Programming Protocol-Independent Packet Processors

Article *in* ACM SIGCOMM Computer Communication Review · December 2013

DOI: 10.1145/2656877.2656890 · Source: arXiv

Nick McKeown
Stanford University
291 PUBLICATIONS **37,428** CITATIONS
SEE PROFILE

Jennifer Rexford
Princeton University
434 PUBLICATIONS **36,144** CITATIONS
SEE PROFILE

Amin Vahdat
Google Inc.
302 PUBLICATIONS **29,989** CITATIONS
SEE PROFILE

Programming Protocol-Independent Packet Processors

Pat Bosshart[†] Dan Daly[*], Martin Izzard[†], Nick McKeown[‡], Jennifer Rexford[**], Dan Talayco[†], Amin Vahdat[¶], George Varghese[§], David Walker[**]
[†]Barefoot Networks [*]Intel [‡]Stanford University [**]Princeton University [¶]Google [§]Microsoft Research

ABSTRACT

OpenFlow is a vendor-agnostic API for controlling hardware and software switches. In its current form, OpenFlow is specific to particular protocols, making it hard to add new protocol headers. It is also tied to a specific processing paradigm. In this paper we make a strawman proposal for how OpenFlow should evolve in the future, starting with the definition of an abstract forwarding model for switches. We have three goals: (1) Protocol independence: Switches should not be tied to any specific network protocols. (2) Target independence: Programmers should describe how switches are to process packets in a way that can be compiled down to any target switch that fits our abstract forwarding model. (3) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed in a network. We describe how to write programs using our abstract forwarding model and our *P4* programming language in order to configure switches and populate their forwarding tables.

1. INTRODUCTION

Software-Defined Networking (SDN) gives owners and operators programmatic control over their networks. In SDN, the control plane is physically separate from the forwarding plane, and one control plane controls multiple forwarding devices. While forwarding devices could be programmed in many ways, having a common, open, vendor-agnostic interface enables a control plane to control forwarding devices from different vendors, both hardware and software.

OpenFlow [1] was first proposed in 2008 as a way to externally control existing switches and routers using a simple match+action paradigm. From the outset, OpenFlow was a pragmatic compromise. The long-term goal was to allow a control plane to add almost arbitrary match+action flow entries. In the short-term, to ease adoption, the OpenFlow specification was limited to a small number of fixed protocol headers (e.g., Ethernet, VLAN, IPv4) processed by existing switch ASICs. Since 2011 OpenFlow has been under the stewardship of the Open Networking Foundation (ONF [2]), and is currently at version 1.4 [3].

OpenFlow faces several limitations in its current form. First, the base definition of OpenFlow is specific to particular protocols, specifying over twenty different fields for Ethernet, IPv4, MPLS, VLANs, IPv6 and so on. Over time, the specification has become heavier with each new protocol added. Adding a user-defined header field (say, to aggregate multiple flows into a bundle) requires significant effort, as well as care to avoid colliding with existing OpenFlow headers. Second, OpenFlow does not take into consideration that different switches can perform different actions, including fairly complicated actions for specific encapsulations. Third, OpenFlow does not provide a clear and consistent way for a switch to express its capabilities to the control plane (e.g., the number of match tables, their width and depth, support for user-defined headers, etc.). Recent progress with "typed tables" [4] provides some limited information to the control plane and is a good start. However, today a control plane cannot program a set of different target switches without considerable knowledge of the internal details of each switch.

These drawbacks led us to create this proposal—a stake in the ground to stimulate debate about the next generation of OpenFlow, so-called "OpenFlow 2.0". We do *not* attempt to design a specific switch; rather, we define an *abstract forwarding model* representing how packets are pro-

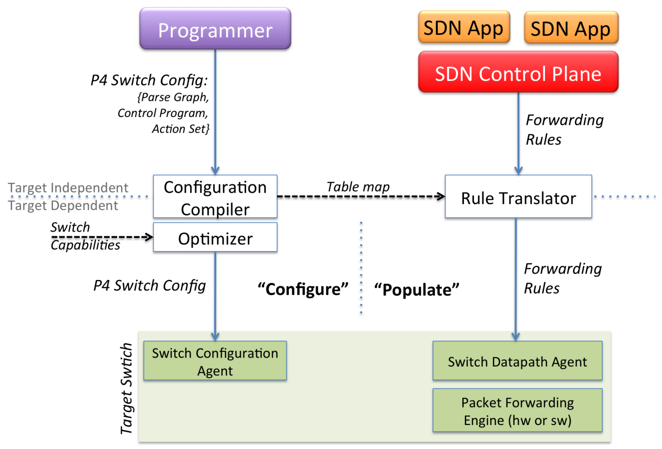

Figure 1: Overall Approach.

