The P4→NetFPGA Workflow for Line-Rate Packet Processing

Stephen Ibanez
Stanford University
sibanez@stanford.edu

Gordon Brebner
Xilinx Labs
gjb@xilinx.com

Nick McKeown
Stanford University
nickm@stanford.edu

Noa Zilberman
University of Cambridge
noa.zilberman@cl.cam.ac.uk

ABSTRACT

P4 has emerged as the *de facto* standard language for describing how network packets should be processed, and is becoming widely used by network owners, systems developers, researchers and in the classroom. The goal of the work presented here is to make it easier for engineers, researchers and students to learn how to program using P4, and to build prototypes running on real hardware. Our target is the NetFPGA SUME platform, a 4×10 Gb/s PCIe card designed for use in universities for teaching and research. Until now, NetFPGA users have needed to learn an HDL such as Verilog or VHDL, making it off limits to many software developers and students. Therefore, we developed the P4→NetFPGA workflow, allowing developers to describe how packets are to be processed in the high-level P4 language, then compile their P4 programs to run at line rate on the NetFPGA SUME board. The P4→NetFPGA workflow is built upon the Xilinx P4-SDNet compiler and the NetFPGA SUME open source code base. In this paper, we provide an overview of the P4 programming language and describe the P4→NetFPGA workflow. We also describe how the workflow is being used by the P4 community to build research prototypes, and to teach how network systems are built by providing students with hands-on experience working with real hardware.

1 INTRODUCTION

Networking switches, routers, and network interface cards (NICs) have traditionally been dominated by ASICs that process packets using a *fixed function* pipeline. While some programmable devices are used (e.g. NPUs [23], FPGAs [38], CPUs [30]), conventional wisdom in networking states that programmable forwarding devices are slower, more expensive and consume more power. However, this is being challenged by a new breed of programmable switches and NICs matching the performance, power and cost of fixed-function devices [7, 8, 25]. Network system designers are exploiting the programmability to add new features to the forwarding plane, including telemetry [19], layer-4 load balancing [21], encryption, and in-network caching [16].

With various programmable forwarding devices now available, what high-level language should be used to program them? This was the motivation for the creation of the P4 language [4], which in recent years has become the first *de facto* language for programming forwarding devices.

The P4 language was designed with three goals in mind:

- Protocol independence — network devices should not be hard coded to support specific protocols.
- Field reconfigurability — programmers should be able to change the fundamental packet processing behavior of network devices after they have been deployed.
- Portability — packet processing programs should not be tied to a specific device.

There are many benefits to using P4 for programming network devices. Network operators can easily add support for new features into their network devices. Programmers can remove the features that they are not using in order to reduce complexity in their networks. Oftentimes network failures are caused by interactions between protocols that network operators do not even know they are using. The memory and compute resources within network devices can be flexibly allocated amongst the desired features. Programmable data planes are also bringing about much greater visibility into the network as new diagnostic techniques start to emerge, such as In-band Network Telemetry (INT) [19]. The P4 programming model brings with it a software-style development process which enables a rapid design cycle, fast innovation, and the ability to fix data plane bugs in the field. Additionally, network operators are able to keep their own ideas because they do not need to share their P4 programs with anyone. Hence, companies are able to maintain a competitive advantage.

An FPGA-familiar reader will recognize that the benefits of P4 described above directly intersect with benefits of using programmable hardware such as FPGAs. So one question that directly follows from this observation is why FPGAs, and HDLs such as Verilog or generic high-level synthesis, are not just used to implement all network devices. One of the reasons is because of the performance gap between FPGAs and ASICs. While the performance gap has decreased in recent years, top of the line ASICs can still process packets about an order of magnitude faster than top of the line FPGAs [12]. Another reason is because the steep learning curve that is required to program FPGAs using hardware description languages hinders the ability of network programmers to implement new features rapidly. Notwithstanding this, many network devices are in fact implemented using FPGAs, especially network interface cards [12], but typically the implementation has to be done by hardware experts.

Operating at a higher level of domain-specific abstraction, the P4 language identifies the key primitives that are used to build packet processing devices. P4 programs can then be compiled onto any device that supports the language's underlying primitives, including high speed packet processing ASICs [25], NPUs [24], software switches running on CPUs [33], and FPGAs [38].

The need to enable the programming of FPGAs using P4 therefore has a twofold motivation:

- Operators want to have a standard way to configure the behavior of their network devices, in order to ease the burden of managing networks. The industry is converging on using P4 for this purpose, and hence packet processing FPGAs within such devices must become P4 programmable.

