Shoehorn: Towards Portable P4 for Low Cost Hardware

Christopher Lorier, Matthew Luckie, Marinho Barcellos, Richard Nelson
The University of Waikato, Hamilton, New Zealand
clorier@wand.net.nz, mjl@wand.net.nz, marinho.barcellos@waikato.ac.nz, richardn@.wand.net.nz

***Abstract*—Having a consistent application stack for hardware from multiple vendors allows operators to build simpler, more stable networks. However, due to the limitations of low cost, fixed-function networking hardware, creating portable control-plane software with current SDN standards requires accepting limited functionality, navigating inconsistent implementations, or using powerful, flexible hardware that is prohibitively expensive in many scenarios.**

This paper describes Shoehorn, a system for creating portable SDN control-plane software for low cost hardware. Shoehorn allows control-plane software to define a hardware-agnostic virtual pipeline in P4 that can be algorithmically translated to control diverse low-cost hardware without a significant impact on memory usage or the rate that tables can be updated.

To demonstrate the effectiveness of Shoehorn, we created virtual pipelines for a variety of existing control-plane software, and mapped them to low cost hardware. We found that the virtual pipelines could be supported by hardware from multiple vendors in almost all cases.

***Index Terms*—SDN, P4, OpenFlow**

I. Introduction

A key benefit of separating the control and data planes of network devices is allowing a single SDN controller[1] to control diverse hardware from a variety of vendors. Portable SDN gives operators the ability to run the software of their choice on the hardware of their choice, allowing for cost-savings, reliability, resilience of supply, and other advantages.

Fixed-function ASICs are widely used in low-cost hardware to support the packet processing rates required for high speed networking. These ASICs can process millions of packets per second cheaply, both in terms of production cost and power consumption. However, ASICs designed by different vendors have subtle differences that hinder the development of portable SDN software.

SDN standards such as the Portable Switch Architecture (PSA) [4] and OpenFlow [5] ostensibly support portable controllers. Both standards allow controllers to define pipelines of arbitrary match–action lookup tables. Entries in the tables specify the actions datapaths apply to matching packets, including modifying header fields or outputting the packet, and the path through the pipeline is determined based on the outcome of table lookups.

However, pipelines defined by controllers may be incompatible with those used by fixed-function ASICs. Programmable ASICs, such as the Intel Tofino [6] or the Mellanox NP-5 [7] have the flexibility to support arbitrary pipelines, but are prohibitively expensive for many applications. Vendors of low cost hardware often place limitations on the entries each table supports, forcing controllers to conform to the ASIC pipeline. As the pipelines of each vendor's ASICs are different, controllers must tailor their behaviour to individual pipelines, inhibiting portability.

To simplify the creation of portable controllers, researchers have proposed portability layers between the control and data planes [8]–[10]. Controllers define a virtual pipeline indicating their intended functionality, which is mapped to the physical pipeline, either by vendor provided drivers [8] or by a generic mapping algorithm [9], [10].

Current mapping algorithms focus on mapping in the general case, which comes at a significant performance cost. Sanger et al. [10] and Pan et al. [9] both presented approaches that convert a virtual pipeline to a graph, and then map the graph to the target physical pipeline. These approaches may increase the number of table entries in the physical pipeline. In the worst case, combining multiple virtual tables into one physical table can require an entry in the physical table for every combination of entries in the virtual tables, resulting in an exponential increase in memory usage and the time taken to add or delete entries. This is an unacceptable performance burden when using the mapping for real-time translation between the two pipelines.

We propose a much simpler method: only map tables in the virtual pipeline to tables in the physical pipeline that are capable of completely supporting the same set of entries. This approach means that updating any entry in the virtual pipeline requires only updating a single entry in the physical pipeline, ensuring real-time translation has no significant performance impact. While our approach is less likely to find a successful mapping in the general case, in practice, most pipelines—whether for hardware or controllers—tend to use similar tables. We further augment our approach by recirculating packets so they can be processed an additional time, which can work around otherwise incompatible table orderings, and allows controllers to modify packets throughout the virtual pipeline.

[1]We adopt a general view of the control plane to avoid limiting how Shoehorn is used. The control-plane software dictates the forwarding behaviour of a datapath in a software-defined network. It can be implemented as a controller (e.g. Faucet [1]), a controller application (e.g. OpenNetMon [2]), or a combination of the two (e.g. TouSIX [3]). For brevity, we use the term "controller" to refer to the control-plane software.

ISBN 978-3-903176-48-5

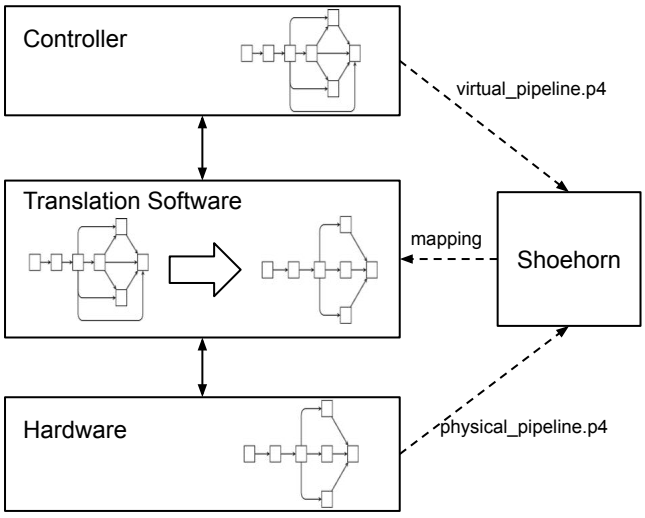

Fig. 1. A possible SDN architecture using Shoehorn. Shoehorn receives a definition of a virtual pipeline for the controller and a physical pipeline for the hardware. Shoehorn finds a mapping between the two that translation software can use to translate control-plane messages between the two pipelines.

