Hindawi
Scientific Programming
Volume 2018, Article ID 2848232, 13 pages
https://doi.org/10.1155/2018/2848232

Research Article

SDN Programming for Heterogeneous Switches with Flow Table Pipelining

Junchang Wang, Shaojin Cheng, and Xiong Fu

Department of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China

Correspondence should be addressed to Junchang Wang; junchangwang@gmail.com

Received 9 March 2018; Revised 3 November 2018; Accepted 6 November 2018; Published 21 November 2018

Academic Editor: Emiliano Tramontana

High-level programming is one of the critical building blocks of the effective use of software-defined networking (SDN). Existing solutions, however, either (1) cannot utilize the state-of-the-art switches with flow table pipelining, a key technique to prevent flow rule set explosion or (2) force programmers to manually organize and manage hardware flow table pipelines, which is time-consuming and error-prone. This paper presents a high-level SDN programming framework to address these issues. The framework can automatically (1) generate rule sets for heterogeneous switches with different flow table pipelining designs and (2) update installed rules when the network state changes. As a result, the framework can not only generate efficient rule sets for switches but also provide programmers a centralized, intuitive, and hence easy-to-use programming API. Experiments show that the framework can generate compact rule sets that are 29–116 times smaller than those generated by other open-source SDN controllers. Besides, the framework is 5 times faster to recover from network link failures in comparison to other controllers.

1. Introduction

Software-defined networking (SDN) brings the appeal to manage complex networks more efficiently by shielding low-level details (e.g., setting up flow tables and handling network events such as link failures). In recent years, SDN is becoming more commonplace in scenarios such as high-performance computing and data centers [1–4].

To that end, a major research direction of SDN is to provide more sophisticated SDN programming interfaces, which we refer to as *high-level SDN programming* APIs. By using these APIs, SDN programmers can manage an SDN network in a centralized, intuitive, and hence easy-to-use manner [5–7]. The basic idea of existing programming APIs is to compile SDN programs statically and then to generate a specific flow table scheme for each network switch. We refer to this approach as *proactive high-level SDN programming model* [8] (henceforth simply *proactive programming*). Proactive programming, however, has limitations. Specifically, to manage network switches, programmers have to explicitly specify matching fields of each flow table, and hence it is programmers' responsibility to correctly handle hardware details.

In recent years, another promising approach, *reactive high-level SDN programming model* (*reactive programming* henceforth), has been proposed. *Reactive programming* depends on a runtime optimizer—rather than programmers—to (1) identify switch configurations when an SDN controller starts running, (2) generate flow table scheme for each switch, (3) and populate flow rules when necessary. With these unique features, reactive programming hides switch details and provides programmers a more general and easy-to-use abstraction. A few proactive programming systems [8, 9] have been proposed. In practice, however, we found that utilizing proactive programming can be complex and challenging, and many programming complexities remain. Following are two major challenges.

Firstly, flow table pipelining is a key technique to prevent flow rule set explosion and has been adopted by almost all of the major SDN switch vendors. For example, recent switches from P4 [6], Domino [10], OF-DPA [11], and POF [7] all

support flow table pipelining. Unfortunately, existing implementations of *reactive programming* (e.g., Maple [8]) only generate rule set for switches with a single flow table. As far as we know, there is no implementation of *reactive programming* that can support switches with flow table pipelining yet. The major challenge is that to support flow table pipelining, each single flow rule must be split into multiple stages, and each of which is inserted into a flow table in the pipeline. And hence, smart strategies are required in splitting rules, inserting different stages into different flow tables, and invalidating all stages distributed among flow tables atomically (detailed in Section 3).

Secondly, when *reactive programming* is used, it is the controller's responsibility to automatically recalculate and update installed flow rules when network state (such as topology) changes. This job is nontrivial because multiple flow tables in the pipeline must be updated atomically; otherwise, performance issue or even security issue arises (detailed in Section 2). Unfortunately, no existing implementation of *reactive programming* can handle network state dependencies correctly and efficiently.

Overall, applying *reactive programming* to the state-of-the-art switches that support flow table pipelining is quite challenging. To the best of our knowledge, no existing work has successfully solved the aforementioned issues. In research, it remains an open question *if reactive programming can be applied to the state-of-the-art switches?* And if the answer is yes, *what is the expected benefit?*

This paper tries to answer the above questions by presenting Maple++, the first implementation of high-level SDN reactive programming for switches with flow table pipelining. Maple++ is an extension of Maple [8], a classic reactive programming model. Maple only supports switches with single flow table. In contrast, Maple++ can support switches with flow table pipeline designs, by addressing the aforementioned issues. Specifically, Maple++ introduces *forwarding tree*, a data structure maintained by the controller runtime system to manage forwarding rule sets of switches in a unified and centralized way. Each leaf in the forwarding tree consists of routing decision, dependencies, and handler to environment snapshots. Based on the forwarding tree, a novel tree compression algorithm is invented to remove redundancies. A compressed forwarding tree, which is basically a directed acyclic graph (DAG), can dramatically reduce the number of flow rules generated. To utilize flow table pipeline design in switches, Maple++ splits the compressed forwarding tree to multiple subtrees, according to the configuration of hardware switches. Then, a novel mapping algorithm maps subtrees to multiple flow tables and organizes them as a pipeline. Besides, to handle network events such as link failures, Maple++ includes a novel programmer-oblivious subscription/notification strategy to efficiently handle network events. The strategy not only provides an easy-to-use API to programmers but also helps Maple++ runtime to efficiently and atomically update retired flow rules. As far as we know, Maple++ is the first implementation of reactive programming for real SDN networks including heterogeneous switches with flow table pipeline designs (the authors emphasize that proactive programming and reactive programming do not exclude each other. In practice, a system may adopt both approaches. This paper focuses on reactive programming).

Experiments show that rule sets generated by Maple++ are much more efficient than those generated by other SDN controllers, in the sense that the rule sets are more compact and consume less flow table space. For example, rule sets generated by Maple++ are 29–116x smaller than those generated by POX, Floodlight, and OpenDaylight, the three most widely used open-source SDN controllers. Besides, Maple++ is 5 times faster than other compilers to recover from link failures. We have implemented an open source version of Maple++ in Python. We hope this implementation could be useful for both the academia and industry.

The rest of the paper is organized as follows: Section 2 gives a motivating example of high-level SDN programming. Section 3 presents the design details of Maple++. We present evaluations in Section 4, discuss related work in Section 5, and conclude in Section 6.

2. A Motivating Example

To motivate challenges reactive programming is facing to manage heterogeneous switches with flow table pipeline designs, we use the following high-level SDN program.

Suppose a programmer is managing a local network by using a policy which consists of three parts: (1) the programmer wants to define a white list of hosts, named *recognizedHost*. Each time when a packet comes in, if either the source MAC address or destination MAC address does not exist in the white list, the packet should be dropped; (2) the programmer wants to derive the access switch of the host. If the access switch is in "programmer-defined" list *protectedAP*, a secure path should be used to forward the packet; (3) otherwise, the default shortest path should be used.

This policy is conceptually simple and straightforward. The reactive programming model allows the programmer to use familiar programming languages (such as Java and Python) and data structures (such as hash map and list) to implement the policy. Figure 1 presents the code that a programmer needs to program if he/she personally likes Python.