- FPGAs are a useful platform on which to prototype P4 designs at hardware line rates that may later be deployed on other network devices with more customized hardware.

The P4 community has produced an open source software emulation environment [28], which has proved to be an extremely useful tool that developers can use to prototype P4 applications. However, this software platform has some significant limitations; most notable is the limited performance which makes it impossible for developers to gauge how their application will perform in a realistic environment. Furthermore, the software target is very flexible and does not realistically capture the constraints of an actual hardware implementation. For these reasons, we believe the P4 community can embrace FPGAs as the basis of an open source hardware environment to resolve these difficulties. In particular, this paper adopts the existing open source NetFPGA family as the hardware platform for research and development of P4 programs. The NetFPGA platform is a low-cost, open-source, FPGA-based networking device, which has been specifically designed for teaching and research.

This paper provides a tutorial introduction to the P4→NetFPGA workflow:

- We provide an overview of the P4 language (Section 2), P4-SDNet (Section 3) and the NetFPGA platform (Section 4).
- We describe the main new contribution which ties these components together: the open source P4→NetFPGA workflow (Section 5), its building blocks, and its operation.
- We discuss use cases of the P4→NetFPGA workflow in research and teaching (Section 6) and its future roadmap (Section 7).
- Finally, we provide information on how to get started immediately on using the P4→NetFPGA workflow (Section 9).

2 P4 LANGUAGE OVERVIEW

This section provides a brief overview of the P4 programming language. The goal is not to provide a comprehensive description, but rather just enough detail for a reader to grasp the fundamental concepts.

Figure 1 depicts the general process of programming a P4 device. The vendor of a packet processing device provides three components to the user:

- The packet processing target device.
- A P4 architecture model to expose the programmable features of the target to the P4 programmer.
- A compiler to map the user's P4 program into a target-specific configuration binary file which is used to tell the target how it should be configured to process packets.

The user provides a P4 main program to instantiate the architecture model, by filling in its programmable components. The user also provides control software (i.e. a control plane) which is responsible for controlling the packet processing device at run time.

In order to make network devices protocol independent, i.e. without built-in implementations of specific protocols, P4 programmers define the format of all protocol headers that they want the device to handle. Here is an example that shows how a programmer might

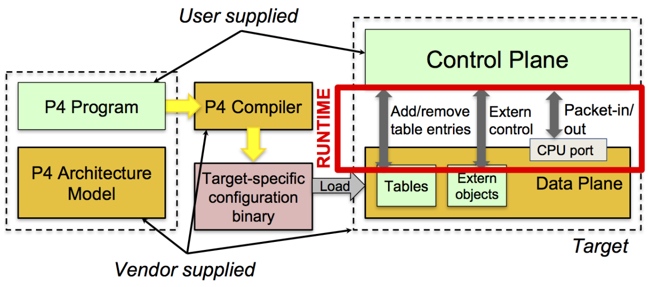

Figure 1: The process of programming a P4 target.

define the Ethernet and IPv4 headers. Note that `typedef` statements can be used to make the code more readable.

```
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
header ethernet_t {
  macAddr_t dstAddr;
  macAddr_t srcAddr;
  bit<16>   etherType;
}
header ipv4_t {
  bit<4>    version;
  bit<4>    ihl;
  bit<8>    diffserv;
  bit<16>   totalLen;
  bit<16>   identification;
  bit<3>    flags;
  bit<13>   fragOffset;
  bit<8>    ttl;
  bit<8>    protocol;
  bit<16>   hdrChecksum;
  ip4Addr_t srcAddr;
  ip4Addr_t dstAddr;
}
struct headers {
  ethernet_t   ethernet;
  ipv4_t       ipv4;
}
```

A P4 architecture model can contain P4-programmable elements of two types: parsers and control blocks. Parsers are responsible for extracting headers out of an incoming stream of bytes. They are implemented as a finite state machine with three predefined states: `start`, `accept`, and `reject`. An implementation may also contain other user defined states. Parsers always start in the `start` state, execute one or more statements, then transition to the next state until reaching either the `accept` or `reject` state.

Below is an example implementation of a P4 parser. In this simple example, the parser first uses the `packet_in` object's `extract` method to fill out the fields of the Ethernet header. It then transitions to either the `parse_ipv4` state or the `accept` state based on the value of the Ethernet header's `etherType` field. Within the `parse_ipv4` state, the parser simply extracts the IPv4 header and then transitions to `accept`.