This paper presents Shoehorn, a system for improving the portability of SDN controllers across low cost hardware. Shoehorn allows controller developers to define a virtual pipeline in P4 for their software to target. Shoehorn finds mappings between the virtual pipeline and the pipeline of a physical device, provided the physical device contains suitable tables to match each virtual table. These mappings preserve the performance of the virtual pipeline.

Shoehorn consists of two new P4 architectures: the Shoehorn Virtual Architecture (SVA) for defining virtual pipelines, and the Shoehorn Physical Architecture (SPA) for physical pipelines; an algorithm for finding mappings between pipelines defined in the two architectures; and a proof-of-concept implementation of that algorithm that we use to demonstrate its effectiveness.

Fig. 1 shows a hypothetical SDN architecture using Shoehorn. The controller defines its virtual pipeline and has no knowledge of the details of the physical device. Shoehorn receives the definitions of the virtual and physical pipelines and finds a mapping between them. Independent translation software intercepts control-plane messages and uses the mapping found by Shoehorn to translate the messages to produce identical behaviour from the virtual and physical pipelines. The design of the translation software is beyond the scope of this work as it is dependent on the control protocol, of which Shoehorn does not require any knowledge.

We evaluate Shoehorn by creating P4 definitions in the SVA of 23 SDN controller pipelines from research and production deployments, and use Shoehorn to map them to a variety of hardware pipelines (defined in the SPA). All but four of the tested pipelines were able to be supported by hardware from at least two vendors.

The rest of this paper is structured as follows: §II reviews details of SDN standards strictly relevant to this work; §III provides an overview of Shoehorn; §IV presents the two Shoehorn P4 Architectures; §V describes the algorithm used by Shoehorn to find mappings between virtual and physical pipelines; §VI demonstrates the effectiveness of Shoehorn by mapping existing controllers to real hardware and discusses the results; §VII compares Shoehorn with related work; and §VIII summarises the work and the results.

II. Background

A. OpenFlow

OpenFlow [5] abstracts the forwarding behaviour of a datapath as a pipeline of match–action tables. Controllers modify the behaviour of datapaths by adding and removing entries in these tables. The datapath applies the first table to all packets and applies subsequent tables based on which table entries the packet matches.

Table entries have a set of matches, a set of instructions, and a priority. When a datapath applies a table to a packet, the table returns the instructions associated with the highest priority matching entry. Each match has a masked value for a header field of the packet. The entry matches packets with masked header fields equal to each match value. The instructions are the actions taken by the datapath after a table look-up. Instructions include modifying packet header fields, outputting the packet, dropping the packet, sending the packet to the controller, or applying a subsequent table to the packet.

B. Switch Abstraction Interface

The Switch Abstraction Interface (SAI) [11] is a widely supported SDN standard that provides a vendor-agnostic API to a variety of datapaths. The SAI is a C-style interface to control a consistent, fixed-function pipeline providing basic networking functions such as switching, routing, QoS and ACLs. SAI is able to provide portability using vendor-provided drivers by limiting the functionality it exposes.

C. P4

P4 [12] is a programming language for specifying the behaviour of a datapath. P4 allows programmers to define the behaviour of the parser, which extracts header fields from the packet; the deparser, which reconstructs packets for output; the control flow; any match–action tables; and other aspects of a datapath with much more detail than OpenFlow. The match–action tables work similarly to tables in OpenFlow, but because P4 allows developers to define the control flow, pipelines can be designed more efficiently. For instance, determining the IP version of a packet can be performed with a switch statement rather than a table.

Match Kinds. In P4, match–action table definitions include a dictionary of header field to *match kind* mappings. The match kind indicates the type of memory used for matches of that field. The core P4 library defines three match kinds: `exact`, for matches that cannot be masked; `lpm`, for matches that use a longest prefix mask; and `ternary`, for matches that use arbitrary masks. For most hardware, match kinds affect the update rate and power consumption.

extern Objects. P4 supports control of queues and stateful functions (such as registers or counters) using extern objects. This allows P4 to support any feature of a datapath not supported natively in P4. externs define a programmable interface to a feature, but none of the functionality.

P4 Architectures. P4 allows programming many components of datapaths, but how the various components fit together is defined by a P4 architecture. P4 architectures provide the interface to a specific datapath, any externs, and the programmable blocks and their restrictions [12].

III. Overview

To map virtual pipelines to diverse fixed-function physical pipelines, Shoehorn relies on two key observations. The first is that most controllers and hardware use similar tables. In particular, tables with large numbers of entries that see the most updates tend to perform common tasks, such as Ethernet switching, IP routing, or 5-tuple matching. While implementations can have subtle variations, for most uses they are similar enough to support the same entries.

The second key observation is that most table access control is very simple. Both physical and virtual pipelines usually apply tables to every packet, every packet of a given protocol (e.g. every IPv6 packet), or every packet that matches one other table (e.g. a Termination MAC address table determining access to a routing table). This greatly simplifies the process of ensuring mapped tables are applied to the correct set of packets.

These observations are generalisations, they are not sufficient to guarantee portability. Shoehorn will be unable to find a mapping when a controller requires a feature or a table that a physical device does not support. Shoehorn can be used to verify the suitability of a device for a given controller.

A physical pipeline can support a virtual pipeline when the physical pipeline has a suitable table for each virtual table, and the pipeline can apply those tables in an equivalent order to the correct set of packets. When the hardware and virtual pipelines apply tables in an incompatible order, Shoehorn will attempt to correct the order by recirculating the packet. This reduces the overall throughput of the switch, but in enterprise networks, bandwidth is often constrained by other factors such as uplink bandwidth rather than overall switch throughput.

The components of Shoehorn are:

Shoehorn Physical Architecture (SPA): a P4 architecture for describing the pipeline of a physical device.

Shoehorn Virtual Architecture (SVA): a P4 architecture for describing the pipeline required by controllers.