Specifically, the programmer defines a dictionary, *recognizedHost*, to save known MAC addresses and related access switches (Line 1 in Figure 1). And then, the programmer defines a list, *protectedAP*, to record the secure access switches. Each time when a switch receives a packet but does not know how to forward it, the switch sends a PACKET_IN message containing the header of the packet to the controller. The controller then invokes the *onPacket* function, which retrieves the source MAC and destination MAC addresses of the packet. If either the source MAC or the destination MAC address does not exist in *recognizedHost*, which is set up and maintained by another independent L2-learning program, the controller instructs the switch to drop the packet (Lines 6-7). Otherwise, the controller instructs the switch to route the packet either along a secure path or the shortest path, by checking if one of the

```
(1)  Map recognizedHost {key: macAddress, value: switchID}
(2)  List protectedAP [value: switchID]
(3)  def onPacket (p):
(4)    sw = recognizedHost [p . srcMac]
(5)    dw = recognizedHost [p . dstMac]
(6)    if sw == NULL || dw == NULL:
(7)      return DROP
(8)    elif sw in protectedAP or dw in protectedAP:
(9)      return securePath (sw, dw)
(10)   else:
(11)     return shortestPath (sw, dw)
```

Figure 1: Example of reactive high-level SDN programming (*secure-or-shortest-forwarding*).

access switches is in *protectedAP*. The controller finally inserts flow rules that match on the source MAC and destination MAC addresses. Despite its simplicity, reactive programming has the following shortcomings:

Flow Table Explosion. Existing implementations of reactive programming could not work efficiently. One problem is that *onPacket* is generating flow rules for single flow table, and as a result, the rule set generated may explode. For example, the number of rules generated by program *secure-or-shortest-forwarding* in Figure 1 is exponential to the number of MAC addresses (hosts) in network. In worst case, the network consists of 2^{48} source MAC addresses and 2^{48} destination MAC addresses, resulting in a rule set of about 2^{96} entries that is far beyond the capacity of forwarding tables in modern hardware switches (detailed in Section 3.4).

Network State Inconsistency. Another problem is that *onPacket* cannot handle network state dependencies correctly. Suppose the access switch of host *A* was not in *protectedAP*, and as a result, packets that were sent to or received from host *A* were forwarded to the shortest path. Then, one manager manually adds the switch to *protectedAP* for some security reasons. Merely forcing subsequent flows to route through secure paths is far from enough because rules that were inserted into network switches also need to be removed to avoid packets walking through unsecure links. We refer to this situation as *network state inconsistency* issues.

As far as we know, no existing work focuses on addressing these two fundamental issues in reactive programming. Maple++ tries to implement and deploy reactive programming in real networks, by solving the issues.

3. System Design

The objective of Maple++ is to implement reactive programming for heterogeneous switches with different flow table pipeline designs. Specifically, Maple++ addresses challenges including rule set explosion and network failure recovery.

To that end, Maple++ introduces a sophisticated SDN controller framework shown in Figure 2. Maple++ adopts the OpenFlow protocol to manage network switches and provides an *algorithmic policy* programming API [8]. *Algorithmic policy* API allows programmers to use familiar programming languages (such as Java and Python) to design programs and manage network. The core of Maple++ consists of four modules, including *Runtime*, *Global Optimization*, *Local Optimization*, and *Environment Information Collection*. When PACKET_IN messages arrive at the controller, module *Runtime* retrieves packet fields, runs programmer-defined programs, and logs network state dependencies. Module *Global Optimization* is to calculate routing decisions for the whole network and to perform global optimizations (such as to choose the shortest routing path). Module *Local Optimization* is to calculate routing decisions for specific network switch and to perform optimizations such as utilizing wildcards in flow rules for rule set compression. The whole system consists of a single instance of *Global Optimization*. In contrast, the system may have multiple instances of *Local Optimization*, each of which is assigned to a switch in network. Module *Environment Information Collection* is to collect network information such as network topology and the status of network devices. Another important role of *Environment Information Collection* module is to notify other modules by invoking callback functions, when the environment information changes. The remaining of this chapter introduces Maple++'s key functions one-by-one, from "programmer-defined" high-level programs to rule sets generated for switches.

3.1. Algorithmic Policy and Northbound API of Maple++. The northbound application programming interface (API) of Maple++ is based on the *algorithmic policy* presented in Maple [8], which allows an SDN programmer to specify how an incoming packet is processed and how the packet should be forwarded, by providing a general function. For example, *onPacket* in Figure 1 is such a function and is invoked for each OpenFlow PACKET_IN message. Function *onPacket* takes one parameter: the header of the packet. Within the body of *onPacket*, programmers can define local variables and calculate forwarding decisions by using familiar algorithms and data structures. Routing decisions may depend on global variables (for example, *recognizedHost* in Figure 1) and network environment context (for example, network topology for calculating shortest path). These data are stored in data stores in module *Environment Information Collection*, which is in charge of keeping the data in data stores up-to-date and notifying other modules if they have registered to some portions of the data. In Maple++, each global data structure is a wrapper of the original data structure in addition to a callback function. The callback function is used to register current forwarding decision to the data store. The return value of function *onPacket* is a forwarding path, which specifies how the packet should be forwarded.

3.2. High-Level Program to Global Forwarding Tree. Each time when a packet arrives at the controller, Maple++ runtime invokes "programmer-defined" programs and generates a forwarding decision (such as DROP, BROADCAST, or a forwarding path). At the same time, Maple++

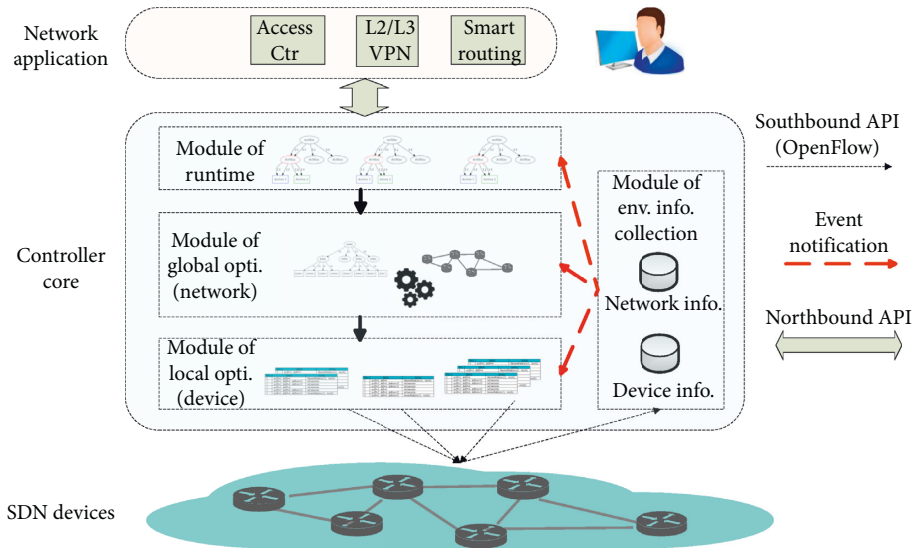

Figure 2: Maple++ system components.

runtime records the essence of the decision dependencies: packet fields accessed (source and destination MAC addresses and Ethernet type in the example program in Figure 1), global variables, and environment context accessed (network topology to make forwarding decisions). We refer to one such record as a *global forwarding decision*.