Table 1: Example entries for the forward table.

Key	Action ID	Action Data
1	set_output_port ID	2
2	set_output_port ID	1

```
parser MyParser(packet_in packet,
                out headers hdr) {
  state start {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      0x800  : parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
  }
}
```

Control blocks can be used to represent both match-action packet processing logic as well as packet deparsers. A match-action processing block uses tables, actions, and imperative code to manipulate input headers and metadata. This match-action packet processing model was originally introduced as the centerpiece of the OpenFlow model for Software Defined Networking (SDN) [20].

When a P4 programmer defines a match-action table, they declare various properties such as the header and/or metadata field(s) to match upon, the type of match to be performed, a list of all possible actions that can be invoked, the number of entries to allocate for the table, and a default action to invoke if no match is found. A table entry contains a specific key to match on, a single action to invoke when the entry produces a match, and any data to provide to the action when it is invoked. Table entries are populated at run time by the control plane software.

The following is an example p4 match-action control block implementation and Table 1 shows how the forward table might be populated. This simple example will forward all IPv4 packets that arrive on port 1 to port 2; and all IPv4 packets that arrive on port 2 to port 1. All other packets will be dropped.

```
control MyMatchAction(inout headers hdr,
                      inout std_meta_t std_meta) {
    action set_output_port(bit<8> port) {
        std_meta.output_port = port;
    }
    action mark_to_drop() {
      std_meta.drop = 1;
    }
    table forward {
        key = { std_meta.ingress_port: exact; }
        actions = {
            set_output_port;
            mark_to_drop;
        }
        size = 1024;
        default_action = mark_to_drop();
    }
    apply {
      if (hdr.ipv4.isValid()) {
        forward.apply();
      } else {
        mark_to_drop();
      }
    }
}
```

Deparsers are special cases of control blocks that perform the inverse operation of parsers. Their job is to reassemble the packet headers onto an outgoing packet byte stream. A header is added to the packet using the packet_out object's emit method. Deparsing is implemented using the P4 control block mechanism because it only involves sequential logic as used for actions. Below is an example deparser implementation: it simply reinserts the Ethernet and IPv4 headers back into the packet.

```
control MyDeparser(packet_out packet,
                   in headers hdr) {
  apply {
    packet.emit(hdr.ethernet);
    packet.emit(hdr.ipv4);
  }
}
```

In addition to defining the interfaces of all parser and control blocks, a P4 architecture definition also defines the format of any standard metadata buses as well as the set of externs that can be invoked within P4 programs. Standard metadata buses, conveying sideband data alongside each packet, are used to allow P4-programmable elements to interact with non-programmable elements within the architecture. Externs are used to execute device specific logic; their implementation is not described in P4 and programs only see the inputs and outputs.

3 XILINX P4-SDNET OVERVIEW

The Xilinx SDNet product was originally built as a design environment centered around an internally-created packet processing language called PX [5], which pre-dated P4 by a number of years. The goals and capabilities of PX and P4 intersect in many — indeed most — ways. As the networking community has converged on using P4 as the standard language, Xilinx has embraced the change and, in the first instance, added a P4 to PX translator to the SDNet design environment. Figure 2 depicts the process of compiling P4 programs using this version of SDNet. The front end translator maps P4 programs into corresponding PX programs and also produces a JSON file with information about the design that is required by the runtime control software. The PX program is passed, along with configuration parameters, into SDNet which then produces an HDL module that implements the user's P4 program. Relative to hand-optimized RTL designs, the result produced by SDNet is generally within about 2x the logic and memory resource utilization. Additionally, SDNet generated designs can be configured to process packets at line rates between 1 and 400 Gb/s. SDNet also produces a SystemVerilog simulation testbench, C drivers to configure the PX tables, and an optional C++ model of the PX program to be used for debugging purposes. Recognizing the momentum behind P4, the next (2019) generation of SDNet provides a native P4 compiler, without the intermediate step via PX. This provides substantial improvements in packet processing pipeline latency, and in FPGA resource use.