cessed by different switch implementations. A programmer writes a program—in a new language we call *P4*—describing how the switch is to process packets in terms of our abstract forwarding model. The P4 program is compiled down to tell the real target switch how to process packets. Rather than have each switch express its capabilities to the programmer, the compiler is assumed to know the specific details of the target switch, and is responsible for mapping the programmer's *target-independent* program down to the target. The key is to find a "sweet spot"—a language that balances the need for expressiveness with the ease of efficient implementation across a range of hardware and software switches. In our approach, the programmer writes a target-independent switch configuration that specifies a parse graph, action set, and control program written in P4, as shown in Figure 1. The compiler converts the switch configuration into a target-dependent configuration of the switch. When the control plane adds or deletes flow entries, the rule translator maps them to the correct location in the switch.

In our proposal we set out to accomplish three main goals:
Goal 1: Protocol independence. The switch should not be tied to any specific network protocols. Instead of installing a rule like "Forward packets with Ethernet destination address A to port 17", we should express the rule as an exact match on a particular sequence of bits in the header, followed by an action constructed from one or more primitives. Primitives might include: "Overwrite field A with value B", or "Insert new field X after field Y". The goal is to allow new headers and actions, while still supporting existing protocols. To enable new header formats, our abstract forwarding model includes a *programmable parser* configured by the programmer. Switches often process each header in a separate match+action stage, where stages are in series or in parallel; therefore, our abstract forwarding model consists of *multiple match+action stages*. However, some switches cannot support arbitrary matches and actions. For example, existing switch ASICs support certain match types on specific locations in the header, with pre-determined actions. Therefore, we specify a small base set of match types and action primitives that all switches should support, and allow switches to identify additional actions they support and any constraints on matching (i.e., its *capabilities*) as a switch profile or a target-specific driver in the compiler.
Goal 2: Target independence. A programmer should be able to write a program that describes how switches should process packets, and then compile the program to a variety of target switches (software or hardware) from different vendors. Just as a C programmer does not need to know the specifics of the underlying CPU, the network programmer should not need to know the details of the underlying switch. Instead, the compiler takes the switch's capabilities and limitations into account when turning a target-independent description (from the programmer, written in P4) into a target-*de*pendent program (used to configure the switch, also in P4). In some cases, the programmer may wish to give hints to the compiler or hand-optimize the compiled code to squeeze performance out of the switch. We therefore propose a single programming language for use above and below the compiler. The only difference is that the code below the compiler has been targeted with specific knowledge of the switch.
Goal 3: Reconfigurability in the field. A programmer should be able to redefine how a switch processes packets, even if the switch is already deployed. This may require describing a new parse graph to identify new headers, and rearranging the match tables because a new header matches in a table previously used for another purpose. Changing the headers might also introduce new dependencies in the processing pipeline. For example, prior to the change two headers might be processed in parallel, whereas a new header might introduce a dependency requiring them to be processed in series. Therefore, we assume a switch has two different kinds of operation: *configuring* how packets are processed, and *populating* the flow tables by adding and deleting flow entries. While the switch is in the process of being configured it may, or may not, forward packets—we expect this to vary from switch to switch.

These three goals lead us to the architecture in Figure 1. We assume that a programmer creates a *switch configuration* which tells the switch how to process packets using our P4 language. Today, the programmer may be a human writing firmware to configure the switch prior to processing packets. In the future, for more flexible switches, the switch configuration might be generated automatically by the control plane.

Related work. Several pieces of work laid the groundwork for our proposal. Most notably in 2011, Yadav et al. [5] proposed an abstract forwarding model for OpenFlow, but with less emphasis on a compiler. Kangaroo [6] introduced the notion of programmable parsing. Recently, Song [7] proposed protocol oblivious forwarding which shares our goal of protocol independence, but is targeted more towards network processors (NPUs). ONF has recently introduced typed tables to express the matching capabilities of switches [4].