The record of forwarding decision is essence because Maple++ runtime can utilize it to generate switch forwarding rules which prevents subsequent packets of the same flow from unnecessarily being sent to the controller. This can substantially save bandwidth between the controller and switches. For example, assume one execution of *onPacket* is as follows: (1) the program reads the source MAC address of the packet, (2) the result is 00:00:00:00:00:01 (henceforth :01 for concreteness), (3) the program then reads the destination MAC address of the packet, (4) the result is 00:00:00:00:00:11 (henceforth :11 for concreteness), and (5) the program calculates and decides to drop the packet because the access switch of this host is not in the white list *recognizedHost*. Maple++ runtime then inserts a flow rule with the matching fields being source and destination MAC addresses (srcMAC == :01 && dstMAC == :11) and the action being DROP. Then, we can infer that if the program is again given an arbitrary packet with source and destination MAC addresses being :01 and :11, respectively, the program will similarly choose to drop the packet. And as a result, we can reuse the forwarding decision of the first packet, by inserting a flow rule in switches.

Forwarding decisions are organized as a tree in Maple++ runtime. Suppose the *onPacket* program runs for a while, and packets of five different flows have arrived. In one of these decisions, the MAC address of the arriving packet is :01, and hence the program returns the forwarding decision *DROP*. In the second trace, the MAC address of the arriving packet is :02, which is in the *protectedAP*, and hence the program returns a secure path between the source and destination switches as forwarding decision. Similarly, assume the subsequent three packets have the same source MAC address, :03 but have different destination MAC addresses (:11, :12, and :13, respectively), Maple++ runtime will generate three different traces, one for each of the packets. Maple++ maintains these decisions by organizing them as a tree shown in Figure 3. We refer to the tree as *forwarding tree*. In this figure, an ellipse represents a matching field and a rectangle an action. The label of an arrow represents the value of the matching field. It is worth noting that an "if" statement in high-level program generates an ellipse node with two branches, one positive and another negative.

3.3. Global Forwarding Tree to Switch's Forwarding Tree. The global forwarding tree is maintained by the *Global Optimization* module and is used to calculate forwarding decisions for the whole network. And as a result, its forwarding decision (for example, shortest path) is a list of (switch ID, port number) pairs. To generate flow tables for a specific switch, Maple++ needs to know the specific forwarding decisions for each switch (for example, output port numbers).

Fortunately, the forwarding decisions for a specific switch can be organized as a tree, which is part of the global forwarding tree, with the leaves being forwarding decisions for the switch. We refer to the forwarding tree for a specific switch as *local forwarding tree*. For example, global forwarding tree shown in Figure 3 will be translated to a local forwarding tree for switch *A* (Figure 4), if *port 1* of switch *A* is connected to the secure path, *port 2* of switch *A* is connected to the shortest path, and hosts with MAC addresses :01 and :02 attach to other switches. For each switch, there is a corresponding local forwarding tree in Maple++.

Local forwarding tree is generated and maintained by module *Local Optimization*. A straightforward strategy to generate a local forwarding tree for a specific switch is as follows: (1) copying the global forwarding tree, (2) replacing the actions by the switch-specific forwarding decisions, (3) traversing the local forwarding tree and deleting branches

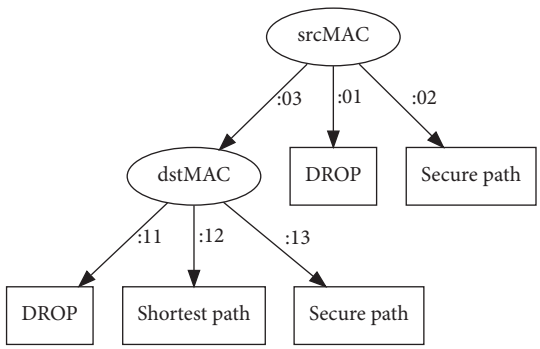

Figure 3: Maple++'s global forwarding tree consisting of five forwarding decisions.

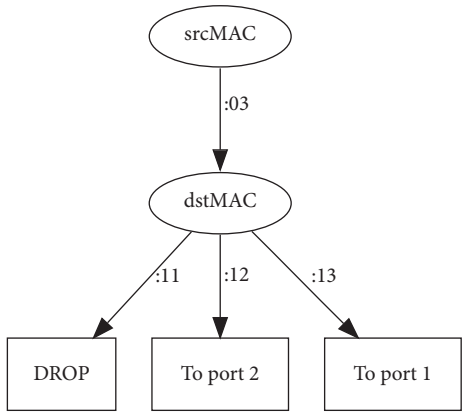

Figure 4: Maple++'s local forwarding tree for a specific switch.

that do not involve the switch, (4) and then monitoring global forwarding tree and updating the local forwarding tree accordingly.

It is worth noting that the architecture of local forwarding tree is not necessarily fixed. In other words, the tree in Figure 4 can choose srcMAC, dstMAC, or any other field of the packet as the root of the tree, only if Maple++ can generate the same forwarding rule set for the switch. In practice, the architecture of a local forwarding tree depends on the flow table pipeline design of the switch.

3.4. Compressing Forwarding Tree. One drawback of *reactive programming* is that the rule set generated may suffer explosion issue, which can be demonstrated by the size of forwarding trees. Figure 5 shows the redundancies in a forwarding tree and the potential rule set explosion. In worst case, node *srcMAC* has 2^{48} children (2^{48} MAC addresses), each of which has $2^{48} - 1$ children (leaves), resulting in a tree with a total number of $2^{96} + 1$ nodes.

To address the redundancy issue, Maple++ adopts a novel compression algorithm to dramatically reduce the size of forwarding tree. Specifically, as shown in Figure 5, many siblings of nodes srcMAC and dstMAC are redundant in the sense that their parents can refer to the same child and cut the others. For example, routing decision for MAC address pair (:01, :11) and routing decision for MAC address pair (:01, :13) are the same, and hence these two branches can be merged (Figure 6(a)). Similarly, routing decision for MAC address pair (:01, *) and routing decision for MAC address pair (:03, *) can be merged because they have the same children set (Figure 6(b)).

The basic idea to compress a forwarding tree is to perform a depth-first tree traversal. The algorithm starts from the root of the tree and then recursively checks each child. Whenever the algorithm reaches a node, it first checks if any two children of this node are the same and merges redundant children by keeping one child and removing others.

Pseudocode of the compression algorithm is shown in Algorithm 1. The algorithm starts by setting the value of field *compressed* of every node in the tree. Then the algorithm starts from the root and invokes function *Compress()* (Line 4), which performs a depth-first tree traversal. Each time when a leaf is reached, the algorithm sets the *compressed* field and returns (Lines 7–9). Otherwise, *Compress()* iterates each child of the current node (Line 11). If any child, denoted as *child_t1* in the algorithm, has not been compressed (Line 12), *Compress()* first compresses this child (Line 13). Otherwise, the algorithm checks if there are any other children which (1) have been compressed and (2) has the same children set as child *child_t1* does (Line 16). If the algorithm finds any such a child *child_t2*, *Compress()* performs compression by (1) instructing *child_t2* to point to *child_t1* and then (2) deleting the node that was pointed to by *child_t2* (Lines 17-18). Finally, *Compress()* sets the *compressed* field of the current node and then returns. Algorithm *Compress()* basically performs a depth-first tree traversal and compresses the size of tree by turning a tree into a DAG. Another benefit of performing *Compress()* on forwarding tree is that it is much more easier to map the resulting DAG to a flow table pipeline (detailed in subsequent subsections).

