HotNets 2019 Paper

Paper #0, 3 pages

ABSTRACT

1. INTRODUCTION

Over the last decade, synergistic development of hardware and software systems based on match-action based abstraction has fundamentally changed packet processing in network devices. In one hand packet processing chips based on the domain-specific hardware (e.g., RMT[2], d-RMT [3]) provide a great flexibility to reconfigure the data plane without modifying the target device, on the other Programming Protocol-Independent Packet Processors(P4 [1], [5]) allows to program this re-configurable chips based target devices (e.g., TOFINO [12], FlexPipe etc.,). P4 allows to describe packet processing logic required in data plane of target devices to realize a network function and expose APIs to configure the data plane objects.

Packet processing in data plane is primarily divided in three phases. (1) Parsing headers, (2) Headers and data modifications using match-action processing and (3) Packet reassembly by deparsing. The re-configurable hardware based targets provide of programmable packet processing blocks(Parser, match-action tables (MATs) and deparser) to support processing of each phase. P4 comprises of multiple sub-languages and each sub-language is designed to express logic for one or more types of programmable blocks. In addition, target devices may have function specific blocks (e.g., Packet Replication Engine to create copies of packets in the device). All the packet processing blocks, programmable or fixed function, may be arranged in different permutations in different target devices creating different architecture models for packet processing. P4 allows to express architectures of target devices using a sub-language. This synergy between P4 and re-configurable hardware have greatly enhanced data plane programmability. However, the same synergy has created tight coupling between the language and hardware, thereby, exposing target device specific architecture details to programmers. Therefore, reuse of code across different target devices have become extremely difficult.

Even for the target devices with the same architecture and set of processing blocks, P4 mandates to write a monolithic data plane program and carefully configure the data plane objects to build an application processing various protocols and performing multiple network functions. For example, `switch.p4` [4] has P4 code written to processes different protocol headers and network functions(e.g., l2 switching, l3 routing etc.,). But, the code globally share different types of metadata structures and parsed headers. To program a target device only for ethernet switching using switch.p4, it is required to understand complete program and macros used in it to disable extraneous code. Without understanding implementation details of the program with multiple features, it is difficult to reuse fine-grained fragments of code from it. The difficulty arises due to absence of language constructs that can allow to define fine-grained packet processing modules with interfaces to communicate with them. Existing data plane programming ecosystem needs modularity that allow programmers to expose interface to reuse code written to process packet at any granularity while abstracting away the implementation details.

Previous work, HyPer4 [11], HyperV [13] use virtualization to support modularity. P4Visor [14] supports testing specific composition operators(A-B and Differential) by merging P4 programs using compiler techniques. In both approaches, minimal unit of re-usable code is an entire data plane program of a network function. These approaches may allow to reuse of network functions but can not allow to reuse of fine-grained packet processing code different network functions and create n

For example, where a module could be processing packets for only set of protocols at a specific layer instead all the

Hence, it does not allow to reuse code at required granularity.

incrementally develop and enrich a network function by reusing code of already developed and tested modules. Also, these approaches lack mechanisms to facilitate inter-module communication and to define interface For example, e.g., next hop in above example).

Encapsulating customized headers inside the packet may allow such communication, but that would require to know implementation details of deparser in one module to write complimentary parser in other and vice versa.

parallel processing -copy semantics

In this paper, we present, a Micro Switch Architecture(μSA) and a compiler, μP4C, for a logical target to build network functions by reusing fine-grained packet processing code. μSA allows define interface to expose code modules as callable

P4 packages. P4 programmers can reuse of the code by invoking the packages interfaces without knowing implementation details of the code. Using μP4C, programmers can compile real target specific executable by linking all the μSA based programs, composing them into a single program.

- Compiler Midend to link all the programmable blocks compose them as dictated by their call location in execution-control of the source program.
- and transform complete CSA specific P4 program to any target architecture (e.g., v1model of BMV2 or PSA)

«Paper outline para » rest of the paper is arranged....