2. ABSTRACT FORWARDING MODEL

Our approach assumes an abstract forwarding model for how switches forward packets, shown in Figure 2. Our model consists of a programmable parser followed by multiple stages of match+action (arranged in series or parallel, or a combination of both). The model is clearly derived from OpenFlow, with a couple of generalizations. While the original OpenFlow model assumes a fixed parser, our model assumes a programmable parser, to allow new headers to be defined. Our model assumes the match+action stages can be in parallel or in series, whereas the OpenFlow model assumes they are in series. And although not visible in the figure, our model assumes actions are built from a set of primitives supported by the switch.

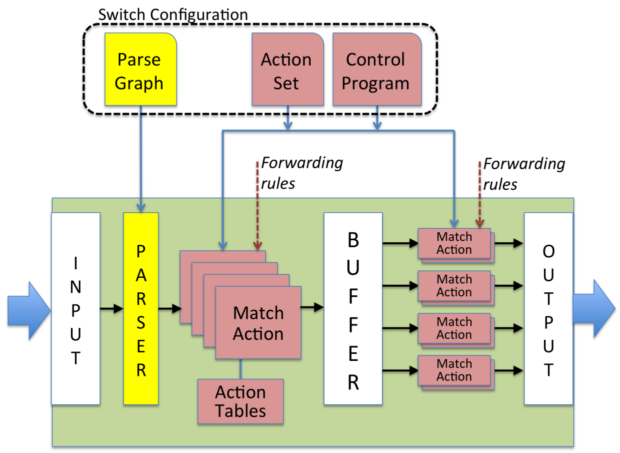

Figure 2: The abstract switch forwarding model.

The forwarding model is controlled by two types of operations. *Configure:* The parser is programmed and the match+action stages are configured to process specific header fields. Configuration determines which protocols are supported and how the switch will process packets. *Populate:* The control plane populates the match+action tables by adding and deleting forwarding rules consistent with the current configuration.

For the purposes of this paper it can help to assume the switch is configured once at boot time, then after that, the tables are populated on-the-fly as and when new forwarding entries are needed; for example, when routing tables change. However, in practice, we expect future switches will be able to forward packets during partial or full reconfiguration, to allow upgrades with no downtime. Our model deliberately allows for, and encourages, reconfiguration that does not interrupt forwarding.

Arriving packets are first handled by the parser. (The packet body is assumed to be buffered separately, and unavailable for matching). The parser recognizes and extracts fields from the header, and thus defines the protocols supported by the switch. The model makes no assumptions about the meaning of protocol headers, only that the parsed representation defines a collection of fields on which matching and actions operate.

The extracted header fields are then passed to the match+action tables. The match+action tables are divided between ingress and egress. While both may modify the packet header, ingress match+action determines the egress port(s) and determines the queue into which the packet is placed. Based on ingress processing, the packet may be forwarded, replicated (for multicast, span, or to the control plane), dropped, or trigger flow control. Egress match+action performs per-instance modifications to the packet header (e.g. for multicast copies). Action tables (counters, policers, etc.) can be associated with a flow to track frame-to-frame state.

Packets can carry additional information between stages, called *metadata*, which is treated the same as packet header fields. Some examples of metadata include the ingress port, the transmit destination and queue, and data passed from table-to-table that does not involve changing the parsed representation of the packet.

Queueing disciplines are handled in the same way as the current OpenFlow; an action maps a packet to a queue, which is configured to receive a particular service discipline. The service discipline (e.g. minimum rate, DRR) is chosen as part of the switch configuration. Choosing from a set of actions using a hash function is also handled in the same way as the current OpenFlow.

Although beyond the scope of this paper, action primitives can be added to allow the programmer to implement new or existing congestion control protocols. For example, the switch might be programmed to set the ECN bit based on novel conditions, or it might implement a proprietary congestion control mechanism using match+action tables.

3. MOTIVATING EXAMPLE

To illustrate how a programmer *configures* a switch and *populates* its tables, imagine introducing a new header field (XTAG) into a network carrying Ethernet, VLAN and IPv4 traffic. The programmer wants the switch to forward packets using XTAG under the following conditions:

Load-Balancer: If an arriving packet is destined to a public virtual IP (VIP) address and does not already have an XTAG, the switch adds an XTAG to the packet by choosing it from a set of N possible tags. The switch will generate a hash using the IP and L4 fields and this result modulo N chooses the entry in the set to use. The XTAG identifies the server the packet will be sent to, thereby spreading the load across the server destinations in the set.