3.5. Generating Flow Tables from Compressed Forwarding Tree. Given a compressed forwarding tree, now we can generate flow tables for switches with flow table pipeline design. This consists of two steps: *setting up an appropriate pipeline* and *generating flow rules*.

The basic idea of setting up flow table pipeline is that the compressed forwarding tree can be divided into multiple branches, each of which can be mapped to a dedicated flow table to avoid rule set explosion in a single flow table. If a branch is too large to fit for a single flow table, then we can divide it and map its branches to multiple flow tables. In an ideal case, given the compressed forwarding tree, each node of the tree can be mapped to a dedicated flow table. Edges between different nodes can be represented by the *JUMP* instructions between different flow tables. Besides, we use *metadata* (a 64-bit long variable in OpenFlow 1.3) to pass information from one flow table to its subsequent tables.

For example, Figure 7 shows how we can map the compressed forwarding tree shown in Figure 6 to multiple flow tables. Specifically, the root of the tree (node srcMAC) can be mapped to a dedicated flow table, *Table-0*. Since the root has two branches (node pointed to by edges labeled as

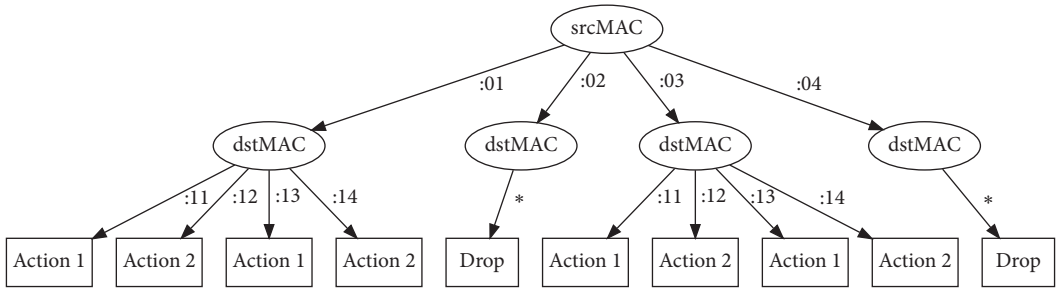

FIGURE 5: Redundancies in forwarding tree and the potential rule set explosion. In worst case, the tree consists of 2^{48} dstMAC nodes, each of which has $2^{48} - 1$ leaves.

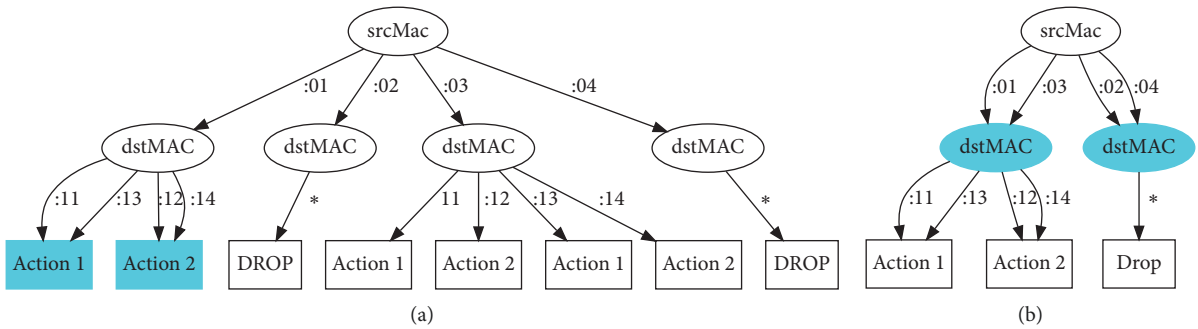

FIGURE 6: Compressing forwarding tree. (a) Compression of the leftmost subtree of forwarding tree. (b) Recursive compression of the the forwarding tree. Compression happens at nodes in blue color. The compressed forwarding tree is a DAG.

```
    Parameters: ft: Forwarding Tree
(1) Procedure CompressFT(ft)
(2)   for node in ft:
(3)     node.compressed = false; /* Initialization. */
(4)   Compress(ft.root);
(5)   return;
(6) def Compress(node): /* A depth-first tree traversal. */
(7)   if type(node) == Leaf: /* Check if node is a leaf. */
(8)     node.compressed = true;
(9)   return
(10)  else:
(11)    for child_t1 in node.children: /* Find a child. */
(12)      if child_t1.compressed == false:
(13)        Compress(child_t1); /* Recursively compress the child. */
(14)        return;
(15)      for child_t2 in (node.children-child_t1): /* Find a child that is the same as child_t1. */
(16)        if child_t2.compressed and child_t1 == child_t2:
(17)          child_t2 points to the same node of child_t1; /* Remove redundancies. */
(18)          node.children.delete(child_t2);
(19)    node.compressed = true;
(20)    return
```

ALGORITHM 1: Algorithm to recursively compress forwarding tree.

:01 and :03 and node pointed to by edges labeled as :02 and :04), we can map nodes of dstMAC to *Table-1* and *Table-2*, respectively. Similarly, children of the left dstMAC node can be mapped to a dedicated table, *Table-ACTION*. It is worth noting that, to save the number of tables; Maple++ may map a few nodes to one flow table. For example, in Figure 7, all of the leaves are mapped to a dedicated flow table, *Table-ACTION*.

Map compressed forwarding tree (DAG) to flow table pipeline is shown in Algorithm 2. It starts from the root of