2. USE CASE

Let's consider a simple scenario as shown in Figure 1. A program, l3.p4, parses IPv4 header from packets, performs longest-prefix match and determines next hop. Moreover, it decrements the ttl field and deparse the packet. Another program, l2.p4, processes the same packet and takes the next hop as input argument, parses Ethernet header, matches on id of next hop and modifies ethernet addresses. Finally, it deparses the packet and sends on appropriate port. In this example, l3.p4 is not generating a functionally correct packet to forward on wire. However, it can be reused with different layer-2 forwarding mechanism or even with MPLS and create functionally correct packet to forward on wire. Similarly, l2.p4 can be reused with IPv6 based routing. Such fine-grained packet processing modules enable code reuse and modular control over data plane objects, thereby facilitating incremental development of network functions.

3. CHALLENGES

P4 is developed to specify data plane functionality of software or hardware targets (e.g., simple_switch [6], TOFINO [12] etc.,) devices. Target device manufacturer provides an architecture description of data plane along with a P4 compiler for the device. The architecture source file (e.g., v1model.p4 [9] for simple_switch) contains declaration of a set of programmable blocks, their data plane interfaces and the target specific metadata (called intrinsic metadata) and extern functions. To program the packet processing pipeline of a device, P4 programmers provide implementation of the declared programmable blocks in its architecture file. Therefore, P4 programs are tightly coupled with architecture of the target devices.

The architecture description of a target specifies data plane pipeline comprising a set of programmable and fixed-function blocks, flow of user-defined and target-specific (called intrinsic) metadata in the pipeline and semantics of target-specific actions and externs. For example, Figure 2a and 2b show data plane pipelines and their packet processing blocks defined in v1model [6] and Portable Switch Architecture (PSA) [8], respectively.

Parser blocks are described using a sub-language of P4, based on abstraction of Finite State Machine. Control blocks are expressed using a sub-language modelling imperative control-flow. Deparsers are special control blocks that allows to use statements, to reassemble packets, that are prohibited in other control blocks. Packet processing logic in P4 programs are sectioned across programmable blocks with heterogeneous abstract machines (e.g., parsers, deparsers, control) and fixed-function blocks (Packet Buffer and Replication Engine). Therefore, a control block can not instantiate and execute parser blocks. Similarly, deparser block can not invoke and execute parser blocks. Consider the example in Figure 1, parser "P" of l2.p4 can not be executed at the end of deparser control block "D" of l3.p4.

Moreover, architectures models expose intrinsic metadata, target-specific actions and extern functions (e.g., resubmit, recirculate, clone) to replicate and/or program packet-path and flow of data across the processing blocks, as described in [6] and [8] In turn, adding different abstract-machine to program packet-path and data flow across the blocks and further increasing heterogeneity in the model of a data plane program. «For example, resubmit or recirculate calls are like event trigger.. effects happen after the block»

Finally, the existing architecture models expose target-specific constraints. E.g., output port can not be changed in egress control block, scope of some intrinsic metadata bounded by particular programmable blocks, etc., Programmers need implement execution logic conforming to constraints, architecture model and semantics of actions and extern functions of the target device.

Due to absence of uniform abstract machine for a program and presence of target-specific constraints, composition of data plane program modules is extremely difficult, even for the same target. Also, existing compilers and architecture specifications do not provide simplified and common abstractions for packet processing blocks in data plane to facilitate target agnostic reuse of data plane programs.

First, we simplify abstract machine for P4 data plane programs by reducing code fragmentation and number of programmable blocks. Second, we abstract out fixed-function blocks by deriving logical externs. Third, we develop compiler mechanisms to have uniform abstract-machine for programmable blocks. Finally, we translate our unified abstract machine and logical externs into real target-specific heterogeneous packet-processing blocks, features and constraints.

In section 5, we explain the design of Micro Switch Architecture for a logical target that reduces number of programmable blocks and introduces logical externs to minimize heterogeneity in abstract model of P4 programs.

