HyPer4: Using P4 to Virtualize the Programmable Data Plane

David Hancock
University of Utah
School of Computing
Salt Lake City, Utah, USA
dhancock@cs.utah.edu

Jacobus van der Merwe
University of Utah
School of Computing
Salt Lake City, Utah, USA
kobus@cs.utah.edu

ABSTRACT

Through virtualization, single physical data planes can logically support multiple networking contexts. We propose HyPer4 as a portable virtualization solution. HyPer4 provides a general purpose program, written in the P4 dataplane programming language, that may be dynamically configured to adopt behavior that is functionally equivalent to other P4 programs. HyPer4 extends, through software, the following features to diverse P4-capable devices: the ability to logically store multiple programs and either run them in parallel (network slicing) or as hot-swappable snapshots; and virtual networking between programs (supporting program composition or multi-tenant service interaction). HyPer4 permits modifying the set of programs, as well as the virtual network connecting them, at runtime, without disrupting currently active programs. We show that realistic ASICs-based hardware would be capable of running HyPer4 today.

CCS Concepts

•**Networks** → **Programmable networks;** *Network manageability;*

Keywords

Software Defined Networking; P4; network virtualization

1. INTRODUCTION

Recent advances in Software Defined Networking have paved the way for *dataplane programmability*, which results in network devices, including hardware, that may be reprogrammed in the field to parse custom protocols and execute custom functionality. The full potential of dataplane programmability, however, (1) remains untapped, and (2) efforts to exploit this feature are at risk of fragmentation among a proliferating collection of software and hardware devices that provide programmable dataplanes. To the first point, we believe virtualization opens up attractive possibilities, and to the second point, an ideal virtualization solution should be portable across many platforms.

OpenFlow[35] has provided a standard for programmability of the network *control plane* and has been instrumental to network operators seeking more freedom and flexibility. It leaves the network *data plane*, however, "fixed" in the sense that operators are still restricted to working with those protocols identified in the OpenFlow specification. In response to the demand for OpenFlow support for novel protocols, the specification has continually expanded [11]. A truly *programmable data plane* is free from such restrictions, permitting operators to reconfigure the data plane according to *completely custom* protocol syntax and semantics. Recent work on a reconfigurable match table (RMT) architecture proved that a programmable data plane is possible even in ASICs hardware [12]. This result inspires tools such as the P4 domain specific language [11] to make this programmable data plane easily accessible in a uniform way on a variety of switches, from ASICs with RMT, to FPGA-based switches, to software switches like PISCES [40], behavioral models [5], and the network data planes on edge servers [21].

Ordinarily, each programmable data plane provided by a P4-capable device represents a single networking context, within which one P4 program defines (1) the collection of protocol headers and the corresponding state machine by which all traffic is *parsed*; and (2) matches and actions by which all traffic is *processed*. In order to support diverse sets of customers or to flexibly compose virtual functions together for complex packet processing, in many cases operators might desire more than one context for a given network device even if this

CoNEXT '16, December 12–15, 2016, Irvine, CA, USA
 ISBN 978-1-4503-4292-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999607

device has only one physical data plane. One answer is virtualization. This technique permits the illusion of multiple programmable data planes and greater flexibility as to how they are used. For example, with data plane virtualization we gain the ability to deploy multiple network functions to the device simultaneously in various configurations, permitting:

- network *slicing*, to isolate sets of customers and/or equipment (e.g., legacy vs. modern, or insecure vs. secure), in which each slice (i.e., isolated networking context) may be unique with respect to supported protocol types and functionality;
- network *snapshotting*, storing multiple network or device configurations while a single configuration is active at a time, and permitting quick transitions between configurations;
- virtual networking *within* the device, providing a solution for both:
  - complex compositions of packet processing programs, allowing modular development; and
  - supporting multiple tenants on the switch that may wish to provide services to each other in a controlled way;
- providing standard high level features such as program profiling, traffic monitoring, and other functions.

Such an environment calls for isolation mechanisms (§4.5) to prevent one program from posing security threats to other programs, to protect the device itself from potentially dangerous programs, and to prevent one program from consuming more than a fair share of resources.

We are focused on the feasibility of virtualization implemented as a user-level program so that the result may work on top of a diverse mix of underlying system stacks. That is, we are intrigued by the possibility that a specially designed P4 program itself may provide the benefits of programmable data plane virtualization by *emulating other P4 programs*. Of the many ways to pursue programmable data plane virtualization, our approach is highly *dynamic*, such that the features listed above may be employed and reconfigured without interrupting the operation of the network device. Our approach is also highly *portable*.

One concern is that virtualization in the application layer comes at a cost. The cost, however, may be amortized over many programs (i.e., functions) sharing the same physical substrate. A second concern is that our prototype is only portable to the extent that sufficient hardware resources are available. Even for the most constrained high performance ASICs-based hardware, however, our analysis (§6) shows that a simple configuration of HyPer4 could be run in a modified version of RMT [12] today.

In the remainder of this paper, first we provide some background (§2) on the rise of data plane programmability. We then make the following contributions:

- **Context:** We provide the operational concept for HyPer4 and present some example use cases for a virtualized programmable data plane (§3).
- **Design:** We explain the design that allows HyPer4 to emulate various aspects of P4 programs and enforce program isolation (§4).
- **Implementation:** We describe the implementation of our prototype which is deployable on an unmodified software P4 switch (§5).
- **Evaluation:** We illuminate the performance and memory costs of HyPer4 and justify our claim that it could be deployed on RMT-like ASICs hardware (§6).

2. BACKGROUND

In this section we provide background material on data plane programmability and P4.

2.1 Data Plane Programmability

The concept of network progammability has a long history [22]. It begins with *active networking* [47] [9] [42] research, and continues with the separation of the control and data planes [49] [39] [16]. The term Software Defined Networking is commonly understood to describe an architecture in which the control plane and data plane are separate, and furthermore, that the control plane is *centralized*, exerting control over a *distributed* data plane [22]. The most well known example is the widely adopted OpenFlow, which provides an open API and thereby defines a standard for *control plane* programmability [30].

The OpenFlow model allows operators to write applications that run on a centralized controller, interacting with devices throughout the network to insert, modify, or remove data plane table entries in response to significant events. The community was energized by this ability to define new network functionality as software running in the controller. Previously operators had to choose between high performing but difficult to modify hardware devices, or low performing but easy to modify software devices. OpenFlow-capable hardware allowed high performance and easily modifiable functionality.

The data plane itself, however, is relatively fixed in this model, by which we mean the supported protocol headers are explicitly identified by each version of the OpenFlow specification [11]. Demand for extending the flexibility of OpenFlow architecture to an ever larger set of protocol header types has led to successive expansions of the OpenFlow specification. Clearly, this approach cannot easily respond to protocol innovation. Attention has therefore turned toward extending programmability to the *data plane*.

The presentation of the Reconfigurable Match Table architecture [12] proved that a constrained but powerful form of data plane programmability could be achieved in the fastest ASICs-based network hardware.

2.2 The P4 language

P4, a language for Programming Protocol Independent Packet Processors, is a recent innovation providing

an abstract model suitable for programming the network data plane [11]. A P4-enabled device is *protocol independent* in the sense that no inherent support for any protocols is assumed. Rather, programs written in P4 define packet headers and specify packet parsing and processing behaviors.

P4 programs include these elements: (i) **header definitions** that specify the field names and widths for protocol headers on which the program is intended to operate; (ii) **metadata**, providing packet-specific state; (iii) **registers, meters, and counters**, for state independent of packets; (iv) packet **parser** specification; (v) **match-action table** specification, identifying the packet and metatdata fields to be read and the possible actions to execute in response; (vi) **actions** - functions that may be parameterized and that invoke one or more primitives; (vii) **control flow** indicating the table execution sequence, with support for conditional branching.