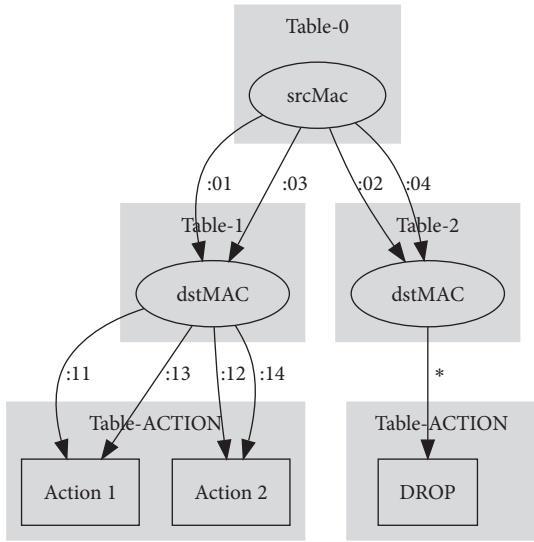

Figure 7: Mapping compressed forwarding tree (DAG) to flow table pipeline.

```
(1) // global variable
(2) tableID: global variable to record the maximum tableID in pipeline.
(3) Procedure GenerateFT(ft)
(4)     tableID = 0; /* Initialization. */
(5)     GenerateSingleFT(ft.root, ∅); /* Allocate a flow table, and generate flow rules for node. */
(6) def GenerateSingleFT(node, metadata):
(7)     if type(node) == Leaf: /* Check if node is a leaf. */
(8)         match = metadata;
(9)         priority = 0; /* Emit a rule for this leaf. The rule matches on the register values in metadata and is inserted in a specific flow
            table, table-ACTION. */
(10)        emitRule (table-ACTION, match, priority, node.action);
(11)        return table-ACTION;
(12)    else /* childrenGroup is the set of edges pointing to the same child node in compressed forwarding tree, and childrenGroup.
        groupID is a unique integer number. */
(13)        for childrenGroup in getChildrenGroup(node):
(14)            tableID_t = tableID;
(15)            tableID = tableID + 1; /* Save the matching condition on metadata. */
(16)            metadata += "reg_tableID = childrenGroup.groupID"; /* Generate flow table for this child node. childTableID is the child
                node's flow tableID. */
(17)            childTableID = GenerateSingleFT(childrenGroup.node, metadata);
(18)            for edge in childrenGroup:
(19)                match = (node.field: edge.value);
(20)                priority = 0;
(21)                action += goto childTableID; /* For each edge pointing to the child node, add a flow rule jumping to child node's flow
                    table. */
(22)                emitRule(tableID_t, match, priority, action);
(23)        return tableID_t;
```

Algorithm 2: Algorithm to map compressed forwarding tree (DAG) to flow table pipeline.

the tree (Line 5) and then traversals the forwarding tree and generates flow tables for each node by invoking function *GenerateSingleFT()*. For each nonleaf node (Lines 13–23), *GenerateSingleFT()* iterates each *childrenGroup* (Line 13), which is the set of edges pointing to the same child node in the compressed forwarding tree. For example, for the forwarding tree in Figure 7, edges ":01" and ":03" belong to the same *childrenGroup*. In addition to edge set, *childrenGroup* has two other fields: *node* and *groupID*. For each *childrenGroup*, Maple++ allocates a dedicated flow table for it and generates flow tables for this node by invoking *GenerateSingleFT()* (Line 17). And then, for each edge in the *childrenGroup*, Maple++ inserts a flow rule in the current flow table. The flow rule matches on the field of current node

and jumps to the flow table of the child node (Line 22). For each leaf (Line 7), a flow rule is inserted into flow table *table-ACTION* (Line 10). The matching field is the accumulated register values stored in *metadata*.

Generated flow table pipeline and rule sets are shown in Table 1. For concreteness, we omit the field *ingress port* in the matching field. Each table corresponds to a node in the compressed forwarding tree and matches on a single field of the packet. Each entry of the table corresponds to an egress edge from the node. Suppose a packet with source and destination MAC address pair (00:00:00:00:00:01, 00:00:00:00:00:11) arrives at the switch, the algorithm first walks through *Table-0*. Then, Algorithm 1 sets the value of *reg0* to 1 in *metadata* which will be passed to subsequent flow tables, and (2) jumps to flow table *Table-1*. Flow table *Table-1* (1) matches on the destination MAC address of the packet, (2) sets the value of *reg1* in metadata to 1, and then (3) jumps to flow table *Table-ACTION*. *Table-ACTION* matches on the *metadata* of the packet.

3.6. Handle Network Events. In SDN programming, it is still quite challenging to correctly and efficiently handle state dependencies. Take code in Figure 1 as an example. Suppose the access switch of host *A* was not in *protectedAP*, and hence packets that were sent to or received from host *A* were forwarded to the shortest path. Then, one manager manually adds the switch to *protectedAP* for some security reasons. Merely forcing subsequent flows to route through secure paths is far from enough because rules that were inserted into the switch before the manager adding the switch to *protectedAP* also need to be removed.

Typically, there are two approaches in handling state dependencies. (1) A naive approach is to simply flush flow tables by removing all of the rules installed whenever network state changes. This approach is intuitive and easy to implement. However, it is inefficient because it probably causes false-positive issues and leads to unnecessary re-executions to generate flow rules. (2) Another approach is to provide subscription APIs to programmers to allow them to enable state dependent programming. For example, whenever a program accesses environment data (for example, network topology or global variables), it needs to register to the data store. Then, whenever the data changes, system runtime will rerun the program and update retired flow rules automatically. Since the program can register to a very specific portion of the data accessed, this approach can accurately remove affected flow rules without touching others. This approach is efficient. However, it is hard to use because it is the programmers' responsibility to handle the complexity of identifying dependent data and subscribing to data store.

Based on the two existing approaches, Maple++ makes a trade-off between efficiency and easy-to-use features by utilizing a programmer-oblivious subscription/notification strategy. Specifically, Maple++ runtime adopts a normal subscription/notification strategy. Based on that, Maple++ provides (1) wrapper data structures which include original data structures in addition to implicit subscription functions and (2) wrapper function calls which include original function calls and implicit subscription functions. As a result, in Maple++, programmers use wrapper functions and data structures to program without subscribing to data store. It is the Maple++ runtime's responsibility to automatically register generated flow rules to the data accessed. At the same time, module *Environment Information Collection* is in charge of receiving network events, classifying them, and notifying different modules to rerun and to update retired flow rules.

4. Experiments

In this section, we demonstrate that Maple++ improves end-to-end performance and programming experience over existing SDN controllers by (1) generating compact forwarding rules for the state-of-the-art switches with flow table pipeline designs and (2) updating retired forwarding rules automatically. Maple++ is currently implemented as a component of Ryu [12]. We chose Ryu because it is component-based and open source. It is worth noting that Maple++ can also be implemented on top of other controllers. Our implementation of Maple++ consists of 1100 lines of Python code. For components such as OpenFlow 1.3 implementation, Maple++ reuses the implementation in Ryu.

4.1. Experiment Setup. We run controllers on a Dell R730 server. The server is equipped with two Intel Xeon E5-2609 processors, each of which consists of 8 CPU cores running at maximum speed of 1.7 GHz. Each CPU core has one 32 kB L1 data cache, one 32 kB L1 instruction cache, and a 256 kB L2 cache. All of the 8 cores on the same die share one L3 cache of 20 MB. The two processors are connected through two QPI links of 6.4 GT/s. The server uses 128 GB DDR4 memory. The server runs an Ubuntu 16.04 system with 64-bit Linux kernel version 4.4.0.

For comparison experiments (Sections 4.2 and 4.3), we evaluate all controllers by using Open vSwitch (OVS) with OpenFlow version 1.3.4. For experiments to measure Maple ++'s performance (Section 4.4), we use two HP ProCurve 5412zl switches.

4.2. Effect of Utilizing Flow Table Pipelines. In this subsection, we compare the rule set generated by Maple++ against those generated by other controllers to highlight the effect of utilizing state-of-the-art flow table pipeline designs in switches. We chose four widely used controllers, including OpenDaylight (ODL), Floodlight, Maple, and POX. ODL and Floodlight are industry-developed open-source controllers that form the basis of commercial systems, while Maple and POX are academic systems.

We run SDN program *secure-or-shortest-forwarding* (shown in Figure 1) on every controller. We choose this program because it is straightforward and can be implemented on every controller with minor modifications. In experiments, we perform an all-to-all ping among the hosts.

Table 1: Flow table corresponding to forwarding tree shown in Figure 7.