Tag-Forwarding: If an arriving packet has an XTAG, or if an XTAG is added by the Load Balancer, then it is forwarded according to the XTAG table. The XTAG table might indicate that the XTAG should be removed on transmit, for example if the packet is entering the public Internet where XTAGs are not recognized.

IP Routing: If a packet does not have an XTAG, and is not destined to a public VIP, it is forwarded based on its IPv4 destination address if possible. Otherwise, it is forwarded based on MAC address, though we elide the details of MAC forwarding for the remainder of the paper.

4. PROGRAMMING THE SWITCHES

In our approach, a programmer tells the switch how to process packets by creating a *switch configuration* consisting of three parts: (1) A *parse graph* identifying the names and types of the header fields recognized (or added) by the switch. In our example, this defines where the XTAG header can appear in a packet, and its format. (2) A *control program* describing how packets are to be processed, using the language described in Section 4.3. And (3) an *action set* describing action macros in terms of primitive operations implemented by the switch. The control program is written in terms of header types defined in the parse graph, and actions to be performed on packets. We call the three parts needed to configure a switch the *P4 configuration*, which is written using the P4 language. The configuration compiler reads the P4 configuration (i.e. the parse graph, the control program, and the list of actions) and decides which tables to place in each match+action stage; for example, which match table should hold the XTAG. To *populate* the switch, the control plane adds new flow entries to the switch, for example to populate the XTAG table.

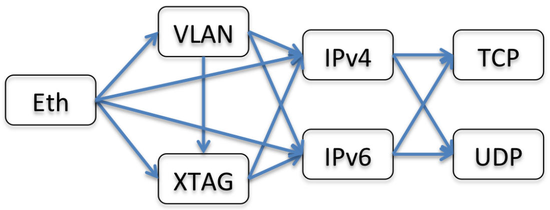

Figure 3: The parse graph for our XTAG example.

4.1 The Parse Graph

The parse graph is a directed acyclic graph that specifies the order and contents of the packet headers to be processed; Figure 3 shows the parse graph for our XTAG example. In practice, each edge is annotated (not shown) with a predicate that defines the conditions under which an edge is taken. For example, if the ethertype field in the "Eth" header indicates an XTAG is present, the parser takes the lower edge. Similarly, if the IP protocol ID indicates a TCP header is present it takes the upper branch; if it is UDP it takes the lower branch. An arriving packet that does not follow a complete path in the parse graph is considered illegal and is not processed. Each header has a type that contains a number of fields, such as *IP.dest* in fixed locations. Included in the type are functions that may determine specific fields of a header (for example an IPv4 checksum). The parse graph specifies named instances of header types, allowing fields in each instance (for example VLAN1 versus VLAN2) to be uniquely identified in the control program.

4.2 The Action Set

The control program calls for specific actions to be performed at each stage. The *action set* defines these actions from a collection of *action primitives* such as:

`Set(Header.Field, Value)`: Set a specific field in a named header. For example, `Set(IP.dest, 10)` sets the IP destination address to 10. Metadata may be set this way too.

`Set(Header.Field, Mask, Value)`: Set specific bits (defined by `Mask`) of a field.

`InsertBefore(NewHeader, Header)`: Insert ("push") a new header (and all its fields) in front of an existing header.

`InsertAfter(NewHeader, Header)`: Insert after an existing header.

`Delete(Header)`: Delete ("pop") a named header (and all its fields) from a packet.

Both the parse graph and action set use the same header types. For headers that have computed fields (such as checksums), actions may set a field as a function of other field values or metadata. Examples include `increment`, `decrement`, `IP checksum`, `IP length`, and so on. For example, the Route action used in our XTAG example updates the Ethernet header in addition to decrementing the IP TTL.

```
def decrementTTL(IP):
  Set(IP.TTL, IP.TTL-1)

def Route(NextHopDMAC, NextHopVID routerMAC):
  Set(Eth.dest, NextHopDMAC)
  Set(Eth.source, routerMAC)
  Set(VLAN.VID, NextHopVID)
  decrementTTL(IP)
```

If a switch can perform a complex action, such as calculating an IP checksum on specific fields, it is free to use that hardware implementation and associate a primitive with that function, rather than having to support arbitrary arithmetic operations on any field.