4. MICROS P4

overview:

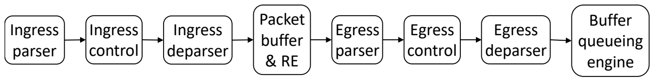
Define types of composition: Sequential Parallel

What do we need to provide interface to code modules?

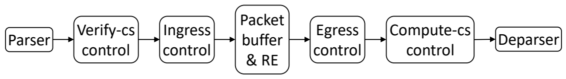
We need runtime behaviour with package. packet, sm, es, inargs, inout args -> package -> packet, sm, es, out args, inout args P4 extended to allow Define new Package types

```
// l3.p4
parser P(packet_in pin, out hdr_t hdrs) {
  state start {
    pin.extract(hdrs.eth);
    transition select(hdrs.eth.ethType){
       0x0800: parse_ipv4;
    }
  }
  state parse_ipv4 {
      pin.extract(hdrs.ipv4);
      transition accept;
  }
}
control Pipe(inout hdr_t hdrs, out bit<16>
    nexthop_id, inout sm_t sm) {
  action process(bit<16> nh) {
    hdrs.ipv4.ttl = hdrs.ipv4 - 1;
    nexthop_id = nh;// setting out param
  }
  table ipv4_lpm_tbl {
    key = { hdrs.ipv4.dstAddr : lpm }
    actions = { process; }
  }
  apply {
    ipv4_lpm_tbl.apply();
  }
}
control D(packet_out po, in hdr_t hdrs) {
  apply() {
    po.emit(hdrs.eth);
    po.emit(hdrs.ipv4);
  }
}
// l2.p4
parser P(packet_in pin, out hdr_t hdrs) {
  state start {
    pin.extract(hdrs.eth);
  }
}
control Pipe(inout hdr_t hdrs, in bit<16>
    nexthop_id, inout sm_t sm) {
  action forward(bit<48> dest_mac, bit<48> src_mac
      , bit<8> out_port) {
    hdrs.eth.dstAddr = dest_mac;
    hdrs.eth.srcAddr = src_mac;
    sm.out_port = out_port;
  }
  table forward_tbl {
    key = { nexthop_id : exact }
    actions = { process; }
  }
  apply {
    forward_tbl.apply();
  }
}
control D(packet_out po, in hdr_t hdrs) {
  apply() {
    po.emit(hdrs.eth);
  }
}
```

Figure 1: Fine-grained packet processing modules - l3.p4 and l2.p4



(a) PSA Model



(b) simple_switch v1 Model

Figure 2: Real Target Architectures

runtime behavior with package types

MicroP4 defines package interfaces.

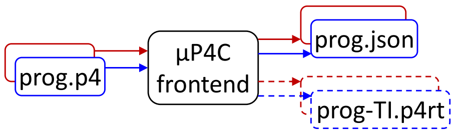
MicroP4 captures intrinsic metadata dependency of real targets using logical extern (egress_spec)

MicroP4 defines common abstractions for target specific operations that can not be expressed in P4 e.g., multicast

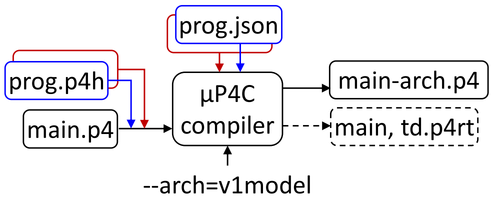
Micro P4 higher-level description and usage:

Architecture model of real targets provides a default path, comprising of processing blocks, for packet and data in data plane pipeline. Programmers can use target-specific externs to change the default path and route the packet through required processing block. Intuitively, we can say that the packet-processing code is stationary whereas packet and data are propagated through the stationary code blocks in pipeline. P4 compiler backends of real targets mandates programmers to map logic and provide code for different processing units in data plane instead of automatically allocating code to the units.

Micro-Switch Architecture, μSA, is designed with completely opposite philosophy. μSA provides abstraction to avail features of maximum processing blocks at any stage of execution-control of a program and μP4C compiler transforms code according to the abstractions to allocate on stationary processing blocks of the real target device.



