FlowConvertor: Enabling Portability of SDN Applications

Heng Pan*†‡, Gaogang Xie*, Zhenyu Li*, Peng He*, Laurent Mathy‡
*ICT, CAS, China, †University of CAS, China, ‡University of Liege, Belgium
{panheng, xie, zyli, hepeng}@ict.ac.cn, laurent.mathy@ulg.ac.be

***Abstract*—Software-Defined Networking (SDN) provides network administrators opportunities to control network devices more simply and easily than in traditional networking. However, heterogeneity in switch hardware, especially in forwarding pipeline architecture, renders the task of network application developers and network administrators tedious, by hampering portability across switch models.**

In this paper, we propose FlowConvertor, an algorithm capable of converting rules from any forwarding pipeline to any other different forwarding pipeline, as long as both pipelines offer compatible operations. More precisely, FlowConvertor is an online algorithm that operates on flow updates issued to the origin pipeline and computes the corresponding updates for the target pipeline in real time. Performance evaluation shows that the latency introduced by FlowConvertor on the path between the SDN controller and the target switch is of the order of 1ms in most cases, and is thus acceptable for practical deployment.

I. Introduction

Software-Defined Networking (SDN) separates the control plane from the forwarding plane to simplify network management and enable complex network applications. The main mechanism supporting this separation is the exposure of the basic forwarding functionality from switches to a logically central controller, through a standardized API such as OpenFlow [20]. While the interface between the controller and the switches in the network is standardized, the details of switch implementation are (rightly) left to vendors, who are free to seek competitive advantages as they see fit. This leads to *switch diversity* [19], [27], [23].

One area where switch diversity is salient is forwarding. Indeed, in order to support complex forwarding scenarios in a scalable way, modern switch forwarding engines are often built as a pipeline of flow tables [2], [22]. The number of tables in the pipeline, their sizes, what header fields each individual table matches, can all differ from one switch model to another. OpenFlow [5] deals with the diversity in forwarding engines by directly exposing the detailed structure of the forwarding engine to the network applications, via the switch hardware capabilities mechanism [6]. In other words, switch diversity is actually passed on to the application to cope with, which burdens the network programmer with low level details and explicit awareness of equipment diversity in the network.

But because of switch diversity, network application programmers must thus explicitly generate specific rules for each specific forwarding pipeline in the network. This is obviously cumbersome and error prone. Worse, it is a clear inertia for network evolvability, as the introduction of a new model of switches in the network may require modifications to the applications themselves. Even if high-level programming framework existed that could issue rules directly to pipeline forwarding engines, the deployment of a new switch model might still require applications to be recompiled, thus disrupting and interrupting the operation of the network.

Several proposals exist, such as Frenetic [14], Pyretic [21], or Maple [8] that aim to raise the level of abstraction for programming SDN applications. However, few of them supports pipelined forwarding engine architecture, and can only be used for one single table abstraction forwarding engines. As a result, network applications must explicitly output pipeline rules to the SDN controller which then actuates these on the switches. The Open Compute Project (OCP) [4], proposes to use a switch abstraction interface (SAI) to abstract the diversity away from the application developers. Yet, the process of translating the abstract forwarding rules into actual switch rules is an unresolved issue.

In this paper, we seek a different approach to support switch diversity in a software defined network: a method to automatically and transparently map any forwarding pipeline ruleset onto an appropriate ruleset for any other (different) pipeline. Such an algorithm, which we call *FlowConvertor*, would then completely isolate the pipeline forwarding output of SDN applications from the details of the actual forwarding engines in the switches. In fact, application output could even be targeted to a completely abstract forwarding pipeline that does not correspond to any of the physical forwarding engines in the network.

In FlowConvertor, each instance simply requires the origin and target forwarding pipeline specifications (e.g. as OpenFlow switch hardware capabilities) and rules for the origin pipeline as input, and it will produce rules for the target pipeline as output. In particular, the forwarding pipeline is abstracted as a directed acyclic graph (DAG). Each flow entry is represented by a vertex. A directed edge exists from a "source" vertex to other "destination" vertex if the later's match field is updated by the former's instructions. Built on top of the graph, FlowConvertor consists of five computation phases, including origin graph maintenance, path generation, path filtering, target priority assignment and rule mapping. The efficiency and overhead of FlowConvertor are evaluated using two types of OpenFlow switches. In summary, the contribution of this work is twofold:

978-1-5090-5336-0/17/$31.00 ©2017 IEEE

- We design and implemented FlowConvertor, an algorithm that can be run anywhere in-between (and including) the application and the switch, providing flexible and transparent portability of SDN applications to any pipeline forwarding engine, without disruption to the network.
- We evaluate FlowConvertor using both commodity switch and software-based switch. The results show that the per-update processing time is as low as 25 μs, and the latency introduced on the path between the SDN controller and the target switch is of the order of 1ms in most cases. The results consistently suggest that and is thus FlowConvertor is able to achieve rule conversion on-line and in real time, and is acceptable for practical deployment.

The rest of the paper is structured as follows. Section II describes the background and the motivating example. The design of FlowConvertor is presented in Section III, followed by the implementation in Section IV. We present evaluation results in Section V. Section VI surveyed related work, and we conclude our work in Section VII.

II. Background and Motivation

A. Background: Pipelined Forwarding

In the OpenFlow [5][1] switch, packets are classified into flows through a pipeline of *flow tables*. Each flow table contains prioritized *flow entries*. A flow entry in a table consists of one or more *match fields* and associated *instructions*[2]. A match field can represent a packet header field, ingress port, or some internal *metadata*. Metadata can be thought of as a value summarizing the matching state for a packet up-to the current pipeline stage.