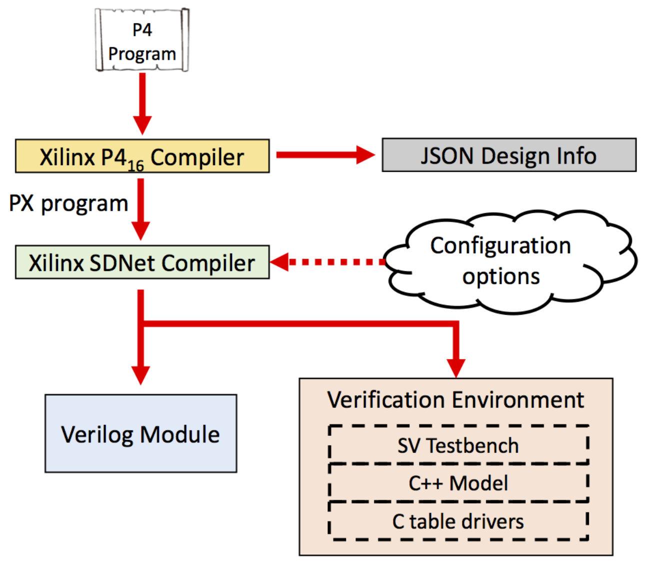

Figure 2: The Xilinx P4-SDNet compilation flow. P4 programs are first translated into a PX program, which is then compiled into a Verilog module using the SDNet flow. SDNet also produces a verification environment.

4 NETFPGA OVERVIEW

The NetFPGA project is a teaching and research tool, designed to allow packets to be processed at line-rate in programmable hardware. The NetFPGA project consists of four elements: boards, tools and reference designs, a community of developers, and contributed projects. The NetFPGA hardware family consists of three generations of FPGA-based networking boards; the latest is the SUME board [39] which has total I/O capacity of 100 Gb/s. All of the NetFPGA boards are designed with a PCIe connector so that networking software running on a host machine is able to interact with the FPGA accelerated packet processing logic. All of the code and documentation is openly hosted on GitHub [22].

Figure 3 depicts a block diagram of the canonical NetFPGA reference design. A similar design is used for NICs, switches, and IPv4 routers. It consists of four 10G SFP+ input/output ports along with one DMA interface for the CPU path. The NetFPGA data path consists of three main components: Input Arbiter, Output Port Lookup, and Output Queues. The Input Arbiter admits packets from the ports into the data path, towards the Output Port Lookup Module, where the main packet processing occurs and an output port is selected. The Output Queues buffer packets while they wait to be sent to the outputs. The core data path uses a 256-bit wide bus and runs at 200 MHz, fast enough to support an aggregate of 40 Gb/s from all four SFP+ ports.

NetFPGA has been used in classrooms for about 15 years with over 2,000 boards deployed. However, it has always required students to program in Verilog or VHDL, placing it off limits to many. While there are many students interested in learning about networked systems, relatively few have the necessary prerequisite knowledge in both hardware design and networking. Similarly, networking researchers wishing to prototype their ideas in hardware have needed to learn Verilog or VHDL.

To bridge this gap, the P4→NetFPGA workflow was created, with the goal of making it much easier for networking students and researchers to process packets in hardware. By allowing students to program NetFPGA using P4, instructors can give their students hands-on experience working with real hardware, while allowing them to focus on learning networking concepts rather than the minutiae involved in FPGA design. Similarly, networking researchers can rapidly prototype new systems without being bogged down in hardware development.

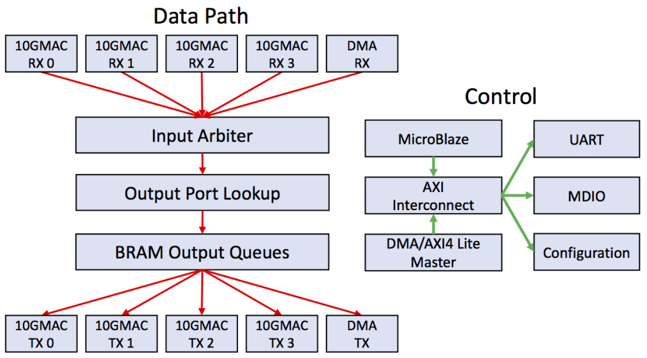

Figure 3: A block diagram of the NetFPGA reference design.

5 P4→NETFPGA WORKFLOW OVERVIEW

P4 designs for the NetFPGA SUME board are based on the *SimpleSumeSwitch* architecture, shown in Figure 4. The architecture is quite simple, consisting of a parser, a match-action pipeline, and a deparser. It is a good starting point for new P4 developers because it is simple and easy to understand, yet flexible enough to implement many different networking protocols and algorithms. It is less comprehensive (and we think much easier to understand for novices) than the standard P4 Portable Switch Architecture (PSA) [29], which is more suitable for full-featured commercial switches.