Shoehorn Mapping Algorithm: for finding mappings between a pipeline defined in the virtual architecture and a pipeline defined in the physical architecture.

Shoehorn targets enterprise hardware using ASICs with published support for either multi-table OpenFlow 1.3 or the SAI. The target hardware is: hardware supporting the SAI, hardware with Broadcom ASICs supporting the OpenFlow Data Plane Abstraction (OF-DPA) [13], Mellanox hardware supporting the Onyx OS [14], Cisco Catalyst 9000 hardware [15], and HPe Aruba hardware supporting OpenFlow [16].

The SAI and OF-DPA pipelines are both strictly specified fixed-function pipelines. Cisco and Aruba both feature pipelines of flexible tables that can be preconfigured to support any matches, masks, actions, and next tables. The Mellanox pipeline is a hybrid of both approaches with a fixed-function pipeline including a series of configurable ACL tables. However, unlike the Cisco and Aruba pipelines, the Mellanox ACL tables only use ternary match kinds.

IV. Shoehorn P4 Architectures

This section describes the SVA and SPA. The SVA is used by controller developers to define the pipeline that their software requires, while the SPA is used by hardware vendors to define the pipeline used by their hardware. We designed the architectures to maximise the likelihood that Shoehorn can find mappings between virtual and physical pipelines. For simplicity, when a design decision is arbitrary, the architectures follow the PSA [4].

Parser. Both Shoehorn architectures specify a fixed parser. Fixed-function ASICs are not capable of supporting arbitrary parsers. When an application requires bespoke headers, more flexible P4 hardware should be used. The parser is modelled on the OpenFlow specification version 1.3 [17], and extracts headers based on the match fields used in OpenFlow, as well as setting metadata fields based on those used in OpenFlow, such as ip_proto.

Metadata. Retaining metadata is challenging when recirculating a packet. Hardware may be restricted in the number of fields, and the number of bits of metadata it can recirculate. Because of this, Shoehorn does not allow user-defined Metadata in the SVA. This is a limitation of Shoehorn: VRFs, for instance, cannot be supported by Shoehorn.

Metadata for both architectures includes the ingress and egress port, and the SPA includes the number of times a packet has been recirculated. The recirculation count is necessary for the physical pipeline to be able to apply tables correctly, and the ingress port is one of the most frequently matched fields. To reduce metadata requirements when recirculating, Shoehorn overwrites the ingress port with the egress port once a forwarding decision is made.

Actions. Actions in Shoehorn are closely tied to the actions used in OpenFlow. The SVA and SPA provide primitive action externs modelled on OpenFlow actions, and to simplify the mapping process, user-defined actions are not allowed control flow logic.

ActionModules. In the SVA and SPA, actions are applied by new externs called ActionModules, which can be instantiated and called within control blocks. ActionModules define where in a pipeline actions will be applied. As recirculation is only necessary when a packet is modified, this allows controller developers to specify when a recirculation can occur.

To maximise the flexibility when reordering tables, the Shoehorn architectures do not impose an order in which ac-

tions are applied to packets. For instance, if a packet has a drop and a notify controller action applied in two separate tables in the same action module, the packet may be dropped before it reaches the table with the notify controller action. If an application requires that actions should be applied in a specific order, the tables should be separated by an ActionModule.

Goto Metadata. Some target hardware allows tables to be linked arbitrarily. To support this scenario, the SPA uses the goto metadata field. goto metadata can be written by the goto action and can be read in conditional statements to control access to tables. The goto action and metadata fields do not exist in the SVA.

Match Kinds. The SVA uses the match kinds defined in the P4 core library: exact, lpm, and ternary, as well as an all_or_exact match kind. The all_or_exact match kind is taken from the OpenFlow Table-Type Patterns Specification [18]. The SVA also includes an annotation that indicates that a match with an exact or lpm match may be supported by a match with a different match kind.

Flexible tables may match any header field with any match kind and can apply any combination of actions. To allow the definition of flexible tables, the SPA has new match kinds. In addition to the match kinds used in the SVA, the SPA adds a configured counterpart to each, as well as a configured_any match kind. The configured match kinds indicate that the table can support virtual tables that match the given field with the corresponding match kind, or a table where that match is absent. The configured_any match kind indicates the table can match the given field with any match kind.

V. Mapping

Shoehorn finds mappings from a virtual to a physical pipeline, where each table in the virtual pipeline maps directly to a table in the physical pipeline. This ensures that any update to a table entry in the virtual pipeline requires translation software to only update a single entry in the physical pipeline. Tables in the physical pipeline can support multiple virtual components (tables and conditional statements). Conditionals cannot be updated and only ever match one value, so mapping a virtual conditional to multiple physical components does not significantly affect the update rate or memory usage.

Shoehorn's mapping algorithm takes a compiled representation of the physical and virtual pipelines. This consists of a list of component trees, each representing the components preceding an ActionModule in the P4 code.

The procedure for finding mappings occurs in two stages. In the first stage, Shoehorn identifies all potential component mappings for each component in the virtual pipeline without consideration of the layout of the pipelines. In the second stage, Shoehorn finds mappings that ensure that each table is applied to the correct set of packets.

A. Stage 1: Identifying Supporting Components

To begin with, Shoehorn identifies every component in the physical pipeline that can support each component in the virtual pipeline without considering the pipeline layout. To be able to support a virtual component, a physical component needs to support all the actions and matches used by the virtual component. Virtual conditionals can be supported by a physical conditional or a physical table, but virtual tables can only be supported by physical tables.

A physical match can always support a virtual match if they match the same field and have the same match kind (or if the physical match has a corresponding configured match kind). While it is possible to support a virtual exact, or lpm match with a physical ternary match, this may increase the power consumption and may cause a considerable reduction in the update rate, which could prevent real time translation. Consequently, Shoehorn only allows such a mapping when explicitly directed with an annotation. Shoehorn always allows all_or_exact matches to be mapped to ternary matches.