Because of the prioritization of flow entries, a packet will match one flow entry at each stage of the matching pipeline (or be dropped/sent to the controller if no such entry exists in the table). Once the matching entry is found, the associated instructions are applied to the packet. An instruction can either immediately apply *actions* to the packet or modify the *action set* associated with the packet. Additionally, there are instructions to steer pipeline processing, by selecting the next table the packet should visit, and setting the current metadata value for the packet.

An action can modify the packet by either modifying some of its header fields or push/pop headers on the packet. There are also forwarding actions, that direct packets to switch ports (including "logical" ports representing sending the packet to the SDN controller or discarding the packets). The action set associated with a packet is simply a series of actions executed when the packet exits the forwarding pipeline (that is, when the packet gets forwarded to a non-discarding port).

B. A motivating example

Let us consider a SDN application whose policy consists of two parts (see Figure 1): (1) a load-balance policy that examines source IP to decide the destination IP; (2) a firewall policy that only forwards TCP flows with destination port 80 based on destination IP address.

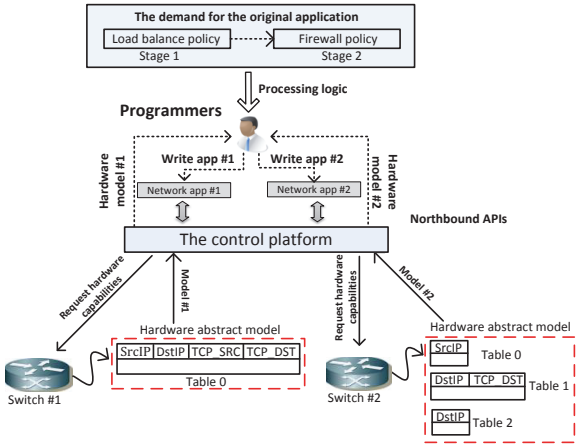

Fig. 1. A motivating example. SDN programmers have to rewrite applications according with diverse hardware models.

This application needs to be deployed to two switches. Unfortunately, the forwarding engines of these two switches are diverse, i.e. Switch `#1` consists of only one table but this table supports all necessary matches; Switch `#2` consists of three flow tables but each table only supports some limited matches. Therefore, SDN programmers have to request the hardware abstract model at first. Then, based on the original policy processing logic, they have to rewrite the application according with the specific hardware model (see Figure 1). As a result, programmers have to rewrite a SDN application many times if they want to deploy it in diverse underlying switches. This is obviously cumbersome and error-prone.

To address this challenge, it is very significant to make SDN application portability. Therefore, we propose FlowConvertor that can automatically and transparently map any forwarding pipeline ruleset onto an appropriate ruleset for any pipeline.

III. Design of FlowConvertor

A. Algorithm Principles

When a packet enters a forwarding pipeline, it will visit a series of tables, "hitting" one flow entry (i.e. matching rule) in each of the visited table. The series of flow entries hit thus forms a path through the pipeline. Conceptually, the flow entries in the forwarding pipeline form a directed acyclic graph (DAG), where each flow entry can be represented by a graph vertex. Remember that each flow entry either directs the packet for processing by another table or it must forward the packet to a port (or drop it). We can therefore consider that there is a directed edge from a vertex (representing the "source" flow entry) to any other "destination" vertex, belonging to the table identified by the `goto-table` instruction in the source flow entry, and whose associated flow entry uses the current value of the metadata as a match field (see Figure 2).

[1]Note that our work is general and not limited to OpenFlow.

[2]A flow entry actually has other components, such as counters, but these can safely be ignored in the context of our discussion.

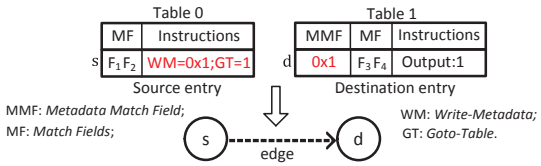

Fig. 2. The key observation in the OpenFlow pipeline processing. Each field of MF represents a packet header field or ingress port.

B. Algorithm Description

FlowConvertor keeps an internal representation of the origin and target pipelines as a graphs. While these graphs are directed graphs, the internal representation is such that, from any vertex, both sets of upstream and downstream neighbours can quickly be found. As FlowConvertor is an on-line algorithm, the internal graphs are actually built incrementally, as a result of `flow-mod` requests.

For each flow entry modification (either "add", "delete" or "update" operations), the algorithm goes through five computation phases, which we describe in further details below:

1) Update of the internal graph representation of the origin pipeline (sect. III-B1);
2) Generation of all the paths through the vertex affected by the request (sect. III-B2);
3) Filtering out of invalid paths (sect. III-B3);
4) Priority assignment (sect. III-B4);
5) Mapping of valid paths onto the target pipeline (sect. III-B5) and update of target pipeline graph representation.

1) Origin Graph Maintenance: Remember that each flow entry (a flow entry is a table entry) is represented in the graph as a vertex[3]. The graph is based solely on `goto-table` instructions (in flow entries) and `metadata` values (through `write-metadata` instructions and metadata values in match fields).

The most straightforward case to consider is that a flow entry that uses a `write-metadata` instruction to set the metadata (m) value, and directs processing (via a `goto-table` instruction) to a table that uses the metadata m as one of its match fields. In this case, an edge is established between the vertex representing this flow entry and each vertex representing the flow entries in the destination table, whose value for the metadata match field equals the value set by the `write-metadata` instruction (see Figure 3).