	Pri	Match	Action
Table-0	0	srcMac = :01	reg0 = 1; goto Table-1;
	0	srcMac = :03	reg0 = 1; goto Table-1;
	0	srcMac = :02	reg0 = 2; goto Table-2;
	0	srcMac = :04	reg0 = 2; goto Table-2;
	0	Otherwise	Punt
Table-1	0	dstMac = :11	reg1 = 1; goto Table-ACTION;
	0	dstMac = :13	reg1 = 1; goto Table-ACTION;
	0	dstMac = :12	reg1 = 2; goto Table-ACTION;
	0	dstMac = :14	reg1 = 2; goto Table-ACTION;
	0	Otherwise	Punt
Table-2	0	Otherwise	reg2 = 1; goto Table-ACTION
Table-ACTION	0	reg0 == 1 && reg1 == 1;	Action1
	0	reg0 == 1 && reg1 == 2;	Action2
	0	reg0 == 2 && reg2 == 1;	Drop
	0	Otherwise	Punt

We then record the round-trip time (RTT) of each ping and then count OpenFlow rules installed in the switch.

Table 2 lists the number of hosts simulated (column *# Hosts*), number of flow tables used (column *# Tables Used*), number of rules generated (column *# Rules*), and median ping RTT (column *Med. RTT(ms)*) for each controller. The number of hosts is fixed to 80, which is adequate to simulate the use case of a small enterprise network. We observe that when 80 hosts are used, Maple++ generates only 162 rules, which is 29–116 times smaller than the rules generated by other controllers.

The rule compression in Maple++ is due to the utilization of hardware flow table pipeline. Column *# Tables Used* shows that Maple++ utilizes up to 12 hardware flow tables to build a sophisticated pipeline, which fundamentally prevents rule set explosion. Theoretically, Maple++ generates approximately $2 * H$ rules. In contrast, each other controller generates approximately H^2 rules.

Column *Med. RTT* presents the round-trip time of each controller. It is worth noting that the median RTT of Maple++ does not improve in this experiment because only 80 hosts are simulated, and hence the generated rule sets can reside in hardware flow tables even if a single flow table is used. However, as the number of hosts increases, rule sets generated by other controllers may explode and hence must be evicted from the hardware flow tables frequently, resulting in an extremely large RTT. We demonstrate this in the next experiment.

To highlight the benefit of flow table pipelining in avoiding flow table explosion, we compare Maple++ with other controllers by varying the number of hosts in the network. Since Maple, ODL, and POX all have similar trends as Floodlight, we only show the results of Floodlight and Maple++ in Figure 8 for concreteness. Notice that the scale of the horizontal axis of Figure 8 is logarithmic. Figure 8 shows that, as the number of hosts in the system increases linearly, the number of flow rules generated by Maple++ also increases linearly from about 40 to 462 as the number of hosts increases from 20 to 320. In contrast, the number of rules generated by Floodlight reaches 100,000, which is too large to be deployed in a real switch. It is worth noting that, in this experiment, we use Open vSwitch which is software-based and hence its capacity is "unlimited." For production hardware, however, the size of flow tables is limited. For example, the HP 5612zl switch can only support 1,500 hardware rules and 64,000 rules in total. That means if a programmer wants to achieve a good performance, the number of OpenFlow rules generated by the controller must be less than 1.5 K. Otherwise, the forwarding performance of the switch deteriorates sharply.

4.3. Fast Repair of System State Changes. We now evaluate the effectiveness of Maple++ in recovering from system state changes. Specifically, this experiment focuses on network failure, which is unexpected, critical, and hard to handle. We evaluate controllers using three topologies: "Linear," "Square," and "FatTree" [16]. The Linear topology consists of 4 switches. The Square topology is a small cyclic topology with 4 switches. The FatTree topology consists of 20 switches and two hosts per edge switch, with $k = 4$.

In the experiment, we remove one link from the network to simulate a link failure. We then measure the time to complete pings between all hosts. For Maple++, once the *Environment Information Collection* model receives the link failure message, it immediately notifies Maple++ runtime system which in turn reruns "programmer-defined" functions registered to data store. In contrast, for other controllers, programmers need to implement a function which is responsible for cleaning up retired rules and installing new ones. Besides, for ODL and POX, all of the forwarding rules must be manually removed because there is no way to identify the affected flow rules. Since Maple has the similar trend as Floodlight, we only show the results of Floodlight. Similarly, since ODL has the similar trend as POX, we only show the results of POX.

Figure 9 shows that Maple++ provides substantial improvement in the recovery from link failures in all topologies. Specifically, the mean time to complete an all-to-all ping after a link failure in Maple++ is 0.72 seconds, 1.55

Table 2: End-to-end performance comparison and number of rules generated for SDN program secure-or-shortest-forwarding.

Controller	# hosts	# tables used	# rules	Med. RTT (ms)
Maple [8]	80	1	5120	3.1
POX [13]	80	1	18827	8.0
Floodlight [14]	80	1	5332	3.0
OpenDaylight [15]	80	1	4692	1.7
Maple++	80	12	162	2.2

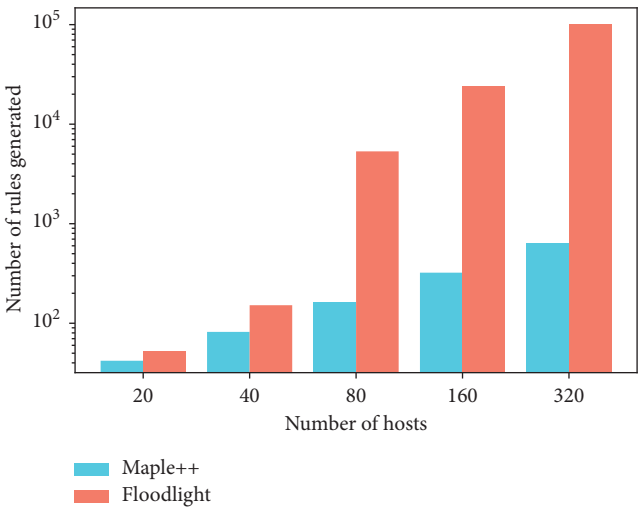

Figure 8: Total number of OpenFlow rules generated (horizontal axis is logarithmic.).

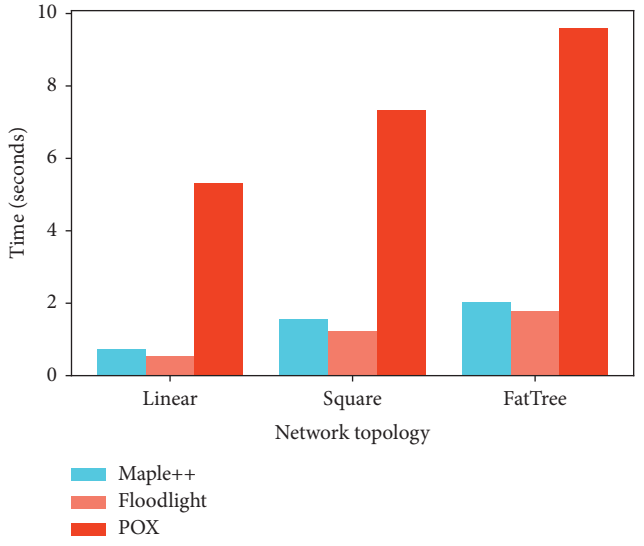

Figure 9: Time to repair all-to-all connectively after a link failure.

seconds, and 2.03 seconds, for the three topologies. The result is competitive to Floodlight and is much better than controllers based on POX and ODL. POX and ODL perform poor because they flush the flow table upon receiving the link failure message, which causes false-positive issues and forces the controller to recompute for every flow rules. It is worth noting that the reason Maple++ does not perform better than Floodlight is that Maple++ has to update multiple flow tables due to its flow table pipeline design. This bottleneck can be removed by allowing Maple++ to recompute and update multiple flow tables in parallel. We left this optimization as a future work.

