rule, which has a match field \((p\text{.dstMac} = 22)\). This conflicts with the binding in \(\text{state}\) \((p\text{.dstMac} = 11)\). Figure 8 shows that such branching will be excluded from \(G_6\).

We also use \text{state} to constrain a rule’s match fields by replacing them by their intersection with \text{state}’s variable bindings. For example, if a rule \(r\)’s \(p\text{.dstMac}\) match field was \(x1\times1\) and \(\text{state}\) binds \(p\text{.dstMac}\) to \(x0\times0\), we set the rules match field at that node’s edge to their intersection, \(x01001\), since only that intersection will trigger \(r\).
The preceding is almost complete except one issue: using all flow rules may not be necessary. Consider applying the third rule of $L_2$ to node $A$. Although the intersection between the state and the flow rule has no conflict ($p.dstMac = 11 \land p.dstMac = * \rightarrow p.dstMac = 11$), this branching is unnecessary, because the first child of $A$, labeled $B$ in the figure, has already explored the state $p.dstMac = 11$. We say that the third rule is blocked. Blocking typically happens for default flow rules, and we can identify blocked rules with the condition below:

**Lemma 1.** Rule $q$ blocks $r$ if $r.match \land state \subseteq q.match \land state$ and $q$ is processed before either $r$ in $I.cmTable$ or the rule to the current node in an ancestor node.

**Mergable instructions:** A naive FLOWEXPLORE algorithm considers each instruction $I \in P$ individually. This, however, is unnecessary. An instruction $J$ is mergable into its parent instruction $I$ if the optimal pipeline executes both instructions in the same table. Identifying mergable instructions can reduce exploration for a real program, which can have a large number of instructions. Magellan identify mergable instructions with the condition below, and assume that such merging has been processed.

**Lemma 2.** An instruction $J$ is mergable into its parent $I$ if $J$ only matches on variables assigned by $I$.

**The FLOWEXPLORE algorithm:** With the preceding background, we now show FLOWEXPLORE algorithm (Algorithm 1). We present FLOWEXPLORE as a recursive function that calls on a given $(I, state)$. The first FLOWEXPLORE call starts with the first instruction with an empty state. We highlight a few key steps. For a $CM$ instruction (line 4), conflicted and blocked rules are removed at lines 6 and 8 respectively; for non-$CM$ (line 10), FLOWEXPLORE computes state’s number of potential bindings to $I$’s match fields (line 11). If too many, FLOWEXPLORE labels the instruction as SAMPLE, to handle instructions such as that in Proposition 2; otherwise, FLOWEXPLORE enumerates each binding in turn (line 12). FLOWEXPLORE removes dead variables from state (line 20) to better identify such execution path merges, and recurses only if a $(I, state)$ pair has not already been explored (line 23).

### 4.3 Graph to Tables and Updates

**From $G_E$ to $P$’s flow table:** Given $G_E$, computing each instruction $I$’s flow table is straightforward. Let $(I, state).R$

\[
I.table = \text{FUNION}\{(I, state).R : (I, state) \in G_E\}
\]

(1)

where FUNION is a simple function that computes a union of a set of flow rules, eliminating duplicates. Rules not in $(I, state).R$ are never executed and can be safely pruned.

**Incremental updates:** In addition to allowing generating compact flow tables, the flow-explore graph records existing computation to allow fast incremental updates. Suppose that a database table $dbT$ has undergone a change - for addition of a new rule $r$. To update, Magellan simply adds $r$ to the corresponding explorer graph nodes, explores it, removes any edges it blocks, and then updates flow tables with a record of $G_E$ edges added and removed.

### 5. FROM COMPACT PIPELINE TO PRACTICAL, OPTIMIZED PIPELINE

We now give the details of the DEEPEXPANSION pipeline design algorithm. We first present the algorithm framework in Section 5.1, and then give analysis in Section 5.2.