(a) Library Program Compilation



(b) Composition and Translation of Main Instance

Figure 3: Compiling μSA programs

5. MICROS-SWITCH ARCHITECTURE

μSA, is designed for a logical target, unlike fixed pipeline of real target data planes with stationary processing blocks. μSA provides multiple types of logical pipelines. For each pipeline type, it exposes an interface type comprising a set of programmable blocks. Programmers can define a P4 `package` type by implementing all the programmable blocks of a interface type. In the same source file, programmers can provide multiple implementations of the same interface type to define multiple package types. μSA pipelines do not have fixed-function blocks, instead it provides a set of logical externs which can be instantiated and used within control blocks of the μSA pipelines. This design choice reduces heterogeneity in abstract model of data plane programs.

μSA defines various architecture specific structures and externs that allow programmers to express sequential and parallel composition of fine-grained packet processing functions. It defines standard intrinsic metadata as `msa_sm_t` as a struct type. The fields of this struct provide basic information populated by the target e.g., `packet_length`. Some fields are not mutable, however μSA allows to declare instances of the struct and perform assignment operation between two instances to create copies. Sections 5.1 and 5.2 describe pipelines and logical externs, respectively.

5.1 Pipelines

μSA has two types pipeline, Micro and Orchestration. Figure 4 shows programmable blocks of two types. Micro pipeline interface comprises of parser, Micro-Pipe control and Deparser programmable blocks. Every incoming packet is parsed and validated by the parser, if parser terminates in `accept` parse state, then execution-control is transferred to micro-pipe control block. Depending on use of logical externs in implementation, packet may not complete the processing of the control block. If the execution-control reaches till the end of the control block, packet is processed by the deparser block.

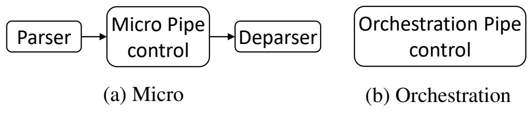


Figure 4: μSA Pipelines

Orchestration pipeline interface has only a control block, orchestration pipe. Programmers can implement imperative control flow using intrinsic metadata of μSA and runtime parameters to the package types. In addition of existing language features, it allows programmers to instantiate package types and invoke them in the body(`apply`) of the control block.

5.2 Logical Externs

TODO rename "function" to "mehod".

5.2.1 Packet Externs

Packets are represented using `packet_in` and `packet_out` externs defined in P4-16 core library [7]. However, these externs do not provide functions and semantics required to describe sequential and parallel composition of programs. We define μSA specific packet externs to allow programmers to express sequential and parallel processing. μSA specific packet externs `msa_packet_in` and `msa_packet_out` contain an instance of `packet_in` and `packet_out`, respectively. We introduce a function, `get_packet_in`, in `msa_packet_out`.

```
void get_packet_in(msa_packet_in msa_pin);
```

Similarly, we introduce copy_from function in `msa_packet_in` that replicates the state of passed argument (`msa_pin`) to the calling instance.

```
void copy_from(msa_packet_in msa_pin);
```

5.2.2 Egress Specifications

To support special features, the architectures of real target devices provide intrinsic metadata that depends on operations on other intrinsic metadata. For example, many real targets allow to measure every packet's queuing latency through the device. The queuing latency is computed by recording enqueue and dequeue timestamps in the device. These timestamps can be measured only after the packet's egress port is finalized and the packet is sent to the packet buffer. The architectures of real target devices expose timestamps as intrinsic metadata which can be accessed only after the egress port is finalized. μSA considers `egress_spec` as a stateful extern object (Figure 5) providing functions to set and get `egress_port` and retrieve values of other egress_port dependent intrinsic metadata. μP4C allows repeated usage of the extern's functions in the single control block of μSA pipelines. μP4C uses their occurrences to transform the code to multi-control pipelines of real target architectures. If `get_value` occurs before `set_egress_port` on any possible execution-control path, μP4C raises a compile-time error.

```
enum msa_metadata_t {
  INGRESS_TIMESTAMP,
  EGRESS_TIMESTAMP
}
extern egress_spec {
  void set_egress_port(in bit<8> port);
  bit<8> get_egress_port();
  bit<32> get_value(in msa_metadata_t ft);
  void copy_from(egress_spec es);
}
```