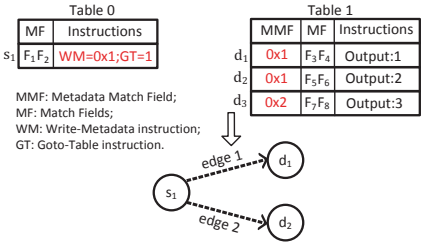

Fig. 3. A graph that is constructed by using write-metadata and metadata

[3]To simplify our discussion, we use the terms "flow entry" and "vertex" interchangeably.

There are a few additional cases to consider. If the destination table does not use the metadata as a match field, then there must be an edge between the redirecting flow entry and every flow entries in the destination table (see Figure 4).

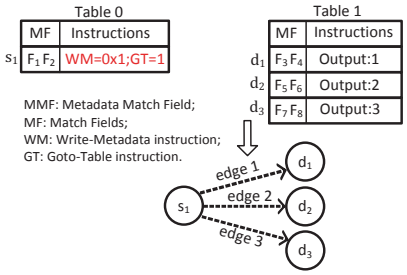

Fig. 4. One target table that does not use the metadata as a match field

Similarly, if a flow entry does not use a `write-metadata` instruction, but redirects processing toward a table that uses a metadata match field, then two cases occur. The first such case is when the flow entry itself has a metadata match field (see Figure 5(a)). In this case, the value of this metadata match field (in the source flow entry) is used to select which flow entry, in the destination table, is connected to the source. The second case is when the source flow entry does not use metadata as a match field (see Figure 5(b)). In that scenario, the source vertex's upstream neighbors must be visited recursively to discover all the metadata values that were last set on the paths to this vertex, and these values are used to select the destination vertices in the destination table.

Using these rules, the algorithm ensures that, on reception of every flow entry modification, the internal representation of the source pipeline is correct and up-to-date. This work is straightforward and localized to the neighbourhood of the modified flow entry.

2) Paths Generation: With the internal representation of the state of the origin pipeline up-to-date, FlowConvertor can now generate all the paths in the graph that pass through the modified vertex.

Realizing that the local paths through a single vertex (a local path comprises one upstream and one downstream vertex) are the "cross product" of the ingress edges to the vertex and its egress edges, all the paths through the modified vertex can easily be generated through a simple recursion starting at that vertex. The internal representation of the (directed) graph maintains, for each vertex, a set of upstream neighbors precisely to support efficient path generation. The recursion is easily arranged so that paths are generated by rank, that is in decreasing priority order (in terms of the relative priority of the corresponding rules): highest priority parent first, through all the children in decreasing priority order, before considering the next parent.

Note that in each table of the pipeline, several entries can share the same priority, but these are normally non-overlapping[4] [5], which causes no issue as their relative

[4]If some equal-priority entries overlap and share the same priority, the chosen one, for a packet "hitting" multiple rules, is explicitly undefined [5], so their relative ranking is irrelevant.

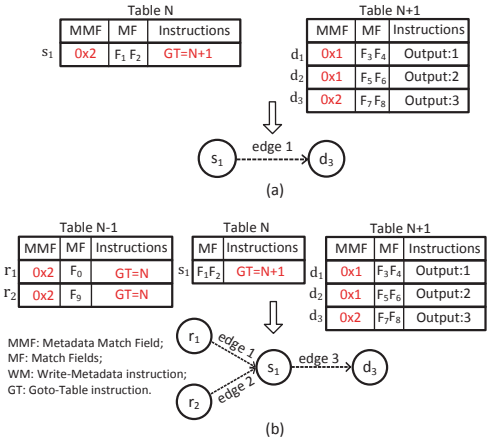

Fig. 5. An example that the source table does not use the write-metadata instruction

ranking is thus irrelevant and any such relative ranking is acceptable.

3) Path Filtering: Because the internal graph representation of the origin pipeline solely reflects the results of `goto-table` instructions and metadata values (see section III-B1), the set of paths generated in the previous phase could comprise paths that are impossible for a packet to take.

Indeed, flow entries can also contain instructions that modify packets, such as instructions that `set` header fields to explicit values, or instructions that `push`/`pop` header fields ("tags") to/from packets. When a header field is pushed onto a packet, its value must also be set through a `set` instruction. Whenever such `set` instruction is encountered, the set value will obviously restrict which flow entries, downstream from the `set` instruction, can match the packet while using the corresponding header field as a match field (see Figure 6). This path filtering phase of the algorithm examines each paths and filters out *invalid* paths that exhibit impossible matching state.

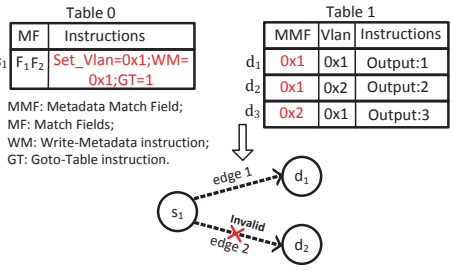

Fig. 6. Example of Set instructions that modify packets.

One might wonder why we do not also somehow remove such paths from the internal graph representation, in order to avoid generating them again. The reason is that doing so would very much complicate the update phase (see section III-B1), as a later modification (for instance a modification changing a `set` instruction) may "re-enable" an otherwise removed path.