Figure 1 depicts the basic operation of a P4-enabled environment. Conceptually, a P4-capable device starts with a clean-slate as in Figure 1(b). Operators *configure* the device with operational functionality by first compiling P4 code (e.g., `foo.p4` in the figure) to produce the binary or equivalent artifact suitable for loading on a P4 target. Once configured with the `foo` "persona", the tables in the switch can be *populated* via a `foo` specific controller at runtime as in Figure 1(c). Loading a different program changes the persona of the P4 target and thereby changes the device functionality itself. Such *reconfigurability in the field* is one of the goals of the P4 environment.

Figure 1(a) depicts the compilation process as typically involving two steps: first, a front-end component (such as the open source p4-hlir [7]) parses the p4 source code to produce a high-level intermediate representation (HLIR). A compiler back-end converts the HLIR into a target-specific form (e.g., binary or JSON). Examples of current P4 compiler back-end offerings include p4c-bmv2 [8] from Barefoot Networks for the software-based "simple switch" [6] or other targets using the behavioral model framework [5], SDNet [14] for Xilinx FGPAs, a P4 back end for Netronome's Network Flow C Compiler and NFP product line [34], and two efforts LLVM_P4 [21] and P4-to-EBPF [15] that compile to EBPF programs for the Linux network plane.

3. EXAMPLES

This section demonstrates some of the use cases for virtualizing the programmable data plane to provide context for the design of HyPer4 presented in §4, with a brief explanation of HyPer4's operational concept first.

3.1 HyPer4 Operational Concept

First, we explain the basic HyPer4 concept for virtualizing the programmable data plane, which involves a P4 program that emulates other P4 programs. This special program is itself a new P4 target to add to the set depicted in Figure 1(a). Figure 2 illustrates the process for deploying and working with foo.p4 within a HyPer4 environment. As shown in Figure 2(a), we configure the P4 target with the HyPer4 persona.

This persona program is itself configurable, endowed with the ability to perform many distinct behaviors, but made to execute specific behaviors through the entries in its tables. This program now becomes the target for configuration by foo.p4. In Figure 2(b), the HyPer4 compiler produces a set of commands for populating a set of HyPer4 tables.

These table entries direct the persona to carry out the behavior expressed in foo.p4. When the controller needs to perform the equivalent of populating *foo's* tables, in order to avoid modifying existing controllers we can send the commands through a Data Plane Management Unit (DPMU) as in Figure 2(c). The name of the DPMU is inspired by the Memory Management Unit in common computer architecture that translates virtual memory addresses to physical addresses and enforces program isolation. Similarly, the DPMU translates virtual table operations (intended for foo.p4) into HyPer4 table operations.

We have written some simple network functions in P4 and converted them into the table population commands necessary to make HyPer4 emulate them:

1. A layer 2 ethernet switch;
2. An IPv4 router;
3. An ARP proxy that responds to ARP requests on behalf of the IPv4 hosts for which the requests are intended; and
4. A firewall that can filter traffic based on IPv4, TCP, and UDP sources and destinations.

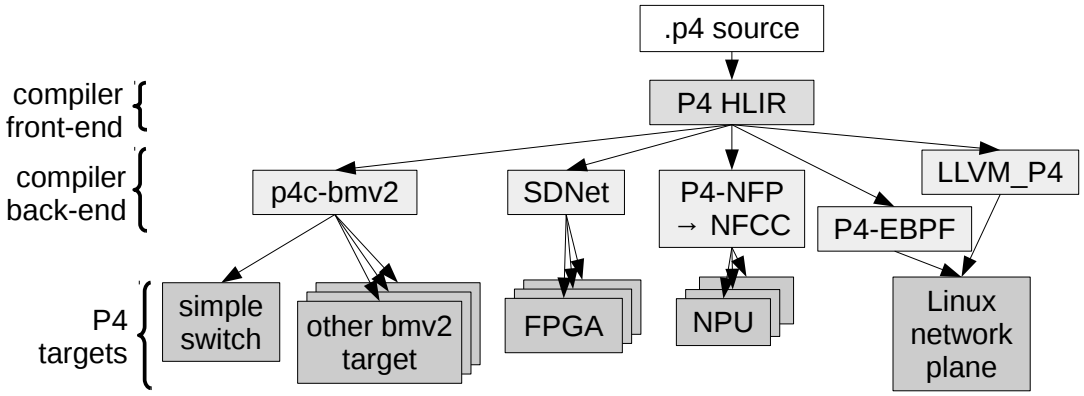
Using these functions, we have constructed three examples of various possibilities enabled by virtualizing the programmable data plane.

3.2 Snapshots and Simple Composition

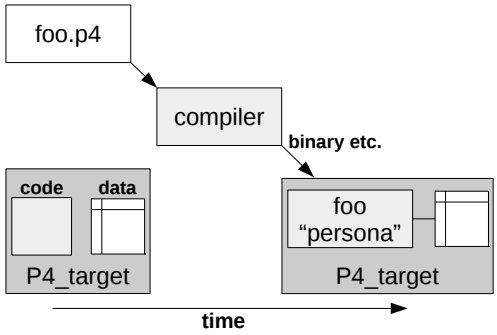
The first (Figure 3) demonstrates network snapshots and composition. The network consists of three connected P4 targets s1, s2, and s3, each running HyPer4 (the state of these is as in Figure 2(a)), two hosts h1 and h2 connected to s1, and two hosts h3 and h4 connected to s3.

At the start, HyPer4 tables are empty and the devices are devoid of functionality. We then populate the HyPer4 tables in each device such that each logically stores all of the programs required for three network configurations (as in Figure 2(b)).

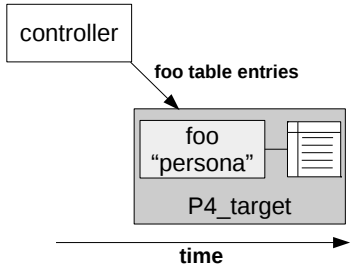
In the first configuration, s1 and s3 each run an arp proxy, while s2 runs a layer 2 switch. Inside s1 and s3 in Figure 3, the dotted-outlined rectangles labeled "A" represent the arp proxy function, while the rectangle labeled "A" within s2 represents the layer 2 switch function.



(a) Compiling a p4 program for various targets

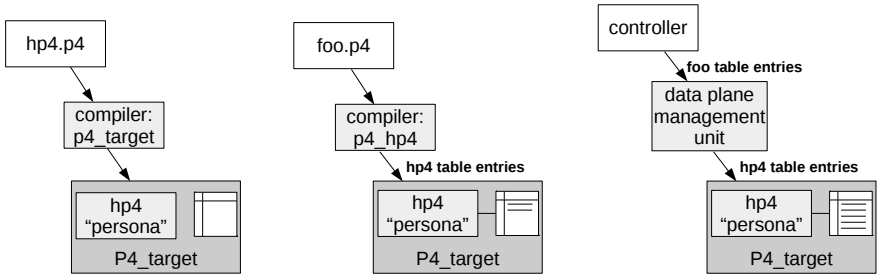


(b) Configuring a p4 target with foo.p4



(c) Populating the tables defined by foo.p4

Figure 1: P4 operational environment



(a) Configuring a P4 target with HyPer4

(b) Configuring HyPer4 with foo.p4

(c) Populating foo's tables within HyPer4

Figure 2: HyPer4 operational environment

In the second configuration, s1 and s3 each run a layer 2 switch, while s2 runs a firewall. These functions are labeled "B" in Figure 3.