#### 5.1 Pipeline Design (DEEPEXPANSION)

As we discussed in Section 3.3, the pipeline-design algo-
Basic ideas: A main issue that the pipeline design algorithm needs to handle is diverse hardware resource constraints. For example, our survey finds constraints on diverse metrics such as the total number of tables, the depth of the longest dataflow, the sum of a subset of flow tables (e.g., [6] limits the sum of every 4 flow tables). To accommodate such diversity, we design the pipeline design algorithm to be a framework, parameterized by a hardware constraint oracle (oracle), which takes a pipeline specification (π) and returns whether the pipeline is acceptable. Let II be the set of potential pipelines that the pipeline-design algorithm queries the oracle. Let resource be another parameterized function calculating the resource usage of a pipeline, our algorithm identifies π ∈ II, where oracle(π) == true and resource(π) is minimal.

Hence, a main challenge then is to identify II. As we discussed in Section 3.3, a naive approach is simple enumeration where all potential designs are enumerated, but this is not scalable. The smaller the set II, the more efficient the algorithm. On the other hand, if II is too small, it may miss good pipeline designs, leading to a poor final pipeline.

A key insight of our pipeline-design algorithm is to consider the pipeline-design as a partition problem. Let V be the set of instructions. Then a pipeline is a partition of the instructions into subsets. Consider that one is already given some subsets where each subset is mapped to a pipeline table. The bigger one can extend these subsets to include remaining instructions, the less instructions left and hence less number of possibilities to partition the remaining instructions. Consider instructions Li and Lj. Assume that Li is in an initial given subset and we can guarantee that adding Lj to the same subset will not miss good pipeline designs, then we can do so, reducing the number of remaining instructions to handle and hence less potential partitions.

Applying the preceding insight leads to the following deep input expansion strategy. In particular, consider a subset Vi of V with the set of external input Vj.eInputs, where an external input is one that does not come from another instruction also in Vi. Assume we already have a pipeline table for Vi. Consider an instruction Lj not in Vi, but the inputs of Lj are subset of Vj.eInputs ∪ Vi.outcomes, where Vi.outcomes represents all variables that instructions in Vi assign to. Then, we can include Lj in Vi without increasing the size of table constructed for Vi.

The preceding is largely correct except one issue: wildcard rules can result in counter examples. Consider L1 : y1 = dstMac&011 which takes the last two bits of dstMac and L2 takes the first 2 bits. Since both are determined by dstMac. One may think that it is OK to always merge L1 and L2. Such merging, unfortunately, can lead to more rules (4 + 4 vs 16):

L1: dstMac y1 L2: dstMac y2 L12: dstMac y1 y2
xxxx00 00 00xxxx 00 00xxxx 00 00

Fortunately, by considering both an input variable and how it is used, we can avoid the preceding issue. How a variable is used can be obtained from symbolic analysis. In particular, CM instructions in Table 2 are organized into groups, where each group has a type, such as BoolConst. If an input variable is not in Native or WildCard, its type is that from Table 2; otherwise, the type is the variable itself. Input variables now are considered as (variable, type) pairs. One can see that according to this rule, the preceding counter example will not happen. Specifically, we have

**Proposition 3.** Consider each variable by both its name and type. If Vi.eInputs ⊆ (Vj.eInputs ∪ Vj.outcomes), we can merge Vi into Vj without increasing the number of rules.

Algorithm: With the preceding basic ideas, we now show the DeepExpansion algorithm (Algorithm 2), which is a recursive algorithm. The algorithm takes two parameters: (1) Vr, which is the set of remaining instructions; and (2) m, which is a table number of the next table to be generated. The first call to DeepExpansion starts with the set containing all instruction and m = 1.

We now highlight a few key points of the algorithm. Line 1 considers all instructions in Vr as a single table and verifies that it is acceptable by hardware constraints, and if so, whether it will be better than the optimal seen so far. Then, for each proper subset S of Vr.eInputs, it computes all variables that can be merged (line 8 is the Proposition 3 condition) and merge them as the m-th table. The algorithm recurses as these instructions are removed (line 12, called with Vr \ Vm and the table number increased by 1).

```
Algorithm 2 DeepExpansion(Vr, m)
1: πc ← π(V1,...,Vm−1,Vr)
2: if oracle(πc) and resource(πc) < resource(πm) then
3: πm ← πc □ record the best pipeline
4: for all S ⊆ Vr.eInputs do
5: Vm ← ∅
6: repeat
7: for each v ∈ (Vr \ Vm) do
8: if v.eInputs ⊆ (S ∪ Vm.outcomes) then
9: Vm ← Vm ∪ {v}
10: until no update for Vm
11: if Vm.eInputs = S and oracle(π(V1,...,Vm−1)) then
12: DeepExpansion(Vr \ Vm, m + 1)
```