4) Target Priority Assignment: Recall that the paths were generated in rule priority order (see III-B2), and path filtering (see section III-B3) preserves this property.

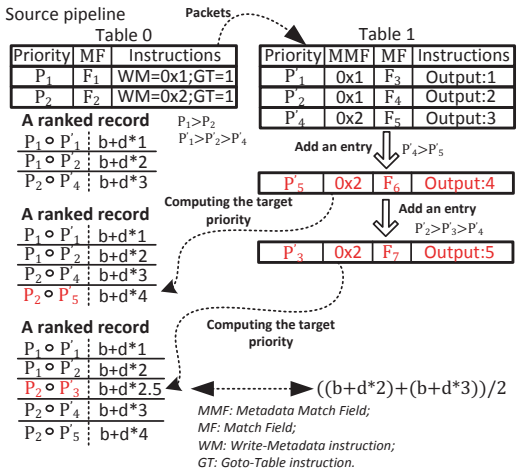

Fig. 7. Example of the target priority assignment.

To compute rule priorities for the target pipeline, we propose the following, simple method: FlowConvertor keeps a ranked record of the rules in the source pipeline. Given a path through the directed graph representing the source pipeline (thus representing a rule), the immediately higher ranked rule can be determined easily. This rule is thus inserted in the ranked record just after its immediately ranked rule (or as the top ranked rule if no such rule exist), thus implicitly increasing the rank of the rules below the insertion point.

Whenever a lowest rank rule is inserted in the record, a simple formula is used to compute the corresponding rule priority in the target pipeline: $b + d * r$, where b and d are constants, and r is the rank of the inserted rule. This target priority value is recorded, along with the rule in the ranked record.

If the inserted rule is not the lowest ranked rule in the ranked record, then the rule priority is chosen in the interval defined by the priorities of the immediately higher ranked and the immediately lower ranked rule (e.g. it is chosen as the middle of that interval). The roles of the two constants in this mechanism therefore becomes clear: b represents a "gap" to facilitate insertion at the top of the ranked record, while d is used to leave a "gap" between rules inserted in order (see Figure 7). Of course, if any of these gaps are full, then priority re-assignment must occur for some of the rules following the inserted one. This re-assignment can stop as soon as appropriate priority values are found in an existing gap.

When new table entries for a rule are issued to the target pipeline (see section III-B5), they will be issued with this computed target priority for the rule (ignoring, for the sake of simplicity, nitty-gritty details such as table priority field width).

5) Rule Mapping: The set of remaining paths represents valid classification rules, to be mapped onto the target forwarding pipeline.

Each node (vertex) in the path represents a flow entry (in the origin pipeline) with associated match field values and instructions. A first observation is that these will include `goto-table` and `write-metadata` instructions, as well

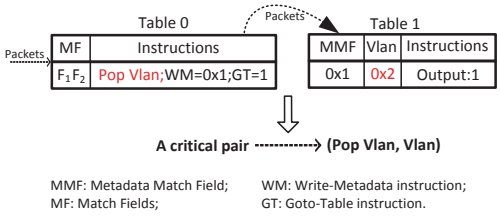

Fig. 8. Example of a critical pair

as metadata match field values. However, these are not only specific to the origin pipeline, their effects have also been "recorded" in the path itself: each path was extracted from a graph, built from these very instructions and values (see section III-B1). These `goto-table` and `write-metadata` instructions, as well as the metadata match fields, can thus be ignored and removed from the path.

In terms of the remaining match fields, the order in which they are matched by a packet is irrelevant from a classification point-of-view: as long as they are all matched by the packet, the corresponding rule will correctly classify the flows. For the remaining instructions, there is an important observation: any instruction that modifies the packet itself (`set` and `push/pop` instructions), must, in the target pipeline, have the same relative order, with regard to the corresponding match fields, as in the origin pipeline. *Critical* $< instruction, MFV^5 >$ and $< MFV, instruction >$ pairs can thus be easily identified by simple path inspection (see Figure 8).

Another important observation, is that a forwarding pipeline may contain *redundant operations* and *null operations*. A null operation is a sequence of instructions and match fields that has no side-effect outside of the pipeline. Examples of null-operations are depicted in Figure 9(a). A redundant operation stems from a pair of instructions and/or match fields that has some side-effect outside of the pipeline, but contains a match field that is not strictly necessary to ensure correct classification or an instruction that has neither any effect on the classification nor any effect outside of the pipeline. Examples of redundant-operations are depicted in Figure 9(b). For instance, the match field value in a critical $< instruction, MFV >$ pair is a redundant operation.

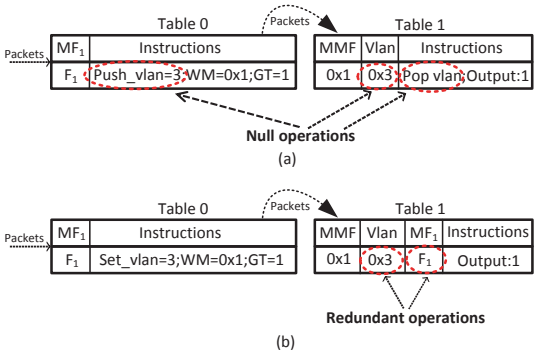

Fig. 9. Example of a null operation and a redundant operation

[5]MFV stands for Match Field Value.