In the third configuration, s1 and s3 run the same layer 2 switch as in the second configuration, while s2 runs a composition (labeled "C" in Figure 3). Traffic arriving at s2 is first handled by an arp proxy, which responds to arp requests. All other traffic is passed to the next virtual function in s2, a firewall, and any traffic allowed through the firewall is handled by a router function.

We also logically populate the tables of the virtual functions running within the HyPer4 devices (as in Figure 2(c)):

- MAC and destination port pairs for the layer 2 switches
- IPv4 and MAC pairs for the arp proxies
- IPv4 destination address and next hop IP and MAC address pairs for the router
- In the firewalls, we add rules to filter traffic with a certain TCP destination port

At any given time, a single configuration is active for each device. Changing from one active configuration to

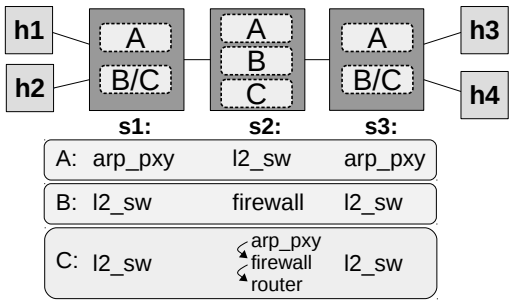


Figure 3: Example One: Network Snapshots and Composition

another is a matter of the controller sending a single table entry modification to each HyPer4 device.

3.3 Network Slicing and Composition

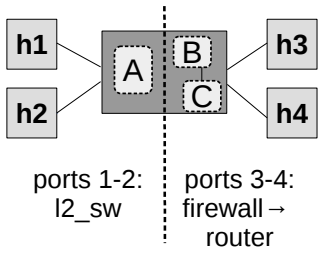


Figure 4: Example Two: Network Slicing and Composition

The second example includes network slicing and composition. Figure 4 depicts a single P4 target s1 running HyPer4, and four hosts h1, h2, h3, and h4 connected to it. IP address and subnet masks are assigned such that h3 and h4 are in separate logical networks.

We use HyPer4 to "slice" s1 such that ports 1 and 2 (for h1 and h2) appear to belong to one device, while ports 3 and 4 (for h3 and h4) belong to another.

At start, HyPer4 tables in s1 are empty, and s1 is devoid of functionality. We then populate the HyPer4 tables such that s1 logically stores three programs:

- Traffic on s1's ports 1 and 2 is handled by a layer 2 switch (program A)
- Traffic on s1's ports 3 and 4 is handled first by a firewall (program B), and secondly, for any traffic that passes through, by a router (program C).

We also logically populate the tables for the layer 2 switch (MAC destination and egress port pairs), the firewall (as in the first example, filtering traffic with a specific TCP destination port), and the router (IPv4 destination address and next hop IP and MAC address pairs).

3.4 Virtual Networking

The third example exhibits virtual networking between virtual devices. The network consists of a single

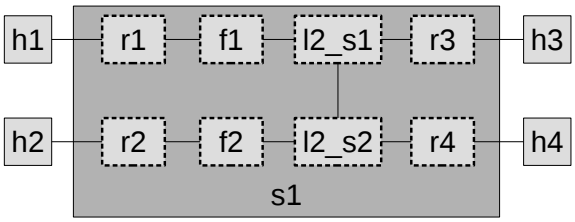


Figure 5: Example Three: Slicing and Composition

P4 target, s1, with four hosts h1, h2, h3, and h4 connected to it. Each host is assigned to a different IPv4 network.

We load eight programs into HyPer4 on s1, creating eight virtual devices:

- a router r1 and a firewall f1 for h1
- a router r2 and a firewall f2 for h2
- a router r3 for h3
- a router r4 for h4
- two layer 2 switches l2_s1 and l2_s2 facilitating connectivity in the internal network

We then populate the tables for each virtual device. This example illustrates how s1, through HyPer4, can support multiple tenants that may wish to provide service to each other but apply security controls.

4. HYPER4 PERSONA DESIGN

At a high level, HyPer4 involves the *persona* (the P4 program running on the network device), a compiler, and a data plane management unit (DPMU). Section 3.1 and Figure 2 provide an overview. In this section, we focus on how the P4 program forming the core of HyPer4 can be made to emulate other P4 programs and virtualize some of the physical resources of the programmable dataplane.

4.1 Design Overview

A P4 program defines the packet-processing *structure*, but during execution, runtime-changeable *state* in the form of table match entries affects how packets are processed within that structure. The aim for HyPer4 is to define a structure general enough that we can permit state changes that change packet processing in arbitrary ways.

Figure 6 depicts an overview of the HyPer4 persona. Conceptually, HyPer4 has three phases. First, the *parsing and setup* phase receives packets and prepares HyPer4 state process packets as specified by the emulated P4 program. In the second phase, HyPer4 emulates the target program's sequence of *match-action stages*. Finally, the *egress* phase handles any egress-specific primitives and prepares the packet for transmission.

In the following paragraphs we identify key tasks the HyPer4 persona must handle in order to properly represent arbitrary P4 programs, and briefly describe the techniques employed to carry out these tasks.

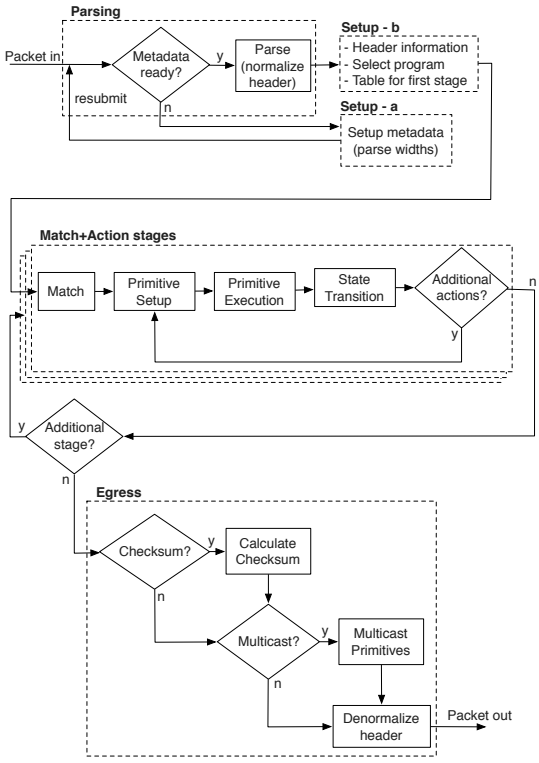


Figure 6: HyPer4 persona

Programmable parsing. The persona must be able to extract an arbitrary number of bytes based on values found in the packet, and it does so by traversing a parse tree, in which each node extracts some portion of the total requirement and branches on a metadata field `numbytes_to_extract` that stores this requirement. The HyPer4 parser is described in more detail in §4.2, but the bottom line is that to achieve the flexibility we need, we implement some aspects of the parser in the setup phase in the ingress pipeline and return to the parser as necessary. Specifically, `numbytes_to_extract` depends on the virtual device associated with the packet. We employ P4's `resubmit` primitive to return the packet back to the parser as necessary. This primitive is powerful because we can pass to it a list of fields that should retain their values (such as `numbytes_to_extract`). Ultimately, the parser extracts a stack of bytes. The `resubmit` primitive could be invoked multiple times for a given packet to accommodate the multiple headers a given virtual device might deal with.

Field representation. The persona uses very wide metadata fields to represent collections of fields used by a virtual device. We define one such field to logically store all of the data extracted from a packet (the stack of bytes extracted by the parser), and another such field to represent all of a virtual device's metadata fields.