Table 2 describes the format of the SimpleSumeSwitch's `sume_metadata` bus and the functionality of each field. The architecture could also be extended by more advanced users, or, for the most adventurous users, it could be completely replaced by writing a new architectural model.

Figure 5 outlines the automated P4→NetFPGA workflow. The user writes a P4 program which is compiled (by Xilinx P4-SDNet) into an HDL instance of the SimpleSumeSwitch architecture. The SimpleSumeSwitch module is then automatically integrated into the

Table 2: Description of the SimpleSumeSwitch `sume_metadata` fields.

Field Name	Size (bits)	Description
`pkt_len`	16	Size of the packet in bytes (not including the Ethernet preamble or FCS)
`src_port`	8	Port on which the packet arrived (one-hot encoded)
`dst_port`	8	Set by the P4 program - which port(s) the packet should be sent out of (one-hot encoded)
`send_dig_to_cpu`	8	Set the least significant bit of this field to send the `digest_data` to the CPU
`*_q_size`	16	Size of each output queue at P4 processing start time, measured in of 32-byte words

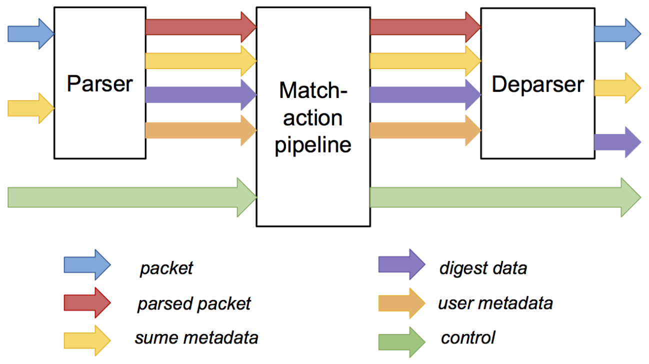

Figure 4: Block diagram of the SimpleSumeSwitch P4 architecture used within the P4→NetFPGA workflow.

NetFPGA Reference Switch design by replacing the default output port lookup module. Here is the basic set of steps used within the workflow:

(1) Write P4 program
(2) Implement custom extern modules (optional)
(3) Write python script to generate test data for simulations
(4) Run HDL simulations
(5) Build bitstream for FPGA
(6) Test the design on hardware

Ideally, users focus all their effort on steps 1-3, while all other steps are automated to eliminate the need for users to work with HDL.

The remainder of this section describes various aspects of the P4→NetFPGA workflow in more detail.

5.1 Extern Function Library

The core P4 language is designed to let users express stateless packet processing. Extern functions can be used for functions that cannot be described in P4, including stateful functions. Extern functions are implemented in HDL and the P4 program just sees the inputs and outputs, as parameters and results. In order to abstract away all HDL details from the programmer, P4→NetFPGA provides a library of commonly-used extern functions, shown in Table 3.

The supported extern functions in Table 3 are divided into two categories: stateless and stateful (inspired by Domino atoms [35]). To guarantee consistency between successive packets, stateful operations cannot be pipelined; each performs an atomic read-modify-write operation per packet. Extern functions can be combined to implement a wide variety of common, more complex algorithms.

For users who are experienced with HDL, the P4→NetFPGA workflow makes it very easy to create new extern functions. First, the user writes the extern module in HDL, then adds a few lines to one configuration file.

5.2 Simulation Environment

It is generally easier to debug program behavior in simulation than in hardware, and so the workflow helps users write and run simulations of their P4 programs.

First, developers can write a script consisting of a set of commands to populate match-action table entries. These entries will be automatically added to the P4 tables at the start of each simulation.

Second, developers can generate test packets (& metadata) using a Python Scapy module [3], along with the corresponding expected output packets & metadata.

Once the packets and metadata have been produced, users run two stages of simulation. The first stage is to run the testbench produced by the SDNet compiler. This will “send” the user defined input packets and metadata to the SimpleSumeSwitch HDL module and then compare the outputs with the expected outputs. Once verification is complete, the user runs a command to install the SimpleSumeSwitch HDL module as a NetFPGA IP core. The second stage of simulations uses the same stimuli and comparisons to verify that the SimpleSumeSwitch module was successfully integrated into the NetFPGA reference design.