The speed at which hardware can update tables depends on the slowest match kind used in that table. For example, if a table has an exact match on Ethernet type and a ternary match on IPv6 destination, the update rate will be limited by the ternary match. Therefore, such a virtual table can be mapped to a physical table that has a ternary match on both Ethernet type and IPv6 destination.

Shoehorn will accept Physical tables that match additional fields, as those fields could be filled by aggregating with another component. For instance, a physical table that has an exact match on Ethernet type and IPv6 destination will still potentially be able to support a virtual table that only matches IPv6 destination, as the virtual table may be aggregated with another component that matches Ethernet type.

B. Stage 2: Finding Mappings

Stage 2 finds the final mappings with correct access control. Shoehorn iteratively attempts to map each virtual ActionModule to ActionModules in the physical pipeline. ActionModules apply actions before any subsequent tables are applied, so a physical ActionModule can only support one virtual ActionModule in each recirculation. On the other hand, because ActionModules apply actions in an undefined order, an ActionModule in the virtual pipeline can be mapped to multiple ActionModules in the physical pipeline.

Shoehorn starts by mapping the components associated with the first ActionModule in the virtual pipeline to components associated with the first ActionModule in the physical pipeline. If there are remaining unmapped components Shoehorn moves onto the next physical ActionModule and repeats the process. This continues until all the virtual components are mapped or Shoehorn reaches the end of the physical pipeline. Once Shoehorn finds a mapping for all of the components, it will continue mapping the next virtual ActionModule, starting with the next ActionModule in the physical pipeline. If Shoehorn reaches the end of the physical pipeline without mapping all of the virtual components, it will recirculate and start again from the first physical ActionModule. This process ends when Shoehorn finds a mapping for all components in the virtual

Algorithm 1 Map components from a virtual module to a physical module

```
1:  function MAP_MODS(cand_l, p_mod_l, v_mod_l, limit)
                ▷ cand_l: partially complete candidate mappings
                    ▷ p_mod_l: physical module components
                      ▷ v_mod_l: virtual module components
                    ▷ limit: int, the maximum size of cand_l
2:      for each component pc in p_mod_l do
3:          for each component vc in pc.supports() do
4:              for each candidate in cand_l do
5:                  new ⇐ candidate.clone();
6:                  new.map_component(vc, pc);          ▷ §V-B1
7:                  new.check_access_ctrl()             ▷ §V-B2
8:                  if new is valid then
9:                      cand_l.add(new);
10:         remove_invalid_match_kinds(cand_l);         ▷ §V-B3
11:         prune_strictly_outclassed(cand_l);          ▷ §V-B4
12:     prune_partially_outclassed(cand_l);             ▷ §V-B5
13:     prune(cand_l, limit);                           ▷ §V-B6
        return cand_l;
```

pipeline, a full recirculation fails to map any components, or the maximum number of recirculations is reached.

Alg. 1 shows how Shoehorn maps a virtual ActionModule to a physical ActionModule. The algorithm takes a list of the current partially-complete candidate mappings, a list of the components associated with the physical ActionModule, a list of the components associated with the virtual ActionModule, and a size limit for the resulting list of candidates. The algorithm iterates through the physical components, and for each component, finds all combinations of virtual component mappings. Each time if finds a new combination, the algorithm creates new candidate mappings, adding the new combination to each partially-complete mapping. The algorithm validates the new candidates and then uses heuristics to discard all but the most promising candidates.

In the worst case, this approach scales exponentially in time and memory usage, and does not guarantee finding a correct result. In practice, however, all of the target hardware fits into two categories, both of which avoid these issues.

Fixed-function pipelines: The OF-DPA and SAI pipelines are both so restrictive that the number of possible mappings for any given table is small enough that the number of candidate mappings never grows unmanageably.

Configurable pipelines: The Cisco, Aruba, and Mellanox pipelines all use a series of identical, configurable tables that can be arranged arbitrarily with goto instructions. Because the tables are identical, all valid mappings that map the same tables are equivalent, allowing the majority of candidates to be discarded.

The steps of Alg. 1 are discussed next (§V-B1 to §V-B6).

1) Map Component (Alg. 1, L6): Candidate mappings contain trees of all the virtual components mapped to each physical component, referred to as the *entry tree*s. When a new

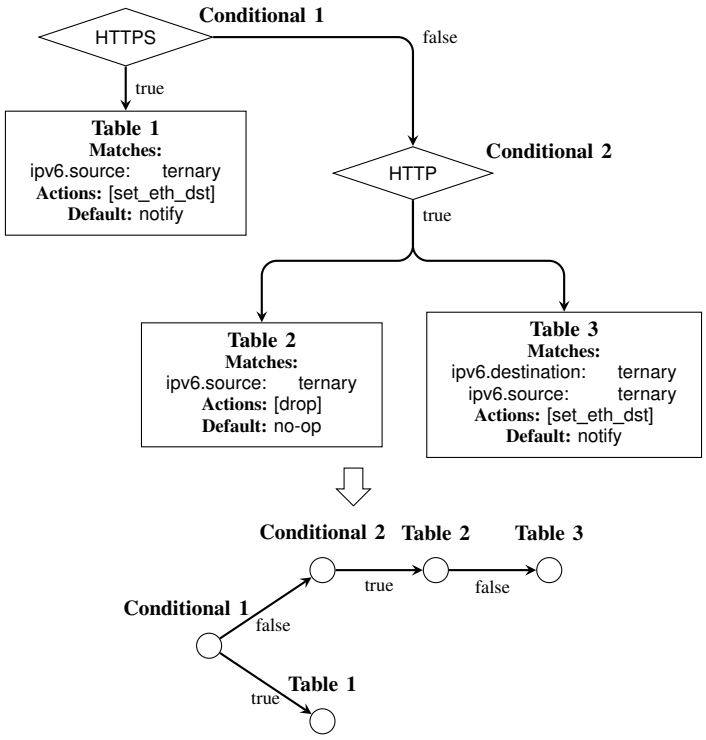

Fig. 2. A section of a virtual pipeline that is aggregated into a single physical table, and the corresponding entry tree. Nodes in the entry tree can only have one child on a true (or false) evaluation. However, as Table 2 drops all packets it matches, it can still be merged with Table 3. Table 2 becomes the immediate child of Conditional 2, and Table 3 becomes the child of Table 2.