Matching. Arbitrary P4 programs match on arbitrary fields. The task for HyPer4 is to support arbitrary matches by isolating the relevant parts of the single wide metadata fields representing a P4 program's defined fields. To this end HyPer4 makes heavy use of P4's ternary matching facility, which permits masks to be supplied with table entries along with match values. These masks are applied to the specified matching field before comparing the result to the match value.

Actions. Matches in P4 programs invoke actions, which may be complex collections of primitives. For each supported P4 primitive, HyPer4 provides a set of tables that collectively carry out the required behavior as discussed in more detail in §4.3. In short, we use a collection of metadata fields to redirect HyPer4 control flow as necessary at every stage of packet processing. Various match-action stages in HyPer4 read (match), write, or read and write these metadata fields. Specifying the values used for these reads and writes by supplying the entries for these tables is how the operator can invoke the relevant behaviors provided by HyPer4 required to carry out arbitrary functionality.

Virtual Networking. HyPer4 relies on P4's `recirculate` primitive to pass packets from one virtual device to another. This primitive marks a packet for sending back to the parser after completing the egress pipeline. It accepts a list of fields as a parameter. Any fields in this list retain their values when the packet reappears at the parser.

The HyPer4 *structure* is anchored by the concept of general purpose match-action stages, in which HyPer4 *state* determines the type of match to peform, the sequences of primitives to execute in response to matches, and the parameters to supply to each primitive. To emulate another P4 program, HyPer4 requires that the target of emulation be transformed into table operations that affect HyPer4 state accordingly. By representing P4 programs as state, HyPer4 enables *live updating* of P4 programs.

4.2 Parsing

The role of the parser in a P4 program is to identify the structure of the first N bits of the packet as a series of bitfields, and associate labels with each of these bitfields. The collection of labeled bitfields is known as the "Parsed Representation" of the packet. The programmer defines header types, indicating the names, widths, and position of bitfields within the header type, and declares header instances of these types to use when extracting bits from the packet. To "extract" a header is to add a collection of bitfields, structured and named according to the header's type, to the Parsed Representation. Enabling the parser to decide which headers to extract is done by making the parse graph branch according to values found in the packet or values of metadata fields.

HyPer4 must parse in a way that is reconfigurable at runtime to extract sufficient data to meet the needs of the various programs HyPer4 is emulating. In HyPer4, we define a header type with a single field that is one byte wide, and declare an array of these one-byte headers to accommodate a variable number of bytes to

extract from the packet. We also declare a metadata field (e.g., `numbytes_to_extract`). Upon receiving a packet, the parser examines this field. A value of zero indicates it is not prepared to guide parsing. In this case, a default number of bytes are extracted (typically, 20), and control flow is directed to an initial setup function (Setup - a in Figure 6), which, if necessary, updates `numbytes_to_extract` and *resubmit*s the packet. The resubmitted packet returns to the parser, but this time, `numbytes_to_extract` has a meaningful value. HyPer4 traverses the parse tree by branching on this field, where each node extracts some portion of the total.

After completing parsing, HyPer4 concatenates all extracted bytes into a single, very wide metadata field that represents the extracted data throughout the remainder of the pipeline. The setup function (Setup - b in Figure 6) then configures the pipeline for packet processing. Specifically, this function sets a metadata field indicating which of possibly several emulated programs within HyPer4 should be executed. It also sets another field identifying the initial table to execute (according to the type of matching required by the first match-action stage of the emulated program) as the packet enters the HyPer4 match-action stage.

4.3 Match-Action

Each emulated match-action stage requires several HyPer4 tables: one to carry out the match, and three for every primitive that must execute in response to a table match.

To carry out the match, we define one table for every combination of match type (exact, ternary, valid, etc.) and data type for every stage. HyPer4 branches on the value of the metadata field `next_table` (set either in an initial setup phase or at the end of the preceding match-action stage) to execute the correct table from this set.

Exact matching against extracted packet data is one match type, for which HyPer4 employs a ternary match against the single, very wide field representing extracted packet data. The ternary match is useful for isolating the bits of this field that are relevant to the match (i.e., to identify which of fields defined in *foo.p4* are involved in the match).

In general, ternary matching helps emulate a variety of match types against a variety of data types because it permits bitmasks to be supplied with table entries at runtime (though comes at a high cost, in unit TCAM cost and power consumption, see §6.3).

In every case, a match triggers an action that sets a variety of metadata fields. These fields force a branch to a control function representing the first primitive of the emulated action. Executing a primitive involves at least three tables: one to set the stage for primitive execution, another to execute the primitive, and another to perform a state transition, which includes indicating to HyPer4 whether the action (and therefore the match-action stage) is complete, or that more primitives must be carried out.

4.4 Deparsing

In P4, throughout packet processing, changes may occur to the packet's Parsed Representation, including not only header field values, but the structure of the packet itself: headers may be removed from or added to the packet. Because of this, *deparsing* is a necessary step that identifies the sequence of headers to serialize for transmission. Deparsers, however, do not appear in P4 source code; rather, the graph specified for the *parser* is once again leveraged for deparsing the packet.

Because HyPer4 uses a metadata field as a proxy for the Parsed Representation throughout packet processing, at the end of the egress pipeline it must perform a "write back" to the actual Parsed Representation to prepare the packet for deparsing. The Parsed Representation consists of a stack of single-byte headers, so preparing for deparsing involves repeatedly masking the proxy metadata field, copying the lowest order byte to the next header in the stack, and shifting the metadata field to the right by one byte.

4.5 Isolation

The current design of HyPer4 isolates programs in the sense that it prevents one program from overwriting the code of another. It also supports memory isolation. We explain these first and then describe how future versions HyPer4 can also support other forms of resource isolation.

Code isolation is achieved by assigning each program a uniquely identifying number at compile time (this could be hash of the .p4 code or downstream compilation artifact). When a packet is received, some operator-controllable criteria (e.g., ingress port, time, network security posture, or a value within the packet itself) determines which program should handle it and accordingly the metadata field `program` is set to the corresponding program ID. This field is one of the match fields for every match-action emulation table in the persona and thereby distinguishes one program's table entries from another's within the same shared physical table. We refer to the emulated program as a *virtual device*. The program ID functions similarly to a VLAN ID, though the mechanism for assigning a program ID to a packet is more dynamic. The DPMU monitors requests for adding table entries to virtual devices and ensures the program IDs in the entries are authorized for the requester.

At the same time, the DPMU can enforce limits for the number of table entries used by each virtual device, assisting partially with the requirement for memory isolation. Other memory requirements include support for stateful memory objects (counters, meters, and registers). HyPer4 design calls for preallocation of sets of such objects, where the number of sets is equal to the number of virtual devices that can be simultaneously

supported. It is the role of the HyPer4 *compiler* to assign these sets to virtual devices.

CPU isolation for our purposes means that no program incurs an action that cannot be completed in a single clock cycle. If one virtual device employs an action that does not fit within a single clock cycle, then for any packet it processes it will introduce stalls in the pipeline that also delay packets behind it (which may belong to other virtual devices). HyPer4 is not currently designed to ensure this kind of isolation. Some of its actions include multiple interdependent primitives. Future designs of HyPer4 will split these actions across multiple tables to keep packets flowing steadily through the pipeline regardless of which functionality is employed by any given virtual device.

The ingress buffer is yet another resource that can lead to interference between virtual devices. In the current HyPer4 design, it is possible that one virtual device might repeatedly trigger primitives that send a packet back to the input buffer, at the possible expense of other packets trying to enter the buffer. Such packets might be coming from external connections or internally from other virtual devices using the same type of primitives. One way to address this issue would be to rely on a meter in HyPer4 at the beginning of the ingress pipeline that drops traffic above a threshold for a given virtual device.