Figure 5: Egress_Spec Extern

5.2.3 Packet Buffer

μP4 allows programmer to express parallel processing of multiple P4 programs by passing the copy of the packet and intrinsic metadata to multiple programs. Programmers can use assignment statements to create copies of standard intrinsic metadata (msa_sm_t) and use `copy_from` functions in `msa_packet_in` and `egress_spec` to create copies of the packet and intrinsic metadata. In addition, it provides logical a packet buffer as an extern (Figure 6) to serialize output packet and intrinsic metadata. The logical buffer extern can only be instantiated in the control block of orchestration pipeline. Its functions can be used only in apply body of the control block. The output of the multiple programs generated by processing the same packet can be enqueued in the logical buffer. The dequeue function allows to fetch the packets and either pass it to another program to process or assign it to the msa_packet_out argument of orchestration pipe control block.

```
extern msa_packet_buffer<ET> {
  msa_packet_buffer();
  void enqueue(msa_packet_out po, in msa_sm_t sm,
      egress_spec es, in ET data);
  void dequeue(msa_packet_in pin, out msa_sm_t sm,
       egress_spec es, out ET data);
}
```

Figure 6: Packet Buffer Extern

5.2.4 *Multicast Extern*

μSA's multicast extern (Figure 7) can be instantiated in the control blocks of its pipelines. The `set_multicast_group` can be used to assign replication group to the packet. The `apply` function is allowed to use only in `apply` body control blocks. It is analogous to `fork` system call in C except that processing of original packet terminates at the apply call. It returns instance id of the replicated packets and populates the egress_spec instance with the port id set by the control plane. μP4C translates this extern to multicast mechanism defined in architectures of real targets. We assume these architectures would have sufficient fields to program their replication engine, so that a combination of the fields can be used as packet instance id.

```
extern msa_multicast_engine {
  msa_multicast_engine();
  void set_multicast_group(GroupId_t gid);
  PacketInstanceId_t apply(egress_spec es);
}
```

Figure 7: Multicast Extern

6. μP4C COMPILER

6.1 Parser Block Transformation

P4 parser blocks describe parse graphs as state machines. Real target devices contain programmable parser module that can be programmed using parse graphs. From the design of programmable parser [10], we make following observations. Programmable parsers are implemented using buffer, state machine logic, Ternary Content-Addressable Memories (TCAM) and Action RAM. TCAM matches values of current state, fields or variables to identify next state. Based on match, headers are copied from bit stream and current state is modified. Programmable parsers essentially perform repeated match and action. Also, we note that network packets are of finite length, hence they can be parsed in a finite number of ways. Successful parsing of a packet is essentially finding a match for a finite number of bytes from finite set of values. However, we need to extract all the bytes that might be required to perform match-actions from packets' bit streams.

We compute number of bytes required to extract in section 6.1.1, followed by an approach to convert parser without loops and variable length headers, called *simple parsers*, into a series of match-action tables. Next, we explain loops unrolling and variable length headers removal techniques transforming every parser into a simple parser. Finally, we discuss an optimization to reduce number of MATs to one.

6.1.1 *Computing Size of Byte Array*

Every `extract` method call statement in parser states advances bit index by number of bits equal to the size of the header type of the instance passed as the argument. We perform symbolic execution of parser, control and deparser blocks to determine size of byte array buffer by evaluating calls of `extract` method of core library extern `packet_in`, `setValid` and `setInvalid` of P4 header types and `emit` method of another core library extern `packet_out`. Symbolic execution of a parser block enumerates every possible path from the start to the accept state in the parse graph and computes total number of bytes extracted on every path. We define length $l_p(x)$ of a path x from the start to the accept state in the parse graph of program p's parser as total number of bytes extracted. And, the input buffer size for program p's parser as $\mathcal{I}(p) = \max_x(l_p(x))$.