![Figure 9: Deep Expansion's Example pipeline](image-url)
Example: We now illustrate **DEEP**\textit{EXPANSION} by applying it to Example under the hardware constraint that the number of pipeline tables \( M \leq 4 \) to produce the \( \pi \) shown in Figure 9. Note that types of inputs are not labeled in the graph. In Example, only one input has type \texttt{BoolConst} and others are \texttt{State}. Hence, we skip \texttt{State} in the figure.

The algorithm starts with \( V_r = V = \{ L_1, L_2, ..., L_{11}, L_{12} \} \), \( m = 1 \). The algorithm computes \( V_r.e\text{Inputs} = \{ (p.d\text{stPort}, \text{State}), (p.d\text{stPort}, \text{BoolConst}), (p.d\text{stMac}, \text{State}), (p.d\text{stMac}, \text{State}) \} \), and enumerates each of its proper subset \( S \). For example, when \( S = \{ (p.d\text{stPort}, \text{State}), (p.d\text{stPort}, \text{BoolConst}) \} \), it obtains \( V_1 \), which includes \( L_1 \) to \( L_4 \) and \( L_7 \). \( V_1 \) is shown as \( T_1 \) in Figure 9. Removing these instructions, \( V_r \) becomes \( \{ L_5, L_6, L_8, L_9, L_{10}, L_{11}, L_{12} \} \) and \( V_r.e\text{Inputs} = \{ (\text{dstSw0}, \text{State}), (\text{dstCond0}, \text{State}), (\text{gI}, \text{State}), (\text{dstMac}, \text{State}), (\text{dstMac}, \text{State}) \} \). The algorithm continues.

Choose resource for optimization/stability: Magellan’s framework allows for multiple possible resource functions. As a default resource estimates the \( \pi \)’s memory footprint, however, the user can also supply a custom function. For an individual table \( V_i \) in a \( \pi \), \( V_i \)’s size is estimated by multiplying \( V_i \)’s total match field length (the sum of \( V_i \)’s external inputs) with \( V_i \)’s rule flow number, found by taking the cross product \( (xprod(V_i)) \) of its constituent tables calculated in \( F_I \) (if a \( F_I \) table was unexplored, the user can annotate to provide hints), as shown below:

\[
\text{resource}(V_1 ... V_m) = \sum_{i=1}^{m} (xprod(V_i) \sum_{v \in V_i.e\text{Inputs}} w(v))
\]

5.2 Analysis

Run-time complexity: **DEEP**\textit{EXPANSION}’s run time complexity is simply the product of the number of times **DEEP**\textit{EXPANSION} is called and the complexity of each call. To calculate **DEEP**\textit{EXPANSION}’s number of calls, we note that it is called is once per search tree node. If \( Q \) is the maximum number of \( V_i.e\text{Inputs} \), a node can have \( O(2^Q) \) children (one for each subset) and thus \texttt{FLOWEXPLORE} can be called up to \( O(2^{M \times Q}) \) times, where \( M \) is the maximum pipeline depth. Each **DEEP**\textit{EXPANSION} execution has complexity \( O(Q \times |V_i|) \), which is dominated by calculating \( V_m.e\text{Inputs} \) and traversing \( V_r \) to determine \( V_m \). Hence, the total complexity is \( O \left( Q \times |V| \times 2^{M \times Q} \right) \). Although exponential, in most real cases \( Q \) is small.

6. EVALUATIONS

We conduct extensive evaluations on the benefits and scalability of Magellan.

6.1 Methodology

SDN programs: We use three high-level SDN programs in addition to Example: GBP, Switch.p4, and Interdomain. The programs span contexts from a single switch (Switch.p4), to data-center (GBP), to interdomain (Interdomain). The derivation of the programs and their functions are detailed in Table 3. GBP and Interdomain are implemented in Java, and Switch.p4 is specified as a per-instruction table \( F_I \).