4.6 Virtual Networking

HyPer4 is designed with a virtual network to control traffic exchange between virtual devices running within the persona. Space does not permit a detailed discussion, but we create *virtual* ports with unique IDs and allot them to virtual devices. We can map these virtual ports directly to physical ports or connect them to virtual links, on the other end of which is a virtual port for another virtual device.

For a packet destined for other virtual devices, HyPer4 invokes P4's `recirculate` primitive to send the packet back to the parser after changing the program ID.

We support *virtual multicasting* with a combination of P4's `clone` and `recirculate` primitives. Briefly, the program ID is updated according to a programmable sequence. Then one of the packet clones is sent back to the parser and ultimately processed by the relevent virtual device. The other packet clone is sent back to the start of the egress pipeline, with the program ID serving as a loop counter and triggering a packet drop once it reaches the end of the multicast sequence.

4.7 Design Consequences

This section illuminates the resource and performance consequences of design choices for HyPer4 as well as for the P4 language specification. These consequences for overhead are evaluated in §6.

First, HyPer4's use of ternary matching to emulate a variety of match types against arbitrary data fields increases TCAM pressure, resulting in increased power consumption as well as a potential TCAM scarcity obstacle to deploying HyPer4 on current hardware. See §6.3.

Second, to permit HyPer4 to emulate behaviors from a *behavioral set* in *arbitrary sequences* of maximum length K, HyPer4 source must declare K copies of the tables carrying out a given behavior from the set. This is because P4 restricts a program from using a table more than once in the processing of a specific packet. This design choice of P4 is sensible to ensure programs are compatible with hardware architectures like RMT [12], which opted to connect tables in a pipeline instead of using a crossbar due to physical wiring constraints and because packet processing algorithms naturally involve sequential dependencies.

Next, the use of the `resubmit` primitive to permit a dynamically programmable parser cuts the throughput. HyPer4 could avoid these `resubmit`s with a P4 target that implements parser exceptions by repeatedly extracting bytes until an exception occurred indicating no more bytes available, up to a certain maximum number. Interestingly, Protocol Oblivious Forwarding [43], unlike P4, has a parse-as-needed approach which would also eliminate the HyPer4's need for resubmission.

The use of recirculation for virtual networking also incurs a throughput penalty, but this is a natural consequence of making one physical device do the work of multiple devices.

Finally, HyPer4 can send packets that are, in effect, completely different than what it can effectively receive, which normally is not the case for P4 programs. P4 serializes the parsed representation with the rest of the packet for transmission by using the same graph specified by the parser functions in .p4 code. The result is that a P4 program can only send packets conforming to a structure that it can also parse. HyPer4, however, moves most of the parsing decision logic into the ingress pipeline, and the actual parse graph for HyPer4 simply extracts a specified number of bytes without any higher level structure. Thus, HyPer4 makes an end run around a restriction normally imposed by P4, for better or for worse.

5. IMPLEMENTATION

This section provides implementation details. All source code is available at our git repository [23].

5.1 Configuration

As noted in §4.7, HyPer4 must declare many copies of the tables carrying out a specific behavior, differing in name only, such that one copy exists for every position in a sequence of behaviors. This results in a large codebase with numerous examples of functionally redundant code. As an example, in the source for HyPer4 one might find two tables, `t1_exact_extracted` and `t2_exact_extracted`. They both do the same thing:

exact matching against extracted packet data. The first table is callable in the first emulated match-action stage, while the second table is callable in the second stage.

This redundancy in the code lends itself to using configuration scripts to produce the P4 source for HyPer4. This approach simplifies HyPer4 development and allows us to tailor HyPer4 according to need and resource availability. Configurable parameters include:

- the maximum number of match-action stages HyPer4 must be capable of emulating
- the max number of primitives per compound action
- default, maximum, and step values for the number of bytes HyPer4 should be capable of parsing.

The configuration script (900 LoC in Python) produced the P4 source for a configuration of HyPer4 capable of executing the demonstrations in §3, supporting four emulated match-action stages with up to nine primitives per action, and supporting five of P4's 21 distinct P4 primitives. This configuration of HyPer4 is approximately 6400 LoC. Figure 7 shows the how the P4 codebase grows linearly in the maximum number of match-action stages emulated and the maximum number of primitives allowed per stage. Figure 7(a) charts the growth of the entire HyPer4 codebase, while Figure 7(b) is focused on the code required to support the `drop` primitive, and Figure 7(c) pertains solely to the code supporting the `modify_field` primitive. The average P4 LoC for the five primitives currently supported by HyPer4 ranges from 128, at one stage and one primitive, to 539, at five stages and nine primitives per stage. Extrapolating for the 16 additional primitives (of P4's set of 21), a version of HyPer4 supporting every primitive would require, in addition to the numbers reported in Figure 7(a), from 2000 more LoC at one stage and one primitive, to 8600 more LoC at five stages and nine primitives per stage.

5.2 Compiling

The HyPer4 compiler is a work in progress. In the meantime, we have manually produced files consisting of the table commands necessary to induce HyPer4 to emulate each of the network functions described in §3.1.

These "commands" files consist entirely of command line interface commands in the style of bmv2 [5]. Before producing the final commands file for a given network function, however, it is conducive to produce an intermediate artifact, which for the most part looks like the final commands file, with two key differences:

- the *intermediate* commands file permits comments and vertical whitespace for readability;
- it uses human readable tokens in the place of numbers wherever possible. This is partly for readability, and partly for operational reasons. For example, the virtual program ID as well as the numbers for the assigned virtual ports are not known at compile time and will be supplied at load time.

At load time, we employ a script to convert the intermediate commands file to the HyPer4-ready commands file, substituting numbers for tokens and eliminating comments and whitespace. Script parameters include the program ID and virtual port assignments.

5.3 Limitations

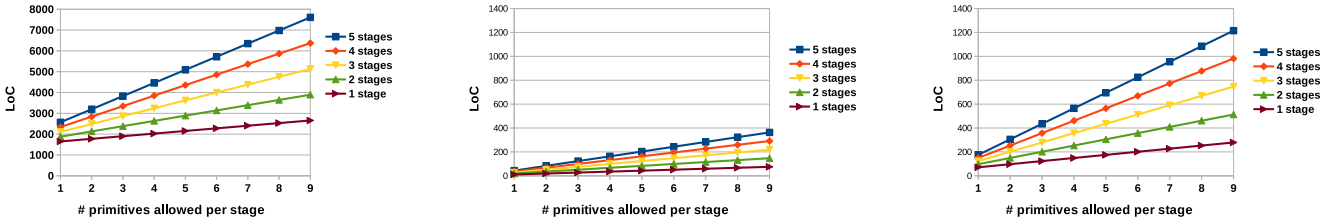
Some of the P4 language features are not currently covered by HyPer4, but will be in time, though the level of coverage may vary by feature. A long term goal is to one day support popular P4 applications like NetPaxos [18] and In-band Network Telemetry [26].

Stateful memory (registers, counters, meters). It is difficult for HyPer4 to anticipate the needs of a wide variety of programs, with respect to registers, counters, and meters, and meet them within the limited amount of physical resources available. The problem is compounded by the fact that P4 permits different modes of declaration for stateful memories. A stateful memory object may be globally accessible, or statically bound to a specific table, or furthermore statically bound to a specific table entry. HyPer4 can preemptively declare statically bound registers, counters, and meters for every table that emulates the "match" piece of a match-action stage, but this approach is likely to prove infeasible for many hardware P4 targets. In particular, registers can vary in width, which for the preemptive approach would require HyPer4 registers be declared with sufficient width to cover the maximum need, resulting in a lot of wasted register memory for the average case.