Table 1

IP Source:	Actions:	Priority:
2001:db8:1::/48	set_eth_dst	100
2001:db8:2::/48	set_eth_dst	100
::/0	notify	0

Table 2

IP Source:	Actions:	Priority:
2001:db8:1::2	drop	100
::/0	no-op	0

Table 3

IP Source:	IP Destination:	Actions:	Priority:
2001:db8:1::/48	2001:db8:3::1	set_eth_dst	100
::/0	::/0	notify	0

Mapped Table

IP Source:	IP Destination:	Port:	Actions:	Priority:
2001:db8:1::/48	::/0	443	set_eth_dst	500
2001:db8:2::/48	::/0	443	set_eth_dst	500
::/0	::/0	443	notify	400
2001:db8:1::2	::/0	80	drop	300
2001:db8:1::/48	2001:db8:3::1	80	set_eth_dst	200
::/0	::/0	80	notify	100
::/0	::/0	ANY	no-op	0

Fig. 3. A demonstration of merging tables. This shows hypothetical table entries for the section of virtual pipeline shown in Fig. 2 and the corresponding entries for the mapped table. Entry priority in the mapped table is determined by the path through the entry tree. The highest priority entries belong to Table 1, followed by the Table 1 default entry. The next highest priority entries are from Table 2, followed by its default (in this case a no-op) combined with the entries from Table 3, and finally a no-op entry for packets that return False when applied to Conditional 2.

virtual component, *vc*, is mapped to a physical component, *pc*, it is added to the entry tree for that physical component. If, after adding *vc*, the entry tree is invalid, then the candidate mapping is invalid.

Fig. 2 shows how an entry tree corresponds to a section of a virtual pipeline. Each node in an entry tree represents a virtual component and can have up to two child nodes, representing the components applied following a true or false evaluation of the parent (for tables the evaluation represents whether the packet matched a table entry). While a virtual pipeline may have multiple components accessed by a true or false evaluation for a single component, they generally cannot be aggregated together. The exception is tables that always drop packets on a match. Such tables are inserted as the immediate child, and the other component becomes its child following a miss. Only conditionals can have a child following a true evaluation, because a child of a table after a match would require an entry in the physical table for each combination of entries in the two virtual tables.

Fig. 3 demonstrates how Shoehorn populates entries for an aggregated table. Every node in the entry tree without two children corresponds to an entry type in the mapped table, matching the combined fields of all components that evaluate true in its path from the root. Shoehorn determines the priority offset for entries by the path through the tree: all entries from nodes descending from a true evaluation of given node have higher priorities than all entries from nodes descending from a false evaluation.

2) Check Access Control (Alg. 1, L7–9): Once Shoehorn finds a potential mapping for a component, it verifies the access control is consistent. Access to tables in the SVA is controlled in one of two ways: either by a conditional, where a metadata or packet header field must be equal (or not) to a specified value; or by matching an entry (or not) in a table. The set of packets that a table will apply to is defined by a list of masked field values and tables that the packet must match (or not).

If the hardware has a configurable pipeline, then Shoehorn only needs to verify it has successfully mapped all components in the access control set. If the hardware has a fixed pipeline, then Shoehorn finds the path through the entry trees to reach the target physical component. Shoehorn verifies that the set of virtual components in the path through the entry trees to reach the target component is equivalent to the access control set of the virtual component.

Shoehorn also verifies that any potential children of the mapped components are reachable. If the physical table does not support arbitrary goto actions, Shoehorn will check whether there is a mapped virtual table that controls access to another table. In that case, the entry tree must have no children on a false evaluation of any node. If there is such a child, and all of its descendants are conditionals, then Shoehorn removes the child and its descendants. If there is a table that is descended from a false evaluation then the candidate mapping is invalid. If the entry tree and access control are valid, the candidate, new, is added to the list of candidates, $cand\_l$.

3) Remove Invalid Match Kinds (Alg. 1, L10): After finding all candidate mappings for each physical table, Shoehorn checks that the match kinds for each candidate mapping do not conflict. Shoehorn only does this after all virtual components have been mapped to the physical table, as it is possible that a conflicting match kind can be resolved by aggregating a new component. If the match kinds conflict then the candidate mapping is removed from $cand\_l$.

4) Prune Strictly Outclassed Candidates (Alg. 1, L11): Shoehorn then does a first pass of pruning candidates, removing any candidate with virtual–physical component mappings that are a subset of another candidate's mappings from $cand\_l$.

5) Prune Partially Outclassed Candidates (Alg. 1, L12): Shoehorn repeats the process of mapping components until it reaches the end of an ActionModule. At that point, either the physical pipeline allows arbitrary goto actions, or, because access control cannot cross an ActionModule, the previous tables will be irrelevant for access control of future tables. Shoehorn removes from $cand\_l$ any mapping that maps a subset of virtual components mapped by another candidate regardless of how they are mapped.

6) Final Pruning (Alg. 1, L13): At this stage, fixed-function pipelines typically have very few remaining candidate mappings. However configurable pipelines may still have many candidates. In order to further decrease the workload, Shoehorn reduces $cand\_l$ to a random set of configurable size from the candidates that map the largest number of components. The size of the set is a trade-off between run time and the likelihood of finding a complete mapping, but we have found that the size of the set has little impact on the likelihood of success except when $cand\_l$ is less than 100.