4.3 The Control Program

The control program specifies two things: a table type for each match+action table (which includes a name and a description of the allowed contents of its entries), and a description of how packets are to be processed (which specifies the order and conditions under which each table is executed).

Table Types: A programmer declares the expected contents of a table by stating its *type*. Table types include the following components:

- **Matches**: The fields that the table can read (e.g. srcip or dstport) and the match mechanism (exact, prefix, wildcard, or range). The available field names are determined by the parse description.
- **Actions**: The fields the table may set or modify, and other actions the table may perform such as decrementing a TTL field or incrementing a counter.
- **Size**: Optional estimate of the number of entries.

For example, the following table declaration states that the `TCP_Monitoring` table exact matches on the `dstport` field and may execute the `count` action.

```
table TCP_Monitoring {
    reads: { dstport [exact] }
    actions: { count }
}
```

Knowing the type of a table allows the compiler to map each table to the appropriate physical resources at configuration time. For example, the TCP_Monitoring table can leverage SRAM that can only perform an exact match,

```
//----- Monitoring -----//
table TCP_Monitoring {
 reads: {dstport [exact]}
 actions: {count}
}

def monitoring():
 if defined(TCP) then
  table(TCP_Monitoring)

//--- XTAG Forwarding ---//
table Load_Balancer_Enable {
 reads: {inport [exact]}
 actions: {set LB_EN}
}

table Load_Balancer {
 reads: {dstip [exact]}
 actions: {add XTAG}
}

table TAG_Forward {
 reads: {XTAG.tag [exact]}
 actions: {forward, del XTAG}
}

def tag_forwarding():
 if !defined(XTAG) then
  table(Load_Balancer_Enable)
  if LB_EN = 1 then
    table(Load_Balancer)
 if defined(XTAG) then
  table(TAG_Forward)
```

```
//------ Routing ------//
table Routing_Enable {
 reads: {dstmac [exact],
          vlan [exact]}
 actions: {set ROUTE_EN}
}

table Routing {
 reads: {dstip [prefix]}
 actions: {Route}
}

def routing():
 if defined(IP) &&
   (!defined(XTAG)
    || LB_EN = 0
    || !written(outport))
   then
    table(Routing_Enable)
    if ROUTE_EN = 1 then
      table(Routing)

//--- L2 Switching ---//
def switching():
 ... omitted ...


//---- The Switch ----//
def main():
 monitoring()
 tag_forwarding()
 routing()
 switching()
```

Figure 4: The XTAG control program, in P4 syntax.

whereas a table that matches on a destination IP prefix may leverage TCAM for more flexible matching. By specifying the set of actions, the table type describes the set of possible actions that must be available to the table entries at population time. The expected size determines the number of entries that should be available during population time.

When the control plane populates switch tables, new entries must obey the table type declared during configuration, otherwise the updates will be rejected. For example, the rule translator should reject an update to the monitoring table with rule that matched on a srcip in 1.2.3.0/24 because the table type only allows an exact match on a dstport, not a prefix match on the srcip. Without a strict contract between control plane and switch, the switch cannot allocate resources in advance. Of course, some switches may allow rapid reconfiguration, making it possible to reconfigure table tables to adjust to dynamically changing requirements.

Packet Processing: Packet processing is expressed as a simple imperative program. These programs are organized as collections of packet processing functions where each function uses a combination of conditional statements (if-then-else), sequencing and primitive operations to express the desired series of packet-processing operations. The most important primitive is the `table(<name>)` primitive, which specifies that packets should be processed by the named table at that point in the computation. As an example, consider the following monitoring function:

```
def monitoring():
  if defined(TCP) then
    table(TCP_Monitoring)
```

This function states that if the `TCP` header is defined then the match+action rules of the `TCP_Monitoring` table should be executed. Prior to being populated, the default action of such tables is to do nothing to a packet.

The XTAG Example: Figure 4 presents a complete control program for implementing XTAG forwarding. The program consists of five components with each component containing zero or more table declarations as well as a function to define the packet-processing computation for the component. The first four components—Monitoring, XTAG Forwarding, Routing and L2 Switching (the latter omitted for the sake of brevity)—each define a separate packet-processing task. "`main`" completes the switch by assembling the four components into one.