Match types: lpm, range. The single wide `extracted_data` field employed by HyPer4 to represent extracted packet fields complicates implementation of lpm matching, but we have at least two options. We can insert a preparatory match-action stage that takes the `extracted_data` field and copies, to a separate metadata field, only the bits pertaining to the represented field of interest. The lpm match is then done against this metadata field, which is made wide enough to accommodate a variety of needs. The second option is to use ternary matching, but have the DPMU identify and manage the priorities of match entries.

Range matching implementation may involve a preparatory copy step to break out the field of interest as in the first option for `lpm` matching.

Arbitrary checksums. HyPer4 cannot easily handle arbitrary checksums, by which we mean applying a checksum algorithm to an unpredictable number of fields. Though we can arbitrarily create a field list by using copies, shifts, and masks to access the emulated fields and concatenate them together into another field, we cannot predict how wide that new field must be, and if wider than the field list defined in the original program, the leading (or trailing) zeros will affect the output of many checksum algorithms, which are often designed to distinguish between messages that are in all ways equivalent except that one message has leading (or trailing) zeros and the other does not [29]. We could, however, declare one such field list-emulating field for each of many different widths we anticipate needing to



(a) Total Lines of Code (b) Drop Lines of Code (c) Modify_Field Lines of Code

Figure 7: HyPer4 Lines of Code by Number of Stages and Primitives per Stage

support, allow the user to indicate the required width via table entry, and direct control flow accordingly to make use of the correct field. In the meantime, we can "cheat" by directly adding support for the checksum requirements of well known protocols. This is what we have done with the IPv4 checksum field.

Field lists. A P4 program may define *field lists*. A field list is passed as a parameter to `clone`, `resubmit`, and `recirculate` primitives and identifies those fields whose values must be retained when the packet reappears at the start of the appropriate pipeline. Because HyPer4 represents all of the virtualized program's extracted data fields in one wide field and all of the metadata fields in another, we can construct a universal field list that includes the consolidated extracted data field e, the consolidated metadata field m, and bitmasks for each, b_e and b_m. At the start of each pipeline, we replace e with e & b_e and m with m & b_m.

Expressions and **action profiles** likely will never be covered by HyPer4.

6. EVALUATION AND ANALYSIS

Our development environment for HyPer4 does not include a P4-capable hardware device. Therefore, to evaluate HyPer4 we examine resource and performance differences in terms that may be used to estimate performance impacts or realizability on any P4-capable device. Specifically, we compare selected native P4 programs with these same programs emulated by HyPer4 in terms of the number of table match-action stages incurred during operation (impacting latency), the space requirements for tables, data, and actions (impacting memory), and the width and frequency of ternary matches (impacting TCAM and power).

We also directly evaluated HyPer4 performance in terms of latency and bandwidth with the use of Barefoot Networks' bmv2 software switch [5], and explain results with discussion about the number of resubmits and recirculations involved (impacting throughput).

Finally, we analyze the difference between RMT [12] specifications and the hardware required to run HyPer4.

6.1 Match-action Stages

Table 1 shows the cost of HyPer4 emulation in terms of the number of match action stages required. Each

	No. Matches	
Program	Native	HyPer4
L2 switch	2	13
Firewall	3	22
Router	4	28
Arp Proxy	4	48

Table 1: Number of matches for most complex processing per function natively vs. in HyPer4

row of Table 1 involves a switch running a single function. Of these, HyPer4 emulation typically requires 6x to 7x as many match-action stages, except the arp_proxy, which requires 12x as many. This is because one of the actions of arp_proxy involves nine primitives in order to build an arp response. Note, this 12x penalty is only incurred when an arp request is received.

6.2 Space

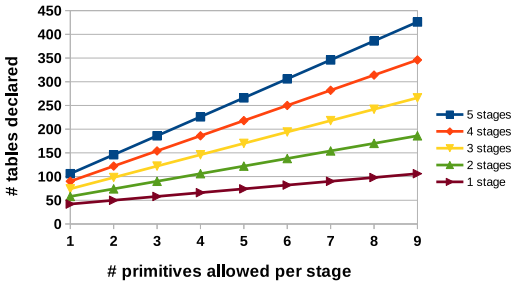


Figure 8: HyPer4 Tables by Number of Stages and Primitives per Stage

Figure 8 charts the number of tables declared in HyPer4 with different numbers of emulated match-action stages and numbers of primitives allowed per stage. Most of these tables will have no entries but exist to meet arbitrary emulation requirements. But these empty tables still occupy space; we expect most implementations of tables include code or function pointers for carrying out the match, and function pointers for actions associated with the table. In most cases, each HyPer4 table references a single action, but for some (e.g., ta-

bles that carry out the `modify_field` primitive) this can range up to 14.

The configuration used throughout HyPer4 testing permits four match-action stages and nine primitives per stage, and declares 346 tables.

	No. Tables Shared			
	l2_sw	arp_proxy	router	firewall
l2_sw	19	14	17	14
arp_proxy		57	23	30
router			33	21
firewall				35

Table 2: Number of tables referenced by both programs

During execution, if HyPer4 hosts multiple programs, many of the tables are shared. For each pair of programs, Table 2 shows the number of shared tables. The diagonal indicates the total number of tables referenced by each program. Not all of these tables are in the same branch for any given program, and hence these numbers differ from those reported in Table 1.

	No. Tables Uniquely Referenced			
	l2_sw	arp_proxy	router	firewall
l2_sw	-	5	2	5
arp_proxy	43	-	34	27
router	16	10	-	12
firewall	21	5	14	-

Table 3: Number of tables referenced uniquely in the program named at the left

Table 3 reveals the number of tables that do not share entries for each pair of programs. Clearly, `arp_proxy` has the most unique tables. This is accounted for by the fact that this program is alone in terms of executing a nine-primitive action in one of its stages.

In eight out of twelve cases, more tables are shared between programs than not, suggesting that table declarations is one area where the overhead of HyPer4 is amortized over multiple emulated programs.

Storage required for table entries, however, is much higher within HyPer4 in return for flexibility. Because each match involving extracted packet data is done against a single 800-bit-wide field, each match entry requires at least 1600 bits of storage: 800 for the value and 800 for the mask bits. Similarly, matches involving emulated metadata are done against a single 256-bit-wide field and require at least 512 bits per entry. The program ID adds a few more to the required number of bits for each entry. This is the price we pay for allowing a single table to be used to match against any of the first 800 bits of a packet, or 256 bits of metadata.

With respect to action storage, the current incarnation of HyPer4 includes 130 actions, many of which are functionally redundant, differing only in a certain constant that cannot be replaced by a variable (field). For example, HyPer4 includes 80 actions that each resize the parsed representation to the correct size between 20 and 100 bytes (accounting for packet structure changes, such as adding or removing headers) prior to writing back any changes to extracted data and deparsing. With few exceptions, most HyPer4 actions use between one and four primitives. The 130 tally does not change with different configurations of HyPer4; it only changes as we add or remove fundamental functionality to HyPer4 such as support for more primitives or revisions to virtual networking.