VI. Evaluation

To evaluate Shoehorn, we mapped a variety of virtual pipelines to physical pipelines based on Shoehorn's target hardware. The virtual pipelines are based on twenty-three SDN controllers, which we chose to reflect a variety of network types and uses. We preferred controllers with real-world production deployments. We made virtual pipelines for each controller in the SVA and then used Shoehorn to find a mapping to the physical pipelines.

A. Physical Pipelines

We implemented physical pipelines based on Shoehorn's target hardware in the SPA. However, as OpenFlow and P4 are not fully compatible, and without access to the full specification of the vendor's ASICs, we cannot guarantee our implementations are perfect recreations. Regardless, our implementations represent a variety of incompatible pipelines, demonstrating Shoehorn's ability to find mappings for diverse hardware. This section discusses some of the challenges for our implementations.

For Shoehorn to function correctly, the physical hardware must be able to keep track of how many times a packet has been recirculated, and the ingress or egress port after recirculation. As recirculation is not part of the OpenFlow standard,

TABLE I
THE NUMBER OF RECIRCULATIONS REQUIRED FOR EACH PHYSICAL PIPELINE TO SUPPORT EACH OF THE 23 CONTROLLERS' VIRTUAL PIPELINES. A DASH INDICATES THE CONTROLLER'S VIRTUAL PIPELINE CANNOT BE SUPPORTED.

Controller	Description	SAI	Aruba	Cisco	OF-DPA	Mellanox
AuthFlow [19]	A host authentication mechanism for enterprise networks.	3	0	0	3	-
Castor [20]	A SDN Internet exchange interconnect that provides telemetry and ARP hygiene.	-	0	0	-	-
Faucet [1]	A widely-deployed enterprise controller performing switching and routing.	3	-	0	3	-
Hierarchical SDN [21]	An architecture for data centre networks for scalable TE.	-	-	-	0	-
In-Packet Bloom Filters [22]	An architecture for data centres using bloom filters encoded in Ethernet addresses to load balance traffic.	1	0	0	1	-
iTelescope [23]	A-bump-in-the-wire system for identifying video and providing telemetry.	-	0	0	-	-
Magneto [24]	A system for providing fine-grained path control in a hybrid legacy–OpenFlow enterprise network, by manipulating MAC learning on the legacy devices.	0	-	0	0	-
NetPaxos [25]	A system for accelerating consensus Paxos protocols in data centres by creating a canonical ordering of messages within the network.	0	0	0	0	-
NFShunt [26]	A system for accelerating Netfilter in hardware for use in a Science DMZ.	-	0	0	-	-
OF-Like PBR [27]	A system for applying policy in a hybrid legacy–OpenFlow network. It uses OpenFlow switches to rewrite IP destinations to control how the legacy devices forward traffic.	-	0	-	-	-
OFLoad [28]	A load balancing system for data centres that generates tunnels for elephant flows.	-	0	0	-	-
OFTDP [29]	A topology discovery technique for SDN.	0	0	0	0	0
OLiMPS [30]	A system for establishing and load balancing across point-to-point VLANs for use in research networks.	0	0	0	0	0
OpenNetMon [2]	A loss monitoring system in reactive OpenFlow networks.	0	-	-	0	0
Precision Medicine [31]	A Science DMZ for campus networks for medical applications.	1	0	0	1	-
Random Host Mutation [32]	A security system for enterprise networks that randomly rewrites IP addresses.	-	0	-	-	-
RouteFlow [33]	An SDN router for OpenFlow v1.0. Our implementation in the SVA uses lpm match kinds to improve scalability, rather than the original single table design.	0	-	0	0	-
SciPass [34]	A Science DMZ for campus networks.	-	0	0	-	-
SDProber [35]	A system for monitoring link latency in SDNs.	0	-	0	0	-
SIMPLE [36]	A system for enforcing middle box policies in enterprise networks.	-	0	0	-	-
TouSIX [3]	A Production deployment at an Internet exchange in Toulouse. The TouSIX deployment provides layer 2 security and fine-grained monitoring.	-	0	0	-	-
VIP Lanes [37]	An architecture for campus networks that allows creation of on-demand Science DMZs.	3	-	0	3	-
VPNs [38]	A system for simplifying the configuration of VPNs in SDN data centres. There are two types of devices described in the paper, a P node and a PE node. However the PE node requires VRF metadata and therefore cannot be easily supported in the SVA. The SAI, Aruba, OF-DPA and Mellanox can all support VRF metadata, but it is unclear whether metadata can be preserved between recirculations.	-	-	-	0	-
	Total number of controllers supported	12	15	18	14	3

we made assumptions as to how it can be achieved. For all implementations, we assumed that the ingress and egress port metadata could be recirculated and that the recirculation metadata could be inferred from the ingress port or tunnel ID. If there was no better alternative, recirculation could be achieved with tunnels over loopback cables.

Cisco and Aruba. Flexible pipelines such as the Cisco and Aruba are easy to express in the SPA. We have assumed both pipelines are able to perform lpm match kinds on IP fields as this is not expressible in OpenFlow.

SAI. The SAI has an existing P4 definition for the behavioural model architecture, making translation into Shoehorn simple. The only limitation of our implementation is the SAI's use of user-defined metadata, currently unsupported in the SPA.

OF-DPA. The OF-DPA pipeline is well specified, and requires few modifications to be expressed in the SPA. The most significant change is the MAC Learning table. The OF-DPA uses a vendor extension to allow the Bridging table to be looked up twice, with both the Ethernet source address and the Ethernet destination address of a packet. For the purpose of this evaluation, we assumed that we could add separate entries to the Bridging table and the MAC Learning table. It may be possible to recreate this behaviour by rewriting VLANs internally, or with another similar method.

Mellanox. The Mellanox OpenFlow pipeline frequently redirects traffic to the legacy pipeline for additional processing. As the legacy pipeline is not clearly defined, our implementation omits that functionality.