<table>
<thead>
<tr>
<th>Program</th>
<th>Derivation and functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBP</td>
<td>Derived from GBP implementation in OpenDaylight. Provides security routing in data centers using the GBP model</td>
</tr>
<tr>
<td>Interdomain</td>
<td>Derived from [3]. Provides no valley routing, WAN optimization, QoS routing, and app aware routing</td>
</tr>
</tbody>
</table>

Table 3: Evaluated high-level programs.

Network topologies: We use network topologies as a key system state and evaluate three topologies: Complete, Linear, and ISP. Table 8 gives their base statistics. Later when we vary their sizes in stability evaluations, we give the specific sizes.

<table>
<thead>
<tr>
<th>Network</th>
<th>#Switch</th>
<th>#Host</th>
<th>#Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>9</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>Linear</td>
<td>5</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>ISP</td>
<td>152</td>
<td>1520</td>
<td>1715</td>
</tr>
</tbody>
</table>

Table 4: Evaluated base network topologies.

Alternative controllers: During our end-to-end evaluations, we use multiple state-of-the-art commercial and academic SDN controllers, including OpenDaylight [8] (Helium release), Floodlight [8] (version 1.0), Maple [27] (version 0.10.0), POX [21] (\texttt{forwarding.12\_learning} module from 0.2.0), and Pyretic [17] (latest version). Maple, POX, and Pyretic are academic systems supporting novel policy languages and compilers, while Floodlight and OpenDaylight 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.

6.2 Impact on Pipeline Properties

We start by showing the impact of Magellan on pipeline properties, in particular the number of tables and the total number of flow rules. Table 5 shows the result, for the Complete topology. Note that there are no hardware constraints for all except the last column results.

<table>
<thead>
<tr>
<th>Program</th>
<th>#tables</th>
<th>#rules</th>
<th>#rules</th>
<th>#rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mag.</td>
<td>PIT</td>
<td>Mag.</td>
<td>PIT</td>
</tr>
<tr>
<td>Example</td>
<td>12</td>
<td>4</td>
<td>24</td>
<td>34</td>
</tr>
<tr>
<td>GBP</td>
<td>13</td>
<td>13</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>Interdomain</td>
<td>18</td>
<td>7</td>
<td>1354</td>
<td>494</td>
</tr>
<tr>
<td>Switch.p4.L2</td>
<td>129</td>
<td>4</td>
<td>270</td>
<td>147</td>
</tr>
<tr>
<td>Switch.p4.L3Vxlan</td>
<td>129</td>
<td>6</td>
<td>172</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 5: Pipeline properties under Magellan and PIT.

We make three observations. First, consider Switch.p4, which evaluates the ability of Magellan to identify pipeline resource sharing optimization, when a PIT pipeline is given. We observe that for both cases (L2 and L3Vxlan) Magellan reduces the number of tables by a factor of \( >20x \) (e.g., 129 to 4 and 6) and the number of flow rules by a factor of \( 2x \). Magellan achieves these substantial reductions by discovering resource sharing among tables, and pruning functional-
ity unused. Second, consider Example, GBP, and Interdomain, which are not given PIT. We can see that Magellan can compute a compact per-instruction pipeline. For Interdomain, resource sharing optimization reduces the number of tables from 18 to 7, a factor of 2.5x, and the number of flow rules from 1354 to 494, a factor of 2.7x. Interestingly, the GBP PIT pine is already highly optimized. Third, compare the number of flow rules when the hardware supports only 1 table and that when there are no constraints. We see that GBP has a factor 777x (171,800/222) difference and Interdomain has a factor 155x difference (76,800/494). This clearly shows the benefit of identifying pipelines. Existing approaches that cannot compute pipelines and hence can use only a single flow table can have large performance disadvantage. Note that the Switch.p4 program uses specialized instructions and hence cannot be mapped to only one table.