6.3 Ternary Matches

The increased reliance on ternary matching increases power consumption. Many efforts discuss the TCAM power issue, including [31], [4], [44], [3]. In the future we plan to evaluate this aspect of HyPer4 more closely, but in the meantime provide a few numbers to give an idea of HyPer4's increased pressure on TCAM.

	bits matched per packet		
program	total	active	no. ternary matches
l2_sw	808	56	2
router	1224	80	4
arp_proxy	1848	66	5
firewall	1928	59	6

Table 4: Ternary Match Usage in HyPer4

Table 4 shows how many bits are ternary matched for packets incurring the most complex processing for each program. The "total" column includes "don't care" or wildcard bits while the "active" column shows only the number of bits actively involved in the match. The number of ternary matches, resulting in the totals for these columns, is provided in the fourth column.

6.4 Latency and Bandwidth Measurements

Here we report the results of bandwidth and latency tests using iperf3 [19] and ping floods. We ran all tests using bmv2 software switch [5]. All evaluations were carried out in an Ubuntu 14.04 virtual machine running Mininet [1]. The VM was allocated three 2.2 GHz Intel Core i7-4770HQ CPUs and 6.15 GB of memory.

	native Mbps		hp4 Mbps		native ms		hp4 ms	
	μ	σ	μ	σ	μ ms	σ	μ ms	σ
l2_sw	110.3	4.81	18.7	0.43	451	81.4	1540	31.0
firewall	63.7	3.53	7.2	0.11	483	63.0	2277	38.7
Ex. 1 B	37.7	1.24	6.3	0.10	1454	113.2	5011	90.7
Ex. 1 C	26.3	1.06	3.1	0.15	2247	235.1	8736	63.3

Table 5: Bandwidth (iperf3) and Latency (ping -f -c 1000)

Table 5 shows the mean and standard deviation of 10 runs of each test. HyPer4 emulation incurs an 83% bandwidth penalty for the L2 switch as measured with `iperf3`, and an 89% bandwidth penalty for the firewall.

For the latency test, we sent 1000 pings using the `-f` option for "flood," in which each successive ping is sent immediately after a response is received for the previous ping. HyPer4-emulated L2 switch takes 3.4x as long as the native L2 switch, while for the firewall the difference is 4.7x.

The emulated L2 switch never invokes the `resubmit` primitive, so all performance penalty is due to the increased number of match-action stages. The emulated firewall performs one `resubmit` for each ping, and two `resubmit`s for each TCP packet sent in the iperf3 test.

The next two rows involve multiple switches. "Ex. 1 B" is described in §3.2 and Figure 3: three switches (L2 switch, firewall, L2 switch) between two hosts. "Ex. 1 C" has the same topology but the middle switch runs the sequentially composed arp proxy, firewall, and router. When HyPer4 is loaded on each switch, the performance drop is similar to the previous two cases: 83% bandwidth penalty for "Ex. 1 B" and 88% penalty for "Ex. 1 C". The latency test also has similar results: HyPer4 switches take 3.4x as long in "Ex. 1 B" and 3.9x as long in "Ex. 1 C".

For "Ex. 1 B", each ping triggers one `resubmit`, while each TCP packet requires two `resubmit`s. For the composition on the middle switch of "Ex. 1 C", pings incur a total of two `recirculation`s and two `resubmit`s, while TCP packets result in a total of two `recirculation`s and three `resubmit`s.

6.5 Deploying on RMT

Implementation details for RMT [12] that determine ability to run HyPer4 include the width of the packet header vector (which is made available to each match-action stage and includes bits extracted from the packet as well as all bits used for metadata), the number of physical match action stages available, and the maximum number of bits each stage can match against.

RMT supports a 4096-bit packet header vector. This meets the requirement for the HyPer4 configuration evaluated here, which uses 3312 bits: 800 bits of extracted packet data, 256 bits of metadata, and 2256 bits of overhead. This overhead includes scratch space supporting manipulations on packet data, which affects any decision regarding how to make use of the remaining packet header vector bits available, whether for extracting more bytes from the packet or representing more metadata.

RMT includes 32 match-action stages in the ingress pipeline and another 32 in egress. The single program evaluated in this section requiring the most match-action stages is the `arp proxy`, which uses 46 match action stages in the ingress pipeline (and two in egress). RMT cannot support this requirement, but for more precision, we compare the number of bits each RMT match-action stage supports to the number of bits HyPer4 match-action stages require, to know in which cases we need multiple physical stages to support a given HyPer4 match-action stage. RMT stages support matches of up to 640 bits in SRAM and 640 bits in TCAM. Of `arp proxy`'s 46 ingress stages, 44 of these can each be covered by one RMT physical stage. The other two stages involve ternary matches against the 800 bit-wide field representing extracted packet data, and each require 1600 bits of TCAM memory (800 for the value, 800 for the mask). Therefore, three RMT physical stages are required for each of these two HyPer4 stages. The final total is 51 physical stages, or 60% more than RMT's capacity. Note that this would meet not only `arp proxy`, but arbitrary compositions including `arp proxy` and any simpler programs. A variant of RMT that shifted 19 of the 32 egress match action stages to the ingress pipeline could meet the requirement today.

7. DISCUSSION OF ALTERNATIVES

The performance penalties incurred by HyPer4 illuminated in § 6 warrant consideration of alternative approaches. This section discusses alternatives to the "full" HyPer4 solution.

7.1 Partial Virtualization

In an environment in which high performance network hardware is too costly to consider spending on HyPer4's capabilities, it may yet be attractive to employ *partial* virtualization. This approach requires fewer physical resources, or may boost performance by sacrificing a portion of flexibility, or compensate for HyPer4's inability to emulate some P4 language feature.

The P4 programming model provides mechanisms to program a packet parser as well as match-action tables (§2.2). For these program elements, various combinations of virtualization and direct implementation are possible to satisfy various use cases.

Figure 9(a) depicts the "full" virtualization, in which each column runs its own program. Each column could be active simultaneously (network slicing), and packets are sent to specific columns by some criteria.

Figure 9(b) illustrates the possibility of separate virtual parsers attached to the same directly implemented match-action pipeline. This pipeline will not be as dynamically flexible as the pipeline in the full virtualization solution, yet may offer limited flexibility by permitting runtime selectable criteria to invoke different directly-implemented functionality based on which packet headers have been extracted.

Conversely, as shown in Figure 9(c), a single directly implemented parser can pass traffic to different virtual match-action pipelines. This "fixes" the set of protocol headers supported, but permits different, dynamically modifiable behaviors as responses to these headers.

Figure 9(d) suggests the match-action pipeline itself could be partially virtualized. This may be attractive for environments in which we are willing to sacrifice dynamic flexibility in one layer of the protocol stack (perhaps L2 or L3) for performance, but still need dynamic flexibility in other layers.

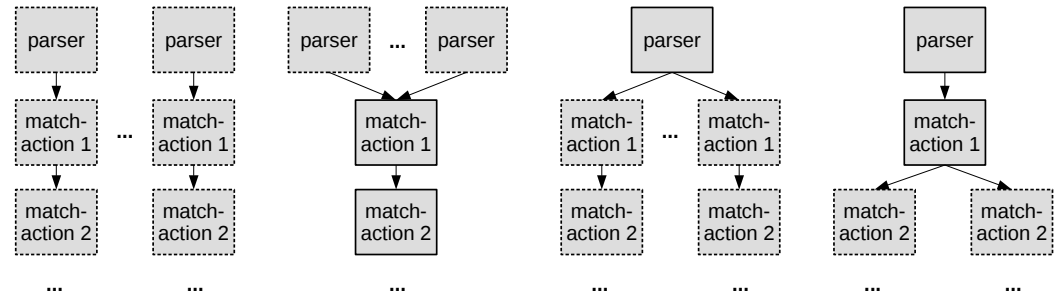


Figure 9: Possible mixes of programmable data plane virtualization

7.2 Other Alternatives

Other approaches for increasing data plane flexibility:

1. A compiler for composing functions written in P4. This tool would produce a program with functionality merged from more than one P4 project. Compiler flags could allow the operator to indicate control flow across the set of included functions using parallel, sequential, and branch composition operators, in the same spirit as Pyretic [32]. This approach should result in high performance, but cannot provide the runtime reconfigurability provided by our approach.