4.4. Maple++ in Real Network. In this subsection, we deploy Maple++ in a real network, show the flow tables generated, and discuss lessons we have learned. A simplified topology of this network is shown in Figure 10. The local network consists of two normal switches (*Switch 1* and *Switch 3*) and two HP 5012zl OpenFlow switches (*Switch 2* and *Switch 4*). All of the four switches are connected through 1 Gbps links. The link between *Switch 2* and *Switch 4* and the link between *Switch 1* and *Switch 4* are secure paths. However other links are not secure because it may exist as malicious switches in between these links. We assume *Switch 1* is in the *secureAP* list, while other switches are not. Each switch is connected by two hosts. For example, *Switch 1* connects to two hosts, *Host 1* and *Host 2*. The IP addresses of these two hosts are "10.0.0.1" and "10.0.0.2," respectively. To simplify the specification, we assume the MAC addresses of *Host 1* and *Host 2* are "00:00:00:00:00:01" (shorted as :01) and "00:00:00:00:00:02" (shorted as :02), respectively. Besides, we assume that the MAC addresses of hosts with green check mark are in the *recognizedHost*. Packets from other hosts should be dropped.

To demonstrate how Maple++ works, we deploy the *secure-or-shortest-forwarding* program shown in Figure 1 and choose *Switch 2* as an example. For concreteness, we omit the field *ingress point* in the matching field. Initially, the flow tables of this switch are empty. Then, the first time when *Host 3* wants to connect to *Host 6*, a PACKET_IN message is sent from the switch to the controller, which in turn invokes the *secure-or-shortest-forwarding* program and then inserts a rule into the switch to instruct the switch to block subsequent packets because *Host 6* has not been recognized. Similarly, the first time when *Host 3* and *Host 5* want to connect to each other, the controller invokes the *secure-or-shortest-forwarding* program and then inserts rules to switch 2 and switch 3 to allow connections to go through the shortest path. In contrast, if *Host 3* wants to connect to *Host 1*, the routing path must be *Switch* 2 < – > *Switch* 4 < – > *Switch* 1 because *Switch 1* is in the *secureAP*, and hence all connections involving hosts behind *Switch 1* must be routed through the secure path.

The flow tables produced by Maple++ for *Switch 2* are shown in Table 3. Table 3 shows that Maple++ utilizes four flow tables in *Switch 2*. Specifically, *Table-0* is used to match on source MAC address. For each host in the network, a flow

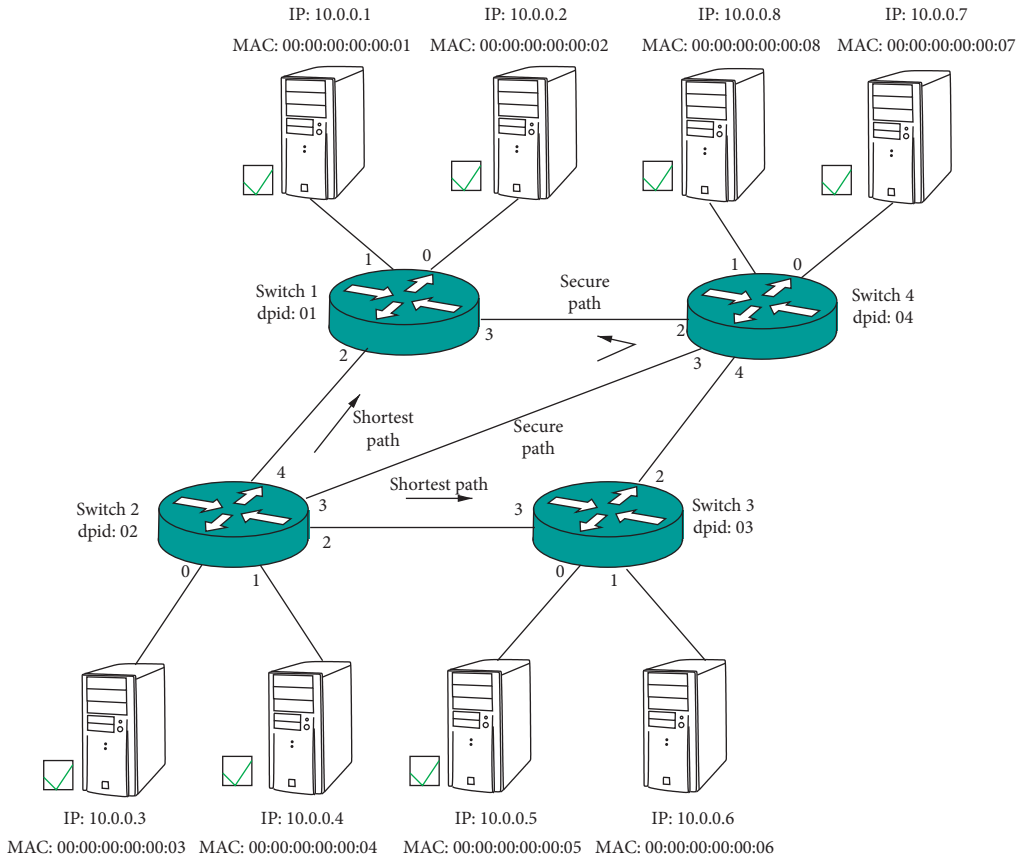

Figure 10: Topology of the testbed network. Hosts with green checkmarks are in the *recognizedHost*, and *switch 1* is in the *secureAP list*.

Table 3: Flow table pipeline and forwarding rules generated by Maple++ for Switch 2. For concreteness, MAC address 00:00:00:00:00:xx is represented as :xx in the table.