### Table 6: Pipeline #rules at given pipeline size constraint.

<table>
<thead>
<tr>
<th>System</th>
<th>#Tables</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBP</td>
<td>171,800</td>
<td>23,490</td>
<td>7,495</td>
<td>4,164</td>
<td>1,345</td>
<td></td>
</tr>
<tr>
<td>Interdomain</td>
<td>76,800</td>
<td>5,191</td>
<td>5,042</td>
<td>664</td>
<td>515</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>56</td>
<td>24</td>
<td>24</td>
<td>17</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

Going beyond the 1-table constraint, we next vary the allowed number of tables as an example of varying hardware constraints. Table 6 shows the result. The pipeline-design algorithm of Magellan allows it to compute optimized pipelines at these different settings. Consider Interdomain as an example. We can see a large reduction when the number of allowed tables increases from 1 to 2 (#rules reduces from 76,800 to 5,191) and from 3 to 4 (#rules reduces from 5042 to 664). Hence, it may be beneficial to have either 2 or 4 tables, but not 3, since gain from 2 to 3 is small. Hence, Magellan can serve as a tool for evaluating hardware impacts and choosing the proper hardware.

### 6.3 Impact on End-to-end Performance

It helps to see the benefit of Magellan in an end-to-end setting. To conduct this evaluation, we evaluate the controllers specified in Section 6.1, and configure them to use 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, $H$, attached to a switch, with each host attached to a distinct port.

A challenge in conducting this evaluation is to identify a policy that is available in all systems. Hence, we choose L2 learning and routing. Although not ideal, it appears to be the only common one. For each controller, after allowing appropriate initialization of hosts and controller, we perform an all-to-all ping among the hosts, record the RTT of each ping, and measure the time for all hosts to complete this task.

Figure 10 charts the median ping RTTs for each system, for each system with $H = 70$ and $H = 140$. We observe that Magellan achieves the lowest finishing time.

To understand the gain of Magellan, consider two factors. First, Magellan is an incremental, proactive system, in that as soon as a state change happens (i.e., host location learned in this case), it will use the incremental algorithm in Section 4.3 to fully compute the whole effects. In contrast, all other controllers (except Pyretic) generate rules only when a sender sends a first packet to a receiver, and hence these systems continually incur flow table misses throughout the task. Second, Magellan leverages multi-table pipelines and hence has fewer rules to generate. Table 7 lists the details on the number of rules, task completion time, and median ping RTT.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 $2 \times H$ rules.

### 6.4 Scalability and Stability

One key concern of Magellan is its scalability and stability. To evaluate this aspect, we vary system topology size, to observe the effect. Table 8 shows the results for GBP and Interdomain, and for scaling of two types of networks. We observe that as we increase the topology size, both programs can maintain the stability of pipeline designed and run time remains nearly constant.

One key concern of Magellan is its scalability and stability. To evaluate this aspect, we vary system state, in particular the topology size, to observe the effect. Table 8 shows the results for GBP and Interdomain, and for scaling of two types of networks. We observe that as we increase the topology size, both programs can maintain stability of pipeline design and run time.

1Tests of Maple at 140 hosts and of Pyretic at both 70 and 140 hosts failed and these measurements are therefore omitted.
7. RELATED WORK

High-level programming models: A design decision of Magellan is the high-level SDN programming model to use. We classify existing models into two categories: “tierless” and split-level. In a “tierless” model (e.g., OBS [2], FML [13], FlowLog [18], Maple [27]), programmers specify forwarding behaviors as a packet handling function which has access to and can update state components (i.e., control) state components which are not accessible from network elements themselves. In a split-level model, in particular, the Frenetic family of languages (Frenetic [9] and Pyretic [17]), a two-level programming model is used, where a controller program specifies events of interest, and responds to these events by calculating a new network policy, which is expressed in a network policy language that describes stateless forwarding behavior. A network policy compiler compiles a given network policy to flow table rules. Magellan chooses a tierless model to show that it can handle both state access and forwarding behavior specification. Recent work such as SNAP [2] considers distributing states to switches (i.e., persistent states at switches). This paper focuses on detecting state updates, and one may consider pipeline design allowing switch local updates (e.g., protect race condition).

Low-level SDN control systems: The main mechanism currently available to SDN programmers to manage pipelines is to use lower-level SDN control systems and APIs (e.g., [7, 8, 12, 15, 19]). In particular, NOX [12] offers C++ and Python APIs for raw event handling and switch control, while Beacon [7], Floodlight [8] and OpenDaylight [19] 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.