Redundant and null operations may appear in a forwarding pipeline for various reasons, such as "workarounds" (for instance to trigger selection of specific flow entries in a table where a metadata match field is unavailable), eager and/or transient pipeline state computations, or application inefficiencies. The algorithm scans the pipeline path and marks all redundant and null operations. Null operations, that do not involve a match field (such as simple push/pop sequence), as well as redundant instructions (such as the first of two set operations on a same header field, without any intervening match field operation on that header field) can, and should be removed from the path.

The algorithm is now ready to map the origin pipeline path onto the target pipeline. We first consider the case of adding/updating a flow entry. To start with, a target path is built, which simply represents a series of "empty" flow entries (one per table) in the target pipeline.

The first step in the mapping process is to fill in match fields in the target path. Starting at the first vertex of this target path, for each match field entry of this flow entry, we seek the corresponding value by scanning the origin path (from left to right, in natural pipeline order) and picking the first suitable value found, while skipping over origin match fields involved in redundant and null operations. The picked match field values are removed from the origin pipeline path. This is repeated for every vertex in the target path.

At this point, all of the metadata match fields, as well as possibly other match fields, are left empty in the target path (Figure 10). If all the match fields in a target flow entry are still empty, this entry is by-passed (i.e. removed) from the target pipeline path. FlowConvertor then looks to use the (skipped) redundant and null operations left in the origin path to fill the remaining non-metadata match fields in the target path.

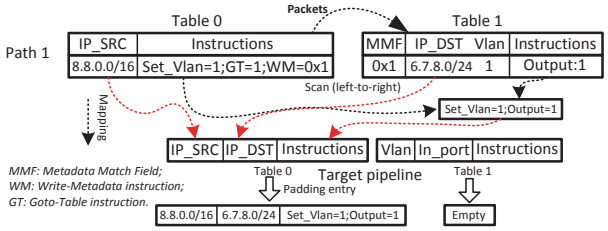

Fig. 10. Example of mapping a path into target tables

Note that if any non wild-card match field value, not involved in a redundant or null operation, is left in the origin path, then the origin pipeline performs some operations that the target pipeline cannot support, and we have a mapping failure (see Figure 11). It should be clear that such mapping failures are not "structural": they do depend on the content of the origin pipeline path, not just the respective structure of the forwarding pipelines.

Also note that, match fields involved in a null operation or redundant match fields should not be used if the corresponding table in the target pipeline has a metadata match field. This is because the null operation may have been inserted in the origin table as a workaround to a possible lack of metadata match

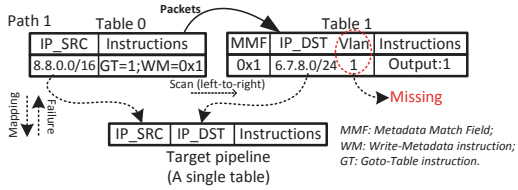

Fig. 11. Example of mapping failure because the target pipeline cannot be supported.

field, and that this null operation would thus unnecessarily duplicate the function of the metadata match field present in the target table.

Instructions that have been involved in critical pairs (see Figure 8) are then placed in the target pipeline, respecting their relative order with the corresponding match fields. For simplicity, but this is not the only possibility, an instruction that must occur before a match field is placed in the flow entry in the table just before this match field, while an instruction that must occur after a match field, is place in the same flow entry as the match field itself. If any of these "critical" instructions cannot be placed appropriately, then the mapping has failed (see Figure 12).

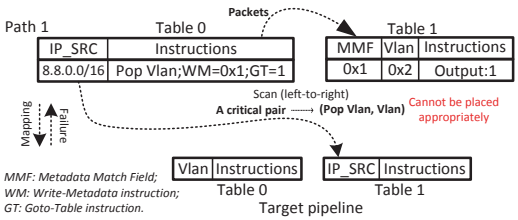

Fig. 12. Example of mapping failure because the critical pairs cannot be placed appropriately.

All `goto-table` instructions are also placed in the corresponding flow entries. At this point, any empty non-metadata match field left in the target path is filled with a wild-card value. If wild-cards are not supported for the match field, the first value found in either an equivalent match field or a set instruction up the target path is used. If such a value cannot be found, the match field is marked as "open"[6]. Every metadata match field is also marked "open". All remaining instructions in the origin path are then added to the last vertex of the target path.

For each table in the target pipeline, the algorithm keeps a representation of every flow entry it has previously issued to the switch. Using recursive backtracking, FlowConvertor will thus compute the final `flow-mod` messages (see Algorithm 1): for each vertex in the target path, starting from left-to-right, FlowConvertor checks if an equivalent flow entry has already been issued. An *equivalent* flow entry is one that, ignoring open match fields (and any instructions on these), contains the same value for the match fields, as well as the same instructions.

[6]An open match field in the first table of the target pipeline would require an enumeration of all possible values for the match field.

Algorithm 1: COMPUTEFLOWMOD(*e,tables*)

```
Input: e: the current vertex in the target path
Input: tables: the views of the target pipeline
if e == NULL then
    return;
e' ← GetEquEntry(e, tables);
if e' ≠NULL then
    if HasOpenFields(e) then
        f← FillOpenFields (e, e');
        BackTrack (e, f);
    Copy instructions from e' to e;
    ApplyInst. (e, e.next);
    e.tag ← old;
else
    if HasOpenFields(e) then
        f←CreateOpenFields(e, tables);
        BackTrack (e, f);
    e.tag ← new;
ComputeFlowMod(e.next, tables);
```

If such flow entries exist[7], the algorithm picks one and fills in the missing open entries. If the flow entry contains any instructions that sets an open value in the next vertex, it sets this value forward (changing the corresponding open match field in the next vertex into a set field). If such flow entry does not exist, FlowConvertor creates one, creating a *combined* unused value to fill the still open match fields. Note that when creating a new flow entry, as it uses new values for some open match fields, backtracking is necessary, as instructions to set the corresponding fields for the packet (either setting the metadata and/or creating null operations) must be inserted in the previous flow entry (see Algorithm 2). When this recursive backtracking finishes, FlowConvertor has found or created a flow entry for the last vertex of the path and all new flow entries created can be issued to the switch[8].