B. Virtual Pipelines

We implemented a virtual pipeline for each controller in the SVA, based on a best-effort interpretation of the published details. We required that the SDN controllers used features compatible with OpenFlow version 1.3. When a controller sends a packet to be processed by the hardware's legacy pipeline, the Faucet [1] pipeline was used in its place, as Faucet is an OpenFlow implementation of standard switch functions.

C. Results

Table I shows support for each virtual pipeline by the target hardware. Excluding the Mellanox pipeline, every physical pipeline was able to support the virtual pipelines of over half of the controllers, without any customisation of the controllers to the target hardware. Furthermore, 83% of the controllers were supported by hardware from multiple vendors.

The Cisco and Aruba implementations supported 78% and 65% of the virtual pipelines, respectively, and never required recirculation. The fixed-function OF-DPA implementation supported 60% of the virtual pipelines, while the SAI supported 52%. The Mellanox OpenFlow implementation performed poorly, only supporting three controllers, as its pipeline is based predominantly on ternary matches. The Mellanox hardware also supports SAI, and therefore is more capable than these results show.

In every case where Shoehorn was unable to find a successful mapping, the controller required a table that was unable to be supported by the physical pipeline. This demonstrates that the observations in §III, coupled with packet recirculation, give Shoehorn *enough flexibility to allow portable SDN controllers for fixed-function hardware, provided the hardware is capable of supporting the controller in question*.

Notable reasons for incompatibility between virtual and physical pipelines were:

- SAI, Mellanox, and OF-DPA cannot do exact matches except on specific fields. This is most notable when virtual pipelines match individual flow 5-tuples. This is a common use-case, and it seems likely that the hardware is capable of supporting such matches, but it is not exposed in their OpenFlow support. iTelescope, NFShunt, OFLoad, SciPass and SIMPLE use these matches.
- OF-DPA is the only physical pipeline to support MPLS, and therefore is the only pipeline to support Hierarchical SDN or VPNs.
- Aruba cannot decrement TTL fields, meaning it cannot support L3 applications.
- Cisco cannot set IP source and destination fields, which is required for OF-Like PBR and Random Host Mutation.
- Castor and TouSIX both match ARP target protocol address fields to improve ARP handling in IX networks, which is only supported in the Cisco and Aruba pipelines.

Recirculations. No mapping required more than three recirculations. For many networks this would not be noticeable as throughput is often limited by other factors, eg. uplink bandwidth or firewall throughput. For example, a Broadcom Wolfhound BCM5334x series chip provides 64Gb/s of switching at line rate [39]. If the throughput bottleneck for a switch with such a chip is incoming traffic on a 10Gb/s port, then three recirculations leaves 6Gb/s of switching capacity for the remaining traffic. Even if the uplink is saturated in both directions, then three recirculations only causes an increase in the minimum line-rate packet size from 64B to 80B.

Run Time. In all cases the mapping was found (or failed) in less than five minutes on a standard PC (Intel Core i3-2120 CPU with 4GB RAM). As the mapping algorithm can run at compile time, speed is not a significant concern, and this result is satisfactory for practical use.

VII. Related Work

FlowAdapter [40], its successor FlowConvertor [9], and Sanger et al. [10] presented techniques for mapping between a virtual OpenFlow pipeline to a fixed-function physical pipeline. Shoehorn has three main advantages over these systems. First, Shoehorn uses P4 rather than OpenFlow, a more expressive standard, that allows vendors to better define the capabilities of their hardware. Second, Shoehorn ensures that the memory requirements and update rate are the same in the virtual and physical pipelines, whereas the previous work would possibly add multiple entries to the physical pipeline for a single entry in the virtual pipeline. Third, Shoehorn is capable of mapping to flexible hardware as well as fixed-function hardware. Pan et al. never tested FlowConvertor on flexible hardware, and Sanger et al. were unable to support it, as the growth in the potential solutions was too large.

Previous work improving portability of P4 focussed on programmable hardware. P4 Transformer [41] is a system for improving portability with P4 with programmable hardware, presenting techniques for supporting certain P4 operations on hardware that does not support them natively. Hyper4 [42] is a P4 program that can be used as a target for other P4 programs, improving portability and composability on programmable hardware. MACSAD [43] is a system for controlling OpenDataPlane [44], a portable API for networking software.

VIII. Conclusion

Shoehorn enables the creation of portable SDN control-plane software by allowing developers to define a hardware-agnostic virtual pipeline for their controller to target. We presented two new P4 architectures, one for defining virtual pipelines, and one for defining physical pipelines; as well as an algorithm for finding mappings between them.

We demonstrated that for a wide variety of controllers, Shoehorn is able to find a mapping from the virtual pipeline to multiple diverse, low cost hardware pipelines. Of twenty-three controllers tested, Shoehorn found mappings for at least one physical pipeline in all cases, and all but four were portable across multiple pipelines.

In every case where Shoehorn was unable to find a mapping, it was because of a table in the virtual pipeline that could not be supported by the physical pipeline. The control flow of the pipeline was never a factor in a failed mapping. This demonstrates that Shoehorn enables fixed function pipelines to support portable SDN controllers provided: the hardware is able to recirculate packets efficiently while retaining a small amount of metadata, and the hardware supports the individual tables used by the controller.

References

[1] J. Bailey and S. Stuart, "Faucet: Deploying SDN in the enterprise," *Communications of the ACM*, vol. 60, no. 1, pp. 45–49, 2016.

[2] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[3] R. Lapeyrade, M. Bruyère, and P. Owezarski, "OpenFlow-based migration and management of the TouIX IXP," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 1131–1136.

[4] P4.org Architecture Working Group. (2021, 2021-04-02) $P4_16$ portable switch architecture (PSA). [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[5] 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 computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] A. Agrawal and C. Kim, "Intel Tofino2-A 12.9 Tbps P4-Programmable Ethernet Switch," in *Hot Chips Symposium*, 2020, pp. 1–32.