After all simulations indicate that the P4 program is behaving correctly, the user runs one command to build the FPGA bitstream and can then start testing the design on hardware.

5.3 Runtime Control

When our switch is up and running, we need to control its behavior from a control plane. To this end, the P4→NetFPGA workflow generates a set of program-specific Python API control functions. The API functions let the user add/remove table entries and read/write stateful externs. The control software is thus able to dynamically influence how the FPGA processes packets without changing the hardware design. The Python API functions are wrappers around the C table drivers generated by SDNet.

One of the most useful features for debugging is an interactive CLI (command line interface), automatically generated by the workflow. The CLI lets the user interact with the P4 program at runtime and query compile time information about the program.

6 P4→NETFPGA IN PRACTICE

For the vast majority of new FPGA developers, learning to write P4 programs is significantly easier than learning to write HDL modules.

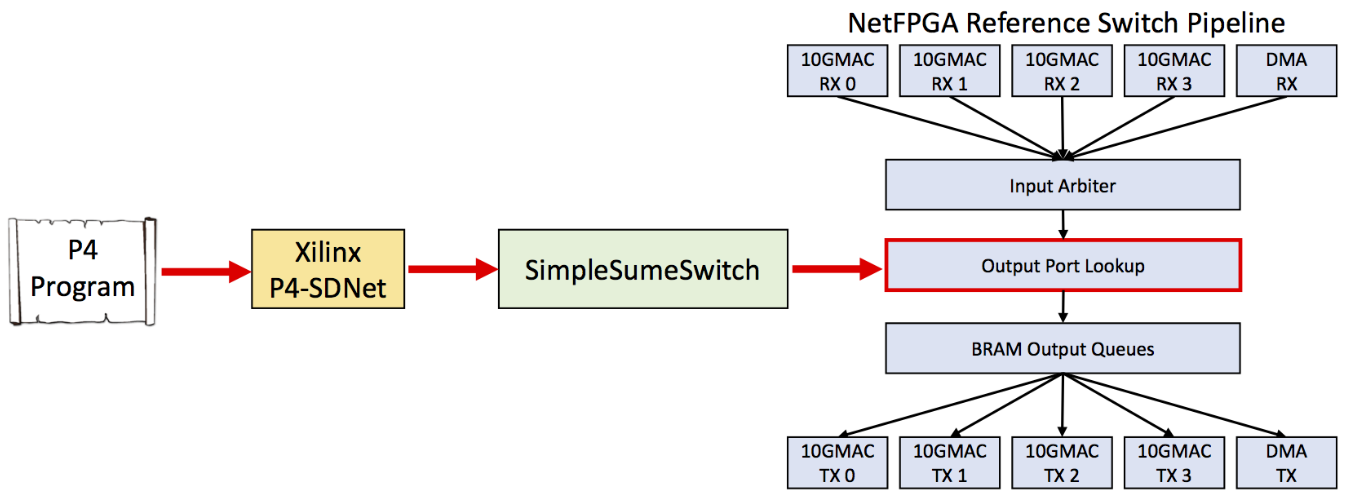

Figure 5: The automated P4→NetFPGA compilation flow. P4 programs are compiled into an HDL instance of the SimpleSumeSwitch architecture, which is then used to replace the output port lookup module in the NetFPGA Reference Switch design.

Table 3: The P4→NetFPGA extern function library.

Stateful Atomic Extern Functions	
Name	**Description**
RW	Read or write state
RAW	Read, add to, or overwrite state
PRAW	Either perform RAW or do not perform RAW based on predicate
ifElseRAW	Two RAWs, one each for when a predicate is true or false
Sub	IfElseRAW with support for subtraction as well
Stateless Extern Functions	
Name	**Description**
IP Checksum	Given an IP header, compute the IP checksum
LRC	Longitudinal redundancy check, simple hash function
timestamp	Generate timestamp (measured in clock cycles, granularity of 5ns)

Table 4 compares the number of lines of code (LOC) required to implement an Ethernet learning switch and an IPv4 router using both Verilog and P4. P4 programs are drastically more concise, largely because P4 exposes the abstractions that are needed to implement these types of applications. If we allow program size to serve as an indicator of code complexity then P4 programs are clearly easier to reason about. As a result, the P4→NetFPGA workflow allows researchers to rapidly implement research prototypes, and instructors to quickly teach students networked-systems design using real hardware.