5. COMPILATION

To implement the P4 configurations described in the previous section, each target switch must have a compiler capable of mapping the device-independent descriptions on to their specific hardware or software platform. Doing so involves compiling parse graphs as well as mapping control programs on to the physical resources provided by the target platform.

5.1 Compiling Parse Graphs

The parse graph provided by the programmer is used to configure the switch's parser. In chips that only support fixed parse graphs, the compiler merely needs to check if the programmer is using a graph consistent with the chip's capabilities. In chips with *programmable* parsers (e.g., [8]), the compiler generates a state machine described by the parse graph. The state machine is executed using a state table built from RAM or CAM in the switch.

In practice, the programmer describes the parse graph using a state transition table, specifying the state transitions taken in response to next-header field values found in the packet header. The table contains columns for the current header type, lookup values, next header type, the current header length, and the lookup offsets for use in the next cycle. Each edge in the parse graph represents a state transition, thus each edge must be encoded as a parse table entry. The state transitions follow the path taken through the parse graph.

In previous work [9] we described a language for representing the parsing state transition table, and showed how to generate the state machine from the table. P4 adopts the same syntax for expressing parse graphs.

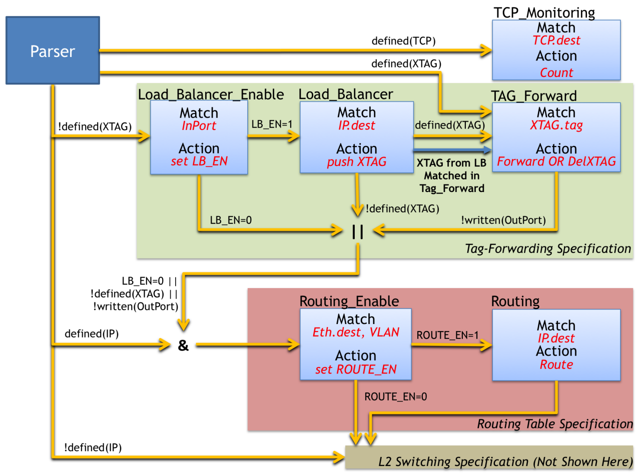

Figure 5: XTAG control-flow graph.

5.2 Compiling to Control-flow Graphs

The control programs in the previous section expressed the end-to-end forwarding behavior of a switch, but none of the concurrency inherent in such packet-processing programs. For example, while the XTAG program specifies that the `Monitoring` and `XTAG_Forwarding` components execute in sequence, they can safely be implemented in parallel, since neither table reads nor writes a header field written by the other. In general, given a set of table type declarations, which provide information on fields read and written (via action specifications), a compiler can detect if two tables (and hence two program components) can safely be executed in parallel. The first step in the control program compilation process is to perform such an analysis and to generate an equivalent *control-flow graph*.

As an example, consider the control-flow graph in Figure 5, which results from compiling the XTAG program defined earlier. In this diagram, the blue boxes represent tables together with their type information. The name of the table appears on top of the box. Each yellow edge represents a possible control-flow path from table to the next. These yellow edges are annotated with predicates that represent the conditions under which control may flow from one table to the next. When multiple yellow edges exit the same table, execution will continue along each edge simultaneously provided the predicates on each edge are true. For example, when the packet emerges from the parser, if both `defined(TCP)` and `!defined(XTAG)`, control will flow simultaneously to the `TCP_Monitoring` component and the TAG forwarding component. These two components can execute concurrently because there are no dependencies between them: they neither read nor write the same fields.

Proper execution of a particular table may depend upon completion of two or more other tables (each might set a different piece of metadata, or write a different packet field). In such cases, you will see multiple yellow edges come together at a join point (i.e., a synchronization point) in the control-flow graph. For example, the join point marked by the conjunction (`&`) indicates the `Routing` component will not execute until the `Parser` completes (and `defined(IP)`) and the Tag Forwarding component completes (and one of `LB_EN=0` or `!defined(XTAG)` or `!written(OutPort)`).

For the sake of illustration, we also added one *blue* edge to the diagram, depicting a data-flow dependency between tables (the XTAG written by the Load Balancer is used by the TAG forward table). Edges indicating data-dependencies are optional, because they may be inferred directly from the control program during the compilation process.