To delete a flow entry, after generating and filtering the paths through the entry (sections III-B2 and III-B3), the algorithm also removes all the remaining paths from the ranked record (section III-B4) and removes the corresponding vertex from the source graph (section III-B1).

Then, for each path, it removes from the target pipeline graph, all the vertices that contain at least one of the removed matched field (contained in the removed source flow entry). Any other vertex in the path is only removed if its in-degree or out-degree (i.e. number of incoming or outgoing edges) is null. For each removed target graph vertex, a corresponding `flow-mod` message is issued to the target pipeline.

IV. IMPLEMENTATION

A. Prototype implementation

Our prototype implementation is a fairly straightforward implementation of the algorithm described in section III.

[7]Because of the ignored open fields and corresponding ignored instructions, several such entries may match the vertex.

[8]In practice, mapping can still fail due to lack of table space in the switch.

Algorithm 2: BACKTRACK(e, f)

```
Input: e: the current vertex in the target path
Input: f: fields that e has filled in with some values recently
if e.table == 0 then
    ret ← SetValue (e, f, wild-card);
    if ret == failed then
        return failed;
else
    e' ← e.pre;
    if e' == NULL then
        ret ← CreateEntry (e', 0, wild-cards);
        if ret == failed then
            return failed;
        Set the instructions of e' based on f;
        e'.tag ← new;
    else
        if e'.tag == new then
            Set the instructions of e' based on f;
        else
            e'.tag ← new;
            Reset (e'.metadata, new_value);
            BackTrack (e', e'.metadata);
```

However, we do use a compressed representation of the DAGs to reduce the memory footprint and speed up some operations.

Recall that these directed graphs reflect the structure of `goto-table` and `write-metadata` instructions, as well as metadata match field values (see section III-B1). This has two direct consequences for the vertices representing the flow entries of a given table:

1) all the vertices with equal metadata match field value have the same upstream neighbors.
2) all the vertices with identical `goto-table` and `write-metadata` instructions have the same downstream neighbors.

B. Deployment

Our FlowConvertor can be placed anywhere in-between the application and the switches. In order to facilitate the deployment of FlowConvertor in the network with OpenFlow-enabled switches, and to use FlowConvertor with unmodified SDN applications and controllers, we implemented it as a proxy process between a controller and underlying switches. In such a scenario, SDN applications running on a controller use its APIs to manipulate forwarding rules to switches, issuing `flow-mod` (in actual fact, OFPT_FLOW_MOD flow table modification messages) messages. Our shim layer will intercept these messages and redirect them to the core modules of our FlowConvertor for rule conversion.

Although all northbound and southbound control plane messages may pass through FlowConvertor, it is not a performance bottleneck even in large networks. This is because that our FlowConvertor does not need any global knowledge to modify control plane messages. Instead, it is interested only in flow table modification messages. Furthermore, an independent FlowConvertor instance is run for each individual switch, making it a natural fit for deployment as a cluster (rather than as a centralized entity).

V. EVALUATION

A. Experimental Setup

We ran our FlowConvertor on an octa-core Intel®Xeon®E5506 CPU, clocked at 2.13GHz. The machine is equipped with 16GB RAM and runs 64-bit Ubuntu Linux 14.04. Figure 13 illustrates three typical control flow programs described in [12], each of which shows how packets are processed by their logical tables. The pipeline as well as the rules in each logical tables are the input to FlowConvertor.

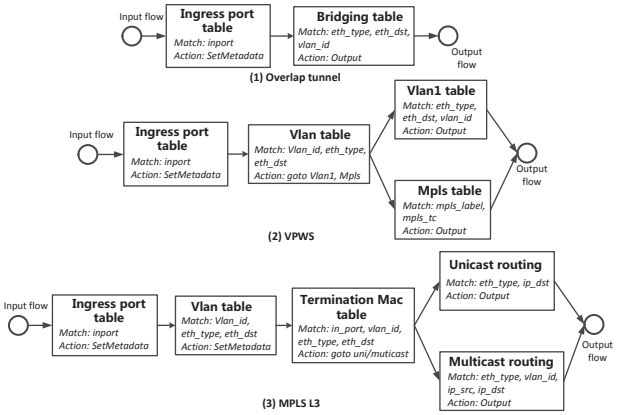

Fig. 13. Control flow programs

We generate synthetic rulesets to fill the logical tables. More specifically, we use a real ACL ruleset [7] for the rule values on five match fields (SIP, DIP, SPORT, DPORT, PROTO), and use randomly generated values for other match fields (IN_PORT, DL_TYPE, MAC_SRC, MAC_DST, VLAN and MPLS tag). While these rules may not be realistic, they are sufficient to verify the functionality and correctness of FlowConvertor and conduct the experiments below. In our evaluation, we use two target switches: (1) H3C switch and (2) software-based switch.

Commodity switch. We use the H3C S6800-2C (named as H3C for short) commodity OpenFlow switch[9] as the first target pipeline. H3C switch consists of two flow tables: (1) *MAC-IP* flow table; (2) *Extensibility* table, supporting 5-tuple matches (see Figure 14).