Table 4: P4 vs Verilog lines of code comparisons.

	Verilog LOC	**P4 LOC**
Ethernet Learning Switch	1213	82
IPv4 Router	3889	266

6.1 Research Applications

The P4→NetFPGA research community is already thriving and has demonstrated that the workflow can be used to develop interesting research vehicles. This section will describe some of these applications.

In-band network telemetry (INT). This has been coined as the killer application for programmable data planes. INT [19] concerns gaining more visibility into the network, as packets traversing it collect programmable information. It provides answers to questions such as: Which path did my packet take through the network? Which rules did my packet take to get where it is now? How long did my packet queue at each switch? Who did my packet share a queue with? It can answer these questions without adding any additional packets into the network. For decades, network operators have been using limited tools like ping, traceroute, and SNMP to debug their networks. The amount of visibility provided by INT is a leap forward and is of great interest to network operators. An example implementation of the INT protocol can be found in the P4→NetFPGA repository [15].

Distributed Proactive Congestion Control. Most Internet congestion control algorithms are *reactive* in the sense that they first measure congestion signals from the network, such as packet loss or queue size, and then *reactively* adjust flow rates. *Proactive* algorithms, on the other hand, use explicit information about the flows and network topology to directly compute optimal flow rates *before* congestion occurs. The P4→NetFPGA workflow was used to develop a prototype proactive algorithm based on [17] in which the network switches exchange messages with each other and compute the optimal flow rates in a distributed fashion.

Stateless load-aware load balancing. Load balancing connections across many replicated instances of the same application is a very important task in modern data centers. The challenging aspect of building one of these load balancers is that all packets belonging to a particular connection must always be passed to the *same* application instance — a property called *per connection consistency* (PCC). Typically, there are hundreds or even thousands of servers dedicated to performing this sort of load balancing in a data center. SilkRoad [21] proposed implementing stateful load balancing in modern programmable switch devices hence eliminating the need for an army of software load balancers, thus saving cost. More recently, Benoit et. al. proposed SHELL [31] — a stateless application-aware load balancer prototyped using P4→NetFPGA that is able to closely approximate PCC. By making the load balancer implementation stateless, the design becomes much more scalable and practical to implement on hardware devices with limited memory.

Programmable Data Plane Verification. As programmable network devices become more common the reality is that network design bugs will become more common as well. The need is to reliably debug and validate the functionality of programmable designs running on hardware targets. NetDebug [6] proposes a P4-based programmable architecture for testing and validating network designs, running at line rate. NetDebug has a prototype implemented on P4→NetFPGA and has demonstrated exposing functional, performance and compiler bugs.

Network Accelerated Consensus. Consensus protocols are not typically the first thing to come to mind when thinking about networking applications. However, P4xos [10] suggests that by leveraging new P4 programmable network devices, significant performance improvements can be obtained over the traditional end host only implementations. Huynh et. al. show that in-network consensus acceleration is also useful for making storage class memory (SCM) fault tolerant [11]. Both of these works used P4→NetFPGA prototypes to demonstrate the practicality of the approach.

In addition to the examples described above, there have been many other designs prototyped using the workflow. These designs range from simple network functions, through in-network computing to games. For example:

- Hardware-accelerated firewall for 5G mobile networks.
- Named data networking with programmable switches.
- Heavy hitter detection.
- Network-accelerated sorting.
- In-Network key-value cache.
- In-Network compression.
- IP packet fuzzer.
- A game of tic-tac-toe, with a switch opponent.

6.2 Teaching Contributions

P4's ease-of-use makes it a good vehicle for teaching networking concepts. The P4→NetFPGA workflow was adopted by members of the P4 Consortium's Education Workgroup [9], and is being used for teaching in world-leading universities. For example, it was used for teaching how to build a fully functioning Internet router on the NetFPGA SUME board [13], replacing a Verilog-based course. The students implemented the routing protocol in software on the Linux host and the FPGA packet forwarding logic in P4. Only 25% of the students had prior experience with Verilog and none had prior experience with P4. Within six weeks of the course, all the students were able to build, in pairs, a functioning IPv4 router. The students demonstrated interoperability with other routers, building a small topology and testing various failure conditions. By transitioning from Verilog to the P4→NetFPGA workflow, the learning curve was dramatically reduced, the pace of the class was accelerated, and it was opened up to students with a wider variety of backgrounds.