5.3 Compiling to Target Platforms

Finally, the control-flow graphs are compiled into a variety of hardware and software target platforms.

Programmable switches: In a programmable platform such as a software switch or an FPGA-based switch, the compiler could map logical tables directly (i.e. one-to-one) to corresponding physical tables. Despite the ability to instantiate as many physical tables as desired, table types are useful for constraining the widths, heights, and matching criterion (e.g., exact, prefix, or wildcard) of each table.

Switches with programmable pipelines [8]: For switches consisting of a pipeline of match+action stages, each with different kinds of physical tables (e.g., TCAM, SRAM), the compiler can map each logical table to multiple physical tables (one-to-many) or many logical tables to one physical table (many-to-one). SRAM could be used when the reads are exact match, and TCAM otherwise. For example, the `TCP_Monitoring` and `Load_Balancer` functions could use SRAM (on inport and XTAG respectively), but the `Routing` table would use TCAM to do prefix matching on dstip. For switches with action processing only at the end of a pipeline, the compiler can have intermediate stages generate metadata that are used to perform the final writes.

Switch with parallel physical tables: Some hardware switches may have multiple physical tables that can operate concurrently. The compiler for this switch could identify operations to safely perform in parallel. For example, using the control-flow graph of Figure 4, the compiler could determine that operations such as `TCP_Monitoring`, `Load_Balancer_Enable`, and `Routing_Enable` could be performed in parallel—because they do not write header fields read by the other—enabling a shallower pipeline.

Switches with a small number of tables: Some control programs may have more logical tables than the underlying switch has physical tables; an OpenFlow 1.0 switch may have only a single TCAM. The compiler can "compose" multiple logical tables into a single physical table using the crossproduct of individual table entries. For example, a `TCP_Monitoring` table with m rules and a `Routing` table with n rules could be combined into a single table with $n \times m$ rules. The best way to compose tables is an interesting optimization problem.

6. CONCLUSION

The promise of SDN is that a single control plane can directly control a whole network of switches. OpenFlow supports this goal by providing a single vendor-agnostic switch API. But the current OpenFlow specification is targeted towards *fixed* function switches, that recognize a pre-determined set of header fields, and process packets using a small set of predefined actions. It does not allow the control plane to express how packets *should* be processed to best meet the needs of the control plane and its applications.

We propose a step towards more flexible switches whose functionality is specified–and may be changed–in the field. We start with an abstract forwarding model that encompasses a wide variety of switch implementations, yet can be configured by a simple imperative higher level language. The programmer decides how the forwarding plane process packets without having to deal with the details of how each switch is implemented. A compiler transforms the programmer's imperative specification into a control-flow graph that can be mapped to many specific target switches, including highly optimized parallel hardware implementations.

We emphasize that this is only a first step, designed primarily as a strawman proposal for OpenFlow 2.0, and as a concrete contribution to the debate. But even in this proposal, several aspects of a switch remain undefined, such as how to configure congestion control primitives or queue service disciplines.

However, we leverage the use of powerful software techniques such as abstract languages, various intermediate forms (as in Figure 4 and Figure 5), and compilers to transform abstract intermediate forms into more concrete implementations. It is the use of these classical software techniques we believe will lead to future switches to provide with greater flexibility, and will ultimately unlock the potential of software defined networks.

Acknowledgments. This work is supported in part by ONR grant N00014-12-1-0757, NSF grants CNS 1111520 and SHF 1016937, and a Google Research Award. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the NSF, ONR, or Google.

7. REFERENCES

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communications Review*, 38(2):69–74, March 2008.

[2] Open Networking Foundation website. `http://www.opennetworking.org`.

[3] OpenFlow Switch Specification. `http://goo.gl/3VO195`, October 2013. Version 1.4.0.

[4] Openflow forwarding abstractions working group charter. `http://goo.gl/TtLtw0`, April 2013.

[5] Navindra Yadav and Dan Cohn. OpenFlow Primitive Set. `http://goo.gl/6qwbg`, July 2011.

[6] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *INFOCOM*, pages 830–838. IEEE, 2010.

[7] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *SIGCOMM HotSDN Workshop*, Aug. '13.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, '13.

[9] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Proceedings*, ANCS '13, pages 13–24, Piscataway, NJ, USA, '13. IEEE Press.