H3C S6800-2C Switch			
Table #	**Table type**	**Match fields**	**Supported actions**
0	MAC-IP Table	DST_MAC, VLAN, IP_DST	Modify MAC_SRC,MAC_DST and VLAN; Output
1	Extensibility Table	5-tuple	OpenFlow 1.3 Required Actions

Fig. 14. H3C S6800-2C OpenFlow Switch Specification

Software-based switch. To enhance our experiments, we use an open source CPqD OpenFlow software switch[10] plat-

[9]http://www.h3c.com/en/

[10]https://github.com/CPqD/ofsoftswitch13

form, also running on the same server where FlowConvertor runs. With its flexible dataplane, various target pipelines can be configured. Indeed, we use match field types (e.g. IP_SRC) from L2 to L4 to configure those pipeline tables depicted in Figure 15.

Software Switch Configuration			
Target Pipeline 1		Target Pipeline 2	
Table #	Match Fields	Table #	Match fields
0	IN_PORT, DST_MAC,SRC_MAC	0	IN_PORT, ETH_TYPE,VLAN
1	ETH_TYPE, VLAN, MPLS	1	DST_MAC, MPLS
2	IP_SRC, IP_DST	2	ETH_TYPE, IP_DST
-	-	3	IP_Proto, IP_SRC

Fig. 15. OpenFlow Software Switch Specification

B. Per-update processing time

We first study the latency introduced by the computations triggered by the reception of a `flow-mod` message.The main takeaway from the experimental results is that the latency introduced by FlowConvertor is low.

To make it more explicit, we divide the whole rule conversion process into two main logical stages: *origin rule analysis* and *rule mapping*. The origin rule analysis operates on the origin pipeline only and comprises rule generation (Section III-B2), rule filtering (Section III-B3), priority assignment (Section III-B4), and the identification of null and redundant operations (Section III-B5). The rule mapping stage does the mapping proper onto the target pipeline and consists in two main steps, namely the generation of a target path, followed by its insertion in the target graph and computation of the resulting `flow-mod` messages to issue to the target pipeline (Section III-B5). We use the three control flow programs described in Figure 13 to evaluate the cost of each step in the origin rule analysis stage. The results of this set of experiments are shown in Figure 16.

We can see that path (i.e. rule) generation and operation identification dominates the cost of this origin rule analysis. This is because these two steps need to perform some complex operations, such as maintaining internal data structures, recursion through graph edge cross-product patterns and deep analysis of every generated path. On the other hand, filtering out invalid paths and assigning target priorities only incur a relatively low overhead. We also observe that, as expected, more complex (i.e. longer) origin pipeline rules will incur increased overhead. However, this increase appears close to linear for the reasonably complex pipelines that were tested.

Next, we evaluate the overhead of the mapping phase. We utilized the aforementioned three types of target pipelines[11]. We used the same source pipeline rules (control flow program #1 in Figure 13). The results show that the overhead of inserting rules into target pipeline graph dominates that of generating the target path (for every rule). This is because rule insertion relies on an expensive recursive backtracking approach.

[11]H3C switch and the software switch pipelines.

Figure 18 shows the CDF (cumulative distribution function) of the whole per-update overhead of our FlowConvertor. We can see the 98^{th} percentile of the conversion operation only cost 22.25 μs, with a maximum of 25 μs at most. We can therefore conclude that our FlowConvertor is able to achieve rule conversion on-line and in real time.

C. Effect on network performance

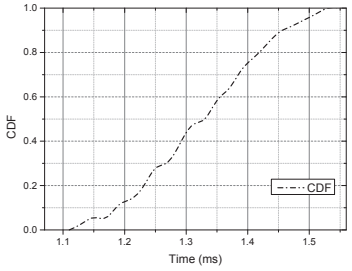

Fig. 19. The CDF of the additional overhead

Since FlowConvertor sits between a controller and a switch, we need to evaluate the impact of our FlowConvertor on the whole network, which is not limited to the overhead of the algorithm. In the real environment, it also includes the extra communication cost (a burst of `flow-mod` messages may be triggered by the original update) and the overhead of message processing (parsing OpenFlow messages). We measured the time between receiving the original `flow-mod` message, and the time when the last `flow-mod` message was issued to the switch (assuming line rate transmissions of message bursts). Figure 19 shows the CDF of the additional overhead compared with the traditional SDN architecture (where the original `flow-mod` message is received directly from the controller to the switch). We observe that this additional communication overhead is two orders of magnitude greater than the computation overhead. However, the 95^{th} percentile additional overhead of 1.5ms with FlowConvertor is still well below the acceptable flow setup time of 5-10ms for LAN environment [16]. Therefore, our FlowConvertor will not affect the network performance in an adverse way.

VI. Related Work

Controller platforms, such as Beacon [1], Floodlight [3] and Onix [17], provide low-level APIs to program the network. However, programmers have to care about many underlying details when they write SDN applications. High-level programming languages like Frenetic[14], Pyretic [21] and PGA [25], respectively design a compiler for each programming language. These programming languages make it easy to program the network. But the compilers only translate high-level policies into target-independent switch-level rules. Therefore, our FlowConvertor can act as a back-end for those compilers.

Lazaris [19] described the switch diversity. Salaheddine [26] discussed the frequent pattern of rulesets. P4 [10] is a protocol-independent programming language for programmable switch chipsets. The associated compiler tool was introduced in [18].