[7] Mellanox Technologies, "NP-5™Network Processor," Product Brief, 2019.

[8] M. Yu, A. Wundsam, and M. Raju, "NOSIX: A lightweight portability layer for the SDN OS," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 28–35, 2014.

[9] H. Pan, G. Xie, Z. Li, P. He, and L. Mathy, "Flowconvertor: Enabling portability of SDN applications," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[10] R. Sanger, M. Luckie, and R. Nelson, "Towards transforming OpenFlow rulesets to fit fixed-function pipelines," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 123–134.

[11] Open Compute Project, "Switch Abstraction Interface (SAI)," Open Compute Project, Tech. Rep., 2015.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] Broadcom, "OpenFlow™data plane abstraction (OF-DPA): Abstract switch specification," Broadcom, Tech. Rep., 2014.

[14] Mellanox Technologies, *Mellanox Onyx User Manual*, 5th ed., Mellanox Technologies, 2019.

[15] Cisco Systems, Inc., *Programmability Configuration Guide, Cisco IOS XE Gibraltar 16.11.x*, Cisco Systems, Inc., 2021.

[16] Hewlett Packard Enterprise Development LP, *Aruba OpenFlow 1.3 Administrator Guide for ArubaOS-Switch 16.07*, Hewlett Packard Enterprise, 2018.

[17] Open Networking Foundation, "OpenFlow switch specification version 1.3," Open Networking Foundation, 2012.

[18] ——, "OpenFlow table type patterns," Open Networking Foundation, Tech. Rep., 2014.

[19] D. M. F. Mattos and O. C. M. B. Duarte, "AuthFlow: authentication and access control mechanism for software defined networking," *annals of telecommunications*, vol. 71, no. 11, pp. 607–615, 2016.

[20] H. Kumar, C. Russell, V. Sivaraman, and S. Banerjee, "A software-defined flexible inter-domain interconnect using ONOS," in *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*. IEEE, 2016, pp. 43–48.

[21] L. Fang, F. Chiussi, D. Bansal, V. Gill, T. Lin, J. Cox, and G. Ratterree, "Hierarchical SDN for the hyper-scale, hyper-elastic data center and cloud," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–13.

[22] C. A. Macapuna, C. E. Rothenberg, and M. F. Maurício, "In-packet bloom filter based data center networking with distributed OpenFlow controllers," in *2010 IEEE Globecom Workshops*. IEEE, 2010, pp. 584–588.

[23] H. H. Gharakheili, M. Lyu, Y. Wang, H. Kumar, and V. Sivaraman, "iTeleScope: Intelligent video telemetry and classification in real-time using software defined networking," *arXiv preprint arXiv:1804.09914*, 2018.

[24] C. Jin, C. Lumezanu, Q. Xu, H. Mekky, Z.-L. Zhang, and G. Jiang, "Magneto: Unified fine-grained path control in legacy and OpenFlow hybrid networks," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 75–87.

[25] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–7.

[26] S. Miteff and S. Hazelhurst, "NFShunt: A linux firewall with OpenFlow-enabled hardware bypass," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, 2015, pp. 100–106.

[27] A. Mishra, D. Bansod, and K. Haribabu, "A framework for OpenFlow-like policy-based routing in hybrid software defined networks." in *INC*, 2016, pp. 97–102.

[28] R. Trestian, K. Katrinis, and G.-M. Muntean, "OFLoad: An OpenFlow-based dynamic load balancing strategy for datacenter networks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 792–803, 2017.

[29] A. Azzouni, N. T. M. Trang, R. Boutaba, and G. Pujolle, "Limitations of OpenFlow topology discovery protocol," in *2017 16th annual mediterranean Ad hoc networking workshop (Med-Hoc-Net)*. IEEE, 2017, pp. 1–3.

[30] H. B. Newman, A. Barczyk, and M. Bredel, "OLiMPS. openflow link-layer multipath switching," California Institute of Technology (CalTech), Pasadena, CA (United States), Tech. Rep., 2014.

[31] N. Pho, D. R. Magri, F. F. Redigolo, B.-D. Kim, T. Feeney, H. L. Morgan, C. J. Patel, C. Botka, and T. Cristina, "Data transfer in a science DMZ using SDN with applications for precision medicine in cloud and high-performance computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.(SC15)*, 2015, pp. 1–4.

[32] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "OpenFlow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 127–132.

[33] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. Corrêa, S. C. De Lucena, and M. F. Magalhães, "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*, 2011, pp. 34–37.

[34] E. Balas and A. Ragusa, "SciPass: a 100gbps capable secure science DMZ using OpenFlow and bro," in *Supercomputing 2014 conference (SC14)*. Supercomputing 2014 conference (SC14) New Orleans, Louisiana, 2014.

[35] S. Ramanathan, Y. Kanza, and B. Krishnamurthy, "SDProber: A software defined prober for SDN," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[36] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 27–38.

[37] J. Griffioen, K. Calvert, Z. Fei, S. Rivera, J. Chappell, M. Hayashida, C. Carpenter, Y. Song, and H. Nasir, "VIP lanes: High-speed custom communication paths for authorized flows," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9.

[38] G. Lospoto, M. Rimondini, B. G. Vignoli, and G. Di Battista, "Rethinking virtual private networks in the software-defined era," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 379–387.

[39] Broadcom, "BCM53346 64 gb/s multilayer switch product brief," Product Brief, 2020.

[40] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: Enable flexible multi-table processing on legacy hardware," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 85–90.

[41] Z. Hang, Y. Wang, and S. Huang, "P4 transformer: Towards unified programming for the data plane of software defined network," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 544–551.

[42] D. Hancock and J. Van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 35–49.

[43] 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*, 2016, pp. 623–624.

[44] "OpenDataPlane (ODP) users-guide," https://opendataplane.github.io/odp/users-guide/, accessed: 2022-02-10.