2. Virtualization implemented below the application layer. This approach could offer runtime reconfigurability along with high performance, but requires non-portable implementation aspects specific to each platform.

3. Directly embedding P4 programs in the network. This technique entails loading different P4-capable switches with different programs, and another switch that selects the appropriate target for processing the traffic according to the criteria that marks a packet as belonging to one virtual network or another. This approach should offer high performance and the possibility of runtime reconfigurability (if spare hardware is available) at the cost of additional switches.

8. RELATED WORK

HyPer4 builds on the P4 language [11, 46]. Perhaps most closely related to our efforts is the P4 parser for OpenvSwitch, to enable flexible packet parsing in OVS, such that supported protocols and fields can be reconfigured without recompiling [37]. This effort focuses on the header and parser elements of P4 programs and is specific to extending OVS.

HyPer4 is also related to earlier work on data plane programmability [27, 45, 28, 10, 33, 25, 13, 36]. This includes the Click modular router [27] and active networks [45]. NetFPGA also represents earlier work that provided data plane programmability [28]. While not fully programmable, efforts that enabled stateful openflow applications to be realized also formed part of the current move towards fully programmable switch data planes [10, 33]. More recent efforts more closely related to P4 include Protocol Oblivious Forwarding [43], tiny packet programs [25] and work on programmable hardware platforms [13, 36].

More broadly, HyPer4 remains related to earlier SDN efforts such as work enabling portability in the SDN control plane [50], and the creation of a control plane virtualization layer [41].

HyPer4 is also related to network function virtualization research [38, 2, 48, 20], particularly concerning efficient data plane realizations [48, 24, 17].

9. CONCLUSIONS

We have described the concept, design, implementation, and evaluation of a hypervisor-like P4 program called HyPer4. It is as yet only capable of emulating a small subset of possible P4 programs, and typically incurs 80% to 90% penalties in terms of performance, in exchange for a portable solution for programmable data plane virtualization that permits reconfigurability *at runtime*. It facilitates useful modes for running network functions in parallel or virtually networked in complex compositions, and is a platform onto which we can add useful modules for features like monitoring and program verification.

Future work includes expanding HyPer4's coverage of P4 language features, improving HyPer4 efficiency, completing the HyPer4 compiler, and providing a data plane management unit.

10. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable feedback, and our shepherd for his guidance and encouragement. This material is supported by the National Science Foundation under grant number 1302688.

11. REFERENCES

[1] Mininet. http://mininet.org/.

[2] OPNFV: An Open Platform to Accelerate NFV. https://www.opnfv.org/sites/opnfv/files/pages/files/opnfv_whitepaper_092914.pdf.

[3] B. Agrawal and T. Sherwood. Modeling tcam power for next generation network devices. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 120–129. IEEE, 2006.

[4] S. I. Ali and M. Islam. An energy efficient design of high-speed ternary cam using match-line segmentation and resistive feedback in sense amplifier. *Journal of Computers*, 7(3):567–577, 2012.

[5] Barefoot Networks. Behavioral Model Repository. https://github.com/p4lang/behavioral-model.

[6] Barefoot Networks. Bmv2 Simple Switch. https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch.

[7] Barefoot Networks. P4-hlir. https://github.com/p4lang/p4-hlir.

[8] Barefoot Networks. P4c-bm. https://github.com/p4lang/p4c-bm.

[9] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An architecture for active networking. In *Proceedings of the IFIP TC6 Seventh International Conference on High Performance Networking VII*, HPN '97, pages 265–279, London, UK, UK, 1997. Chapman & Hall, Ltd.

[10] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, Apr. 2014.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, and C. Schlesinger. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[12] 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 *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[13] 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.

[14] Brebner, Gordon. P4 for an FPGA target. http://schd.ws/hosted_files/p4workshop2015/33/GordonB-P4-Workshop-June-04-2015.pdf.

[15] Budiu, Mihai. Compiling P4 to EBPF. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.

[16] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[17] I. Cerrato, M. Annarumma, and F. Risso. Supporting fine-grained network functions through intel dpdk. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 1–6, Sept 2014.

[18] 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*, page 5. ACM, 2015.

[19] ESnet, Lawrence Berkeley National Laboratory. iperf3. http://software.es.net/iperf/.

[20] ETSI. Network Functions Virtualisation (NFV); Architectural Framework. ETSI GS NFV 002 V1.1.1 (2013-10).

[21] Fastabend, John. P4 on the Edge. http://p4.org/p4-workshop-2016/.

[22] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[23] Flux Research Group, University of Utah. HyPer4 Repository. https://gitlab.flux.utah.edu/hp4/src.git.

[24] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, Apr. 2014. USENIX Association.

[25] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 8:1–8:7, New York, NY, USA, 2013. ACM.

[26] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and

M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[28] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, pages 160–161, June 2007.

[29] H. McKee. Improved {CRC} techniques detects erroneous leading and trailing 0âĂŹs in transmitted data blocks. *Computer Design*, 14(10):102–4, 1975.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2), Mar. 2008.

[31] N. Mohan, W. Fung, D. Wright, and M. Sachdev. A low-power ternary cam with positive-feedback match-line sense amplifiers. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(3):566–573, 2009.

[32] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[33] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 61–66, New York, NY, USA, 2014. ACM.

[34] Netronome Systems, Inc. Programming NFP with P4 and C. https://netronome.com/media/redactor_files/WP_Programming_with_P4_and_C%20.pdf.

[35] Open Networking Foundation. OpenFlow Switch Specifications. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

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

[37] Pfaff, Ben. P4 Parsing in Open vSwitch. http://p4.org/p4-workshop/.

[38] A. Rajan, S. Gobriel, C. Maciocco, K. Ramia, S. Kapury, A. Singhy, J. Ermanz, V. Gopalakrishnanz, and R. Janaz. Understanding the bottlenecks in virtualizing cellular core network functions. In *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, pages 1–6, April 2015.

[39] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an ip services protocol. *RFC3549, July*, 13, 2003.

[40] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. *AT&T Research Academic Summit, Bedminster, NJ, USA*, 2016.

[41] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[42] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. Feldmeier, and W. D. Sincoskie. Switchware: accelerating network evolution (white paper). Technical Report MS-CIS-96-38, CIS Department, University of Pennsylvania, 1996.

[43] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013.

[44] S. Tabassum, F. Parveen, H.-u. Rashid, et al. Low power high speed ternary content addressable memory design using 8 mosfets and 4 memristors-hybrid structure. In *Electrical and Computer Engineering (ICECE), 2014 International Conference on*, pages 168–171. IEEE, 2014.

[45] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *Communications Magazine, IEEE*, 35(1):80–86, Jan 1997.

[46] The P4 Language Consortium. The P4 Language Specification. http://p4.org/spec/.

[47] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *Open Architectures and Network Programming, 1998 IEEE*, pages 117–129. IEEE, 1998.

[48] T. Wood, K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang. Toward a software-based network: integrating software defined networking and network function virtualization. *Network, IEEE*, 29(3):36–41, May 2015.

[49] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and control element separation (forces) framework. Technical report, RFC 3746, April, 2004.

[50] M. Yu, A. Wundsam, and M. Raju. Nosix: A lightweight portability layer for the sdn os. *SIGCOMM Comput. Commun. Rev.*, 44(2):28–35, Apr. 2014.