Programs may increase or decrease size of packets, therefore, considering only parser buffer size is not enough. We analyse control and deparser blocks to determine maximum increase and decrease in packet size by the program. Initially, we set the validity bit of all the headers instances that could be extracted by the parser. We perform symbolic execution of control blocks to evaluate validity of each header instance on every path of the control flow graph. Any header instance not emitted in all the paths of deparser block is considered invalid, because such header instances will not increase size of the packet. We compute maximum and minimum number of bytes that can be emitted by program p and denote them as $\mathcal{O}_m(p)$ and $\mathcal{O}_n(p)$, respectively. We define maximum decrease and increase in packet size by program p as $\delta(p)$ and $\Delta(p)$, as shown in (1) and (2).

$$\delta(p) = \begin{cases} \mathcal{I}(p) - \mathcal{O}_n(p), & \text{if } \mathcal{I}(p) > \mathcal{O}_n(p), \\ 0, & \text{else} \end{cases} \tag{1}$$

$$\Delta(p) = \begin{cases} \mathcal{O}_m(p) - \mathcal{I}(p), & \text{if } \mathcal{O}_m(p) > \mathcal{I}(p), \\ 0, & \text{else} \end{cases} \tag{2}$$

To process packets by a sequence of N programs, we define extract length, $\mathcal{E}l_S$ and buffer length $\mathcal{B}l_S$, as shown in (3) and (4). $\mathcal{E}l_S$ denotes the maximum number of bytes that may be extracted as cumulative effect of deparsers and parsers of all the program in a sequence. $\mathcal{B}l_S$ denotes the maximum number of bytes that may be emitted as cumulative effect of the deparsers of all the programs in a sequence. Similarly, we define extract length, $\mathcal{E}l_P$ and buffer length $\mathcal{B}l_P$, as shown in (5) and (6), to process packets by N programs in parallel.

$$\mathcal{E}l_S = \max_i \left\{ \left(\sum_{j=0}^{j<i} \delta(j) \right) + \mathcal{I}(i) \right\}, \qquad i \in [0, N] \quad (3)$$

$$\mathcal{B}l_S = \left(\sum_{i=0}^{N} \Delta(i) \right) + \mathcal{E}l_S \quad (4)$$

$$\mathcal{E}l_P = \max_i \{\mathcal{I}(i)\}, \qquad i \in [0, N] \quad (5)$$

$$\mathcal{B}l_P = \max_i \{\mathcal{I}(i) + \Delta(i)\}, \qquad i \in [0, N] \quad (6)$$

6.1.2 Simple Parser to MATs

Every path enumerated by symbolic execution of a parser consists of evaluated instances of the parser states. A parser state can be a part of multiple paths, thereby having multiple instances. «>Diagram> P4 parser states consist of `parserStatements` and `transitionStatement`. The select expression in transition statement could be a header field, metadata or local variable declared in the parser. The value of select expression of a state may depend on its ancestors' parser statements. «as shown in diagram» Therefore, we perform Forward Substitution on select expressions in evaluated instances of states on each path and eliminate such data dependency.

We synthesise local binary variables, called $visit$, for each parser state to track the state transition of the parser's FSM. For every evaluated instance of a parser state, we synthesise an action comprising its `parserStatements` and replace extract method call statements to assignment statements. The assignment statements copies bytes from the buffer array to header instances' fields according to their sizes. Next, we add pop method call with the header size as the argument to remove the header from the byte array. We insert `setValid` method call statement for the extracted header instances. We add an assignment statement to set $visit$ variable associated with the parser state.

For every parser state, we create a match key comprising key-fields from sets of keys. (1) $visit$ variable assocociated with the set of all its ancestors and the state itself and (2) union of select expression of all of its evaluated instances. In most cases, the union of select expression would have a single key-field, unless forward substitution has produced different expressions in the state's evaluated instances.