Pipeline specification: The importance of pipelines has motivated researchers to investigate how to specify such hardware. P4 [4] is a system that provides three languages for specifying (1) packet field parsers, (2) a collection of flow tables, and (3) a processing pipeline among the flow tables. PISCES [24] defines a software switch by a protocol-independent, domain specific language. In [23], a typed programming language called Concurrent NetCore is proposed to specify flow tables, and systematic application of type theory to pipeline specification can have great values. The key difference between Magellan and such previous work is that they focus on pipeline specification, and Magellan focuses on deriving, populating, and updating pipelines from high-level, pipeline-oblivious programs. Magellan can work with such systems by using their specifications as output.

Pipeline design: There are some previous studies which can be considered as pipeline design, and hence highly related. In [16], the authors propose an elegant approach to decompose a flow table with complex matching conditions to generate a multi-table pipeline for high-performance realization of the given flow table. Hence, the setting of their system and ours are different: their input is flow tables and our input is high-level programs and system states. DIFANE [28] divides a single table among multiple switches, and hence has a different setting as well (e.g., one cannot pass register values among switches). In [25], the authors design a system where each low-level instruction is mapped to a hardware processing unit. This has similarity to our intermediate PIT design. Our FLOWEXPLORE algorithm goes beyond PIT to compute and update flow table contents and our DEEPEXPANSION algorithm computes optimized pipelines. The work by Jose et al [14] is highly related with our DEEPEXPANSION algorithm. The authors design an interesting integer programming algorithm and consider real constraints from devices including RMT and Intel’s FlexPipe. A key difference between their design and our pipeline design is that they take P4 layout as input and ours takes advantage of the content computed by the FLOWEXPLORE algorithm and uses a generic framework to handle hardware diversity, and optimize for the state while also consider stability.

<table>
<thead>
<tr>
<th>Network # sw / # hosts</th>
<th># Rules</th>
<th># Tables</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program: Interdomain; Network: ISP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 / 250</td>
<td>4868</td>
<td>7</td>
<td>461</td>
</tr>
<tr>
<td>60 / 360</td>
<td>5828</td>
<td>7</td>
<td>429</td>
</tr>
<tr>
<td>80 / 640</td>
<td>10308</td>
<td>7</td>
<td>380</td>
</tr>
<tr>
<td>100 / 1530</td>
<td>16068</td>
<td>7</td>
<td>362</td>
</tr>
<tr>
<td>153 / 3060</td>
<td>44156</td>
<td>7</td>
<td>404</td>
</tr>
</tbody>
</table>

Table 8: Pipeline design scalability and stability as network topology size varies.

designed and run time remains nearly constant.

One key component that helps with the scalability of Magellan is its efficient pipeline design algorithm (DEEPEXPANSION). To demonstrate the benefit of DEEPEXPANSION, we contrast the number of pipelines that DEEPEXPANSION needs to evaluate vs those by two other approaches (full enumeration and enumeration considering DAG constraints). We use a variant of Example accessing 3 packet attributes and consider the number of instructions from 5 to 10, and the hardware limits the number of flow tables to be 5. Table 9 shows the results. We can see that naive enumerations need $M^N$ iterations. Enumerations considering DAG constraints have performed better than naive. DEEPEXPANSION has a substantially lower number of enumerations. For example, for $N = 10$, it evaluates only 14 pipelines, compared with 1061 considering DAG constraints, and 78125 for naive.

<table>
<thead>
<tr>
<th>Alg</th>
<th>#Inst.</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEEPEXPANSION</td>
<td></td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>DAG constraints</td>
<td></td>
<td>16</td>
<td>41</td>
<td>94</td>
<td>227</td>
<td>472</td>
<td>1,061</td>
</tr>
<tr>
<td>Full</td>
<td></td>
<td>25</td>
<td>125</td>
<td>625</td>
<td>3,125</td>
<td>15,625</td>
<td>78,125</td>
</tr>
</tbody>
</table>

Table 9: #pipelines evaluated.

7. RELATED WORK
8. REFERENCES