	Pri	Match	Action
Table-0	0	srcMac = (:03-:04)	reg0 = 1; goto Table-1
	0	srcMac = (:01-:02)	reg0 = 2; goto Table-2
	0	srcMac = (:05–:08)	reg0 = 2; goto Table-2
	0	Otherwise	Punt
Table-1	0	dstMac = (:01-:02)	reg1 = 1; goto Table-ACTION
	0	dstMac = :03	reg1 = 2; goto Table-ACTION
	0	dstMac = :04	reg1 = 3; goto Table-ACTION
	0	dstMac = :05	reg1 = 4; goto Table-ACTION
	0	dstMac = :06	reg1 = 5; goto Table-ACTION
	0	dstMac = (:07-:08)	reg1 = 6; goto Table-ACTION
	0	Otherwise	Punt
Table-2	0	dstMac = :03	reg2 = 1; goto Table-ACTION
	0	dstMac = :04	reg2 = 2; goto Table-ACTION
	0	Otherwise	Punt
Table-ACTION	0	reg0 == 1 && reg1 == 1	To port 3 (secure path)
	0	reg0 == 1 && reg1 == 2	To port 0
	0	reg0 == 1 && reg1 == 3	To port 1
	0	reg0 == 1 && reg1 == 4	To port 2 (shortest path)
	0	reg0 == 1 && reg1 == 5	DROP
	0	reg0 == 1 && reg1 == 6	To port 2 (shortest path)
	0	reg0 == 2 && reg2 == 1	To port 0
	0	reg0 == 2 && reg2 == 2	To port 1
	0	Otherwise	Punt

rule is added into *Table-0*. In practice, after compressing the forwarding tree, a flow rule can correspond to a group of MAC addresses. For example, the first flow rule in *Table-0* can match packets from both *Host 3* and *Host 4*. The first rule instructs the switch to set the value of *reg0* to 1 in *metadata* and then jumps to *Table-1* to process the destination MAC addresses of incoming packets. The second rule instructs the switch to set the value of *reg0* to 2 in *metadata* and then jumps to *Table-1*. Similarly, *Table-1* checks the destination MAC address of the packet, logs the routing decision on *metadata* by setting the value of field *reg1*, and then jumps to *Table-ACTION*. *Table-ACTION* checks the values in *metadata* and then takes actions.

Table 3 shows that the size of *Table-0*, *Table-1*, and *Table-2* increases linearly with the number of hosts in the network. The size of *Table-ACTION* increases linearly with the number of policies defined by programmers, which is typically a constant number. Overall, the number of rules generated by Maple++ is linear, which makes Maple++ a practical SDN controller by preventing rule set explosion.

5. Related Work

The importance of flow table pipelining has motivated researchers to provide corresponding high-level programming languages. For example, in [17], a typed programming language called Concurrent NetCore is proposed to specify flow tables. P4 [6] and POF [7] provide forwarding models with configurable flow table pipeline and programmable parsers. Jose et al. [6] study algorithms for mapping flow table designs to particular target switches [18, 19]. These languages, however, require programmers to explicitly specify low-level details of flow tables.

Another type of high-level programming is *reactive programming* [8, 20] which allows a controller to automatically identify switch configuration, generate flow tables, and populate flow rules. Maple [8] is a classic implementation of *reactive programming*. However, Maple is designed for switches with single flow table. Maple++ is an extension of Maple and supports flow table pipelining switches by addressing inherent issues in Maple.

Another trend in supporting heterogeneous switches is by providing a uniform API to controllers. For example, MACSAD [21, 22] aims at hiding data plane programming complexity by using P4 programming language while keeping the flexible data plane portability. TableVisor [23, 24], by providing a transparent proxy layer, allows pipeline processing and enables the extension of hardware flow table sizes using multiple hardware switches. These works focus on the switch side, whereas Maple++ focuses on the programming language.

6. Conclusion and Future Work

This paper explores an efficient and programmer-friendly SDN programming framework for state-of-the-art switches with flow table pipeline designs. We present novel techniques to compress the rule sets and to map them to flow table pipelines and show that the generated rule sets are highly compact on a variety of benchmarks using both simulated and real network workloads. This work can be optimized, for example, by utilizing priority numbers in generating rules.

Data Availability

Projects POX, Floodlight, and OpenDaylight, which are used to support the findings of this study, can be found at https://github.com/noxrepo/pox, http://floodlight.openflowhub.org/, and http://www.opendaylight.org, respectively. Source code of Maple and Maple++ are currently under embargo while the research findings are commercialized. Requests for the source code, 12 months after publication of this article, will be considered by the corresponding author.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

The authors are grateful to anonymous reviewers whose valuable comments helped to improve the quality of this paper. This work was supported in part by the National Natural Science Foundation of China under Grant No. 61602264, China Post-Doctoral Science Foundation under Grant No. 2017M611882, and Primary Research and Development Plan of Jiangsu Province under Grant No. BE2017743.

References

[1] B. Heller, S. Seetharaman, P. Mahadevan et al., "Elastictree: saving energy in data center networks," in *Proceedings of NSDI*, vol. 10, pp. 249–264, San Jose, CA, USA, April 2010.

[2] S. Jain, A. Kumar, S. Mandal et al., "B4: experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[3] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pp. 11–18, ACM, Barcelona, Spain, August 2009.

[4] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," *China Communications*, vol. 11, no. 12, pp. 72–82, 2011.

[5] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," Technical Report of USENIX, Berkeley, CA, USA, 2013.

[6] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *Proceedings of NSDI*, pp. 103–115, Oakland, CA, USA, May 2015.

[7] 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, ser. HotSDN '13*, pp. 127–132, ACM, Hong Kong, China, August 2013.

[8] 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, ser. SIGCOMM'13*, pp. 87–98, ACM, Hong Kong, China, August 2013.

[9] A. Voellmy, S. Chen, X. Wang, and Y. R. Yang, "Magellan: generating multi-table datapath from datapath oblivious algorithmic sdn policies," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pp. 593-594, ACM, Florianopolis, Brazil, August 2016.

[10] A. Sivaraman, A. Cheung, M. Budiu et al., "Packet transactions: high-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 15–28, ACM, Florianopolis, Brazil, August 2016.

[11] B. Limited, *Openflow Data Plane Abstraction (OF-DPA, Version 2)*, 2015, https://docs.broadcom.com/docs/12378911.

[12] Ryu SDN Controller, 2018, https://osrg.github.io/ryu/.

[13] POX SDN Controller, 2018, https://github.com/noxrepo/pox.

[14] Floodlight OpenFlow Controller, 2018, http://floodlight.openflowhub.org/.

[15] OpenDaylight, 2018, http://www.opendaylight.org.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, ser. SIGCOMM'08*, pp. 63–74, ACM, Seattle, WA, USA, August 2008.

[17] 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, ser. ICFP'14*, pp. 11–24, ACM, Gothenburg, Sweden, September 2014.

[18] P. Bosshart, G. Gibb, H.-S. Kim et al., "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, ser. SIGCOMM'13*, pp. 99–110, ACM, Hong Kong, China, August 2013.

[19] R. Ozdag, *Intel Ethernet Switch FM6000 Series—Software Defined Networking*, 2016, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[20] C. He and X. Feng, "Pomp: protocol oblivious sdn programming with automatic multi-table pipelining," in *Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 998–1006, IEEE, Honolulu, HI, USA, April 2018.

[21] P. G. Patra, C. E. Rothenberg, and G. Pongrácz, "Macsad: multi-architecture compiler system for abstract dataplanes (aka partnering p4 with odp)," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 623-624, ACM, Florianópolis, Brazil, August 2016.

[22] P. G. K. Patra, F. E. R. Cesen, J. S. Mejia et al., "Towards a sweet spot of dataplane programmability, portability and performance: on the scalability of multi-architecture P4 pipelines," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 3–14, 2018.

[23] S. Geissler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, and M. Jarschel, "Tablevisor 2.0: towards full-featured, scalable and hardware-independent multi table processing," in *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*, pp. 1–8, Bologna, Italy, July 2017.

[24] S. Geissler and T. Zinner, "Tablevisor 2.0: hardware-independent multi table processing," in *Proceedings of the KuVS-Fachgespräch Fog Computing 2018*, p. 33, Darmstadt, Germany, March 2018.