The `start` being a special state has only one evaluated instance. We create an action, called `start_action`, from the only instance of the start state. Next, we visit parser states in topological order. We create match key for the current parser state as described above. `keysetExpression` and all possible paths represented $visit$ bit vector match-key entries are synthesised. For each match key entry, we use appropriate action synthesised by next state's evaluated instance.

The apply body of the control block consists of an action call invalidating all the header instances, followed by the `start_action` call and apply calls to the MATs created for parser states in topological order.

6.1.3 Variable Length Headers and Loops Elimination

The two-argument extract method is replaced with a synthesised sub-parser with three arguments. First to arguments are the same as two-argument extract method and the third arguments is maximum size of variable field. The start state of the sub-parser contains a `transitionStatement` with a `selectExpression`. The sub-parser contains a state for each possible value of variable field size that extracts constant number of bits using single argument extract method and transits to accept state. The `selectExpression` of the start state uses the variable field size parameter(the second parameter) to match on and transit to the corresponding next state.

How to unroll loops in parser? Solution: First, we need to find loops. If there is a loop, all(or at least one?) the extract stmts in the loop must(should ?) on header stack. yet to find a proper general solution.

6.2 Deparser Block Transformation

Recall that deparser blocks are specialized control blocks having `packet_out` extern as one of the arguments. The extern's `emit` method inserts bits on packets' bit-stream, if the header instance provided in the argument is valid, else no operation is performed. We precisely perform the same operation on byte array buffer, but first we create a match-action table to push fixed number of bytes on the byte array buffer. Recall that the program's parser transformation pops the total number of extracted bytes from the byte array for each packet.

The key of the table are created using valid bit of each header instance. The table contains entries enumerating all possible combination of validity of the headers and action corresponding to each match key value to push the number of bytes equals to sum of lengths of all valid headers.

The deparser block may contain P4 language constructs like `if-else` and `switch` statements in addition of `emit` method call statements. Therefore, deparser block may have multiple execution paths emitting the same header instances. We derive a directed-acyclic-graph, `emit graph`, with each node representing a emit call from the control flow graph of the deparser control block. We synthesise a match-action table for every `emit` method call in deparser blocks. Similar to one bit $visit$ variable for parser states, we create one bit variable, *emit*, for each node in the emit graph. The match key for the match-action table for an emit node comprises valid bit of the header instances passed as argument to its ancestors. In addition, the match key includes *emit* variables associated with the ancestors. The idea behind introducing *emit* variable is to record execution of previous emit calls in the control path and copy header instance of emit call at appropriate location in the byte array buffer.

6.3 (De)parser Control Block Optimization

The transformation of parser and deparser blocks into match-action control blocks induces huge cost in terms of number of binary variables in data plane and multiple match-action tables. Also, number of entries in the tables are of exponential order of binary variable fields in the tables. In this section, we describe optimization methods to reduce per state match-action tables in transformed parser blocks into a single match-action table and eliminate all the binary variables.

6.4 Cross-Architecture Code Translation

Acknowledgments

7. REFERENCES

[1] 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 Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[3] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

[4] P. L. Consortium. The switch.p4 program describes a data plane of an l2/l3 switch, 2013.

[5] P. L. Consortium. P416 language specification, 2018.

[6] P. L. Consortium. The bmv2 simple switch target, 2019.

[7] P. L. Consortium. core.p4 - p4-16 core library, 2019.

[8] P. L. Consortium. Portable switch architecture, 2019.

[9] P. L. Consortium. v1model.p4 - architecture for simple_switch, 2019.

[10] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24, Oct 2013.

[11] 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*, CoNEXT '16, pages 35–49, New York, NY, USA, 2016. ACM.

[12] B. Networks. Tofino, 2019.

[13] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, July 2017.

[14] P. Zheng, T. Benson, and C. Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 98–111, New York, NY, USA, 2018. ACM.