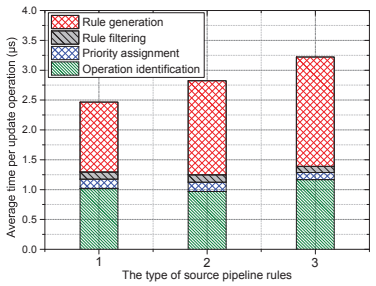

Fig. 16. The overhead of each part in the origin rule analysis stage.

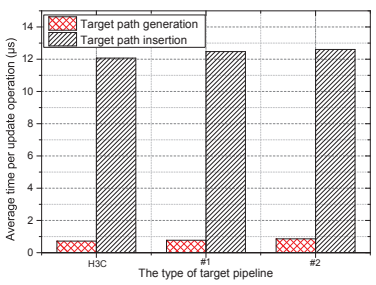

Fig. 17. The overhead of each part in the rule mapping stage

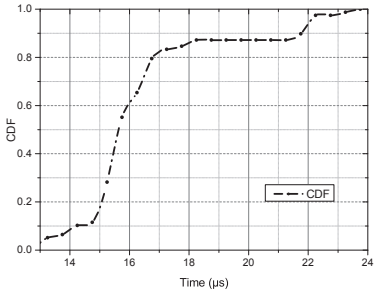

Fig. 18. The CDF of FlowConvertor overhead

However, efficient rule table mapping is still in its infancy. Open vSwitch [9] provides flexible flow processing, but with low performance for many-field packet classification [15]. Pan [24] discussed how to compute action for compositional SDN. Ennan [13] studies the independence of service deployments. RuleScope [11] tracks SDN forwarding. NOSIX [27] proposes an architecture where mapping controller-issued pipeline rules to the underlying switch hardware pipeline is possible. However, as an architecture, NOSIX does not describe algorithms to achieve this mapping, and our FlowConvertor thus complements this work. Likewise, FlowAdapter [23] proposed an adaptation layer to omit underlying hardware details, but lacked rule conversion mechanisms.

VII. Conclusion

In this paper, we have presented the principles of an online algorithm, called FlowConvertor, to translate forwarding pipeline updates from any origin pipeline to any target pipeline. Such an algorithm can obviously play a crucial role in enabling portable network applications by decoupling application logic from switch details. As such, FlowConvertor complements many existing SDN proposals. While we have described the principles of the algorithm and realized a proof-of-concept implementation, we are under no illusion that many aspects of our algorithm can be improved. Still, as the first such proposal, we believe that FlowConvertor describes the problem space, explores interesting solutions, and may inspire others to further study and contribute to the exciting topic application portability in heterogeneous SDN networks.

Acknowledgments

We thank the IEEE INFOCOM reviewers for their insightful feedback. This work is supported in part by National High Technology Research and Development Program of China (Grant No. 2015AA016101 and 2015AA010201) and National Natural Science Foundation of China (Grant No. 61502458, 61502462 and 61572475).

References

[1] Beacon. http://www.beaconcontroller.net.
[2] Cisco switches. http://www.cisco.com/.
[3] Floodlight openflow controller. http://floodlight.openflowhub.org/.
[4] The open compute project. https://www.opennetworking.org/.
[5] Openflow specification. https://www.opennetworking.org/.
[6] Openflow table type patterns. https://wiki.opendaylight.org/view/Table_Type_Patterns:Main.
[7] The rules set of evaluation packet classification. http://www.arl.wustl.edu/hs1/PClassEval.html.
[8] J. W. Andreas Voellmy, B. F. Y. Richard Yang, and P. Hudak. Maple: Simplifying sdn programming using alogrithmic polices. In *ACM SIGCOMM*, 2013.
[9] T. K. B. Pfaff, J. Pettit and E. J. Jackson. The design and implementation of open vswitch. In *NSDI*, 2015.
[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 2014.
[11] K. Bu, X. Wen, B. Yang, and e. a. Chen, Yan. Is every flow on the right track?: Inspect sdn forwarding with rulescope. In *IEEE INFOCOM*, 2016.
[12] B. Corp. Openflow data plane abstraction (of-dpa): Abstract switch specification. In *Tech. Rep.*, 2014.
[13] D. I. W. Ennan Zhai, Ruichuan Chen and B. Ford. Heading off correlated failures through independence-as-a-service. In *OSDI*, 2014.
[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.
[15] C.-L. Hsieh and N. Weng. Many-field packet classification for software-defined networking switches. In *ANCS*, 2016.
[16] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. McKeown. Maturing of openflow and software-defined networking through deployments. *Computer Networks*, 61, 2014.
[17] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.
[18] Lavanya, G. Lisa, and N. McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.
[19] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *ACM CoNEXT*, 2014.
[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.
[21] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. 2012.
[22] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *ANCS*, 2008.
[23] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie. The flowadapter: Enable flexible multi-table processing on legacy hardware. In *ACM HotSDN*, 2013.
[24] H. Pan, G. Xie, P. He, L. Zhenyu, and M. Laurent. Action computation for compositional software-defined networking. In *IFIP Networking*, 2016.
[25] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM*, 2015.
[26] H. Salaheddine, K. Blaiech, and et al. Compiling packet forwarding rules for switch pipelined architecture. In *IEEE INFOCOM*, 2016.
[27] M. Yu, A. Wundsam, and M. Raju. Nosix: A lightweight portability layer for the sdn os. *ACM SIGCOMM CCR*, 44(2), 2014.