7 FUTURE WORK

We are supporting, and continuing to evolve, the P4→NetFPGA workflow. With this in mind, there are a number of aspects that can be improved including the runtime control interface and support for user-defined target architectures.

Looking forward, we plan to add support for P4Runtime [27], which is quickly becoming the standard way to control P4 programs at run time. Supporting P4Runtime will enable P4→NetFPGA developers to easily write either a local or remote control plane and will simplify the task of integrating designs into systems that already support P4Runtime, such as ONOS [2].

In contrast to P4-programmable ASICs, FPGAs are flexible enough to support arbitrary packet processing architectures. In an effort to take advantage of this flexibility we plan to add support for users to be able to define custom P4 architectures rather than being constrained to use the SimpleSumeSwitch. This will allow developers to create more expressive P4 programs as well as experiment with their own custom modules within the architecture.

8 RELATED WORK

High level languages such as Vivado HLS and OpenCL are increasingly used to compile C-style programs onto FPGAs. Despite their popularity, they are normally used as a productivity tool by hardware engineers rather than by networking or software engineers. The flow of these languages was originally intended to target compute acceleration rather than packet processing. In principle, the C language is more expressive than the domain-specific P4, and could potentially be used for networking purposes. However, at the moment, it does not expose convenient abstractions or libraries that allow network developers to rapidly prototype new designs in hardware.

There have been two published attempts to build P4 compilers that target Vivado HLS [18, 32]. Similarly, P4FPGA [37] compiles

P4 programs into Bluespec [26], another high level synthesis language. Unlike P4→NetFPGA, these compilers require the user to actively go through an intermediate language, and provide a compiler only, rather than a full workflow. Furthermore, these projects lack community support or have since been deprecated.

Benacek et. al. developed a P4 parser compiler [1] that does not utilize an intermediate high level language. This compiler and workflow are completely closed source, and support only specific platforms, and hence are difficult to use for academic research and teaching.

Emu [36] is an extension to the Kiwi [34] compiler, which provides a way to compile .NET programs into RTL. Emu provides a standard library that can be used to build network systems. Unlike P4, .NET programs have not been widely adopted within the networking community as a way to express packet processing. Additionally, Emu programs are not inherently pipelined and hence do not guarantee line rate performance.

9 GETTING STARTED

The global P4→NetFPGA community consists of over 200 members from both industry and academia and it continues to grow. See the GitHub documentation [14] to find out more and learn how to get started. The documentation includes a set of on-line tutorials with step-by-step instructions that walk through the process of compiling P4 programs, running simulations, building the bitstream, and testing on hardware [15]. Community members are encouraged to contribute in any number of ways including, but not limited to:

- New P4 projects
- Extern function implementations
- Bug fixes or improvements
- Performance analysis tools and benchmarks
- Improved documentation

10 CONCLUSION

As P4 adoption continues to grow, more and more researchers seek out P4-programmable targets. Unfortunately, their choices have been mostly limited to either very low-end software emulation tools or very high-end (and expensive) P4 programmable ASICs. We developed the P4→NetFPGA workflow with the goal of bridging this gap. The community has already demonstrated that it is indeed possible to build low-cost, high-performance, P4 prototypes on real hardware using this workflow.

11 ACKNOWLEDGEMENTS

We thank the many people who have contributed to the NetFPGA project over the years. In particular, we acknowledge Andrew W. Moore and the NetFPGA development team at the University of Cambridge, whose efforts have enabled us to build the P4→NetFPGA workflow using NetFPGA SUME.

We also thank Robert Halstead and Chris Neely for their vital roles in developing the Xilinx P4-SDNet compiler, and in helping with the development of this workflow. Furthermore, we acknowledge the Xilinx University Program (XUP) for its continued support of the NetFPGA project.

This project was partially funded by the the Leverhulme Trust (ECF-2016-289) and the Isaac Newton Trust.

REFERENCES

[1] Pavel Benácek, Viktor Pu, and Hana Kubátová. 2016. P4-to-vhdl: Automatic generation of 100 gbps packet parsers. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 148–155.

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 1–6.

[3] Philippe Biondi. 2018. Scapy. https://scapy.net/

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[5] Gordon Brebner and Weirong Jiang. 2014. High-speed packet processing using reconfigurable computing. *IEEE Micro* 34, 1 (2014), 8–18.

[6] Pietro Bressana, Noa Zilberman, and Robert Soulé. 2018. A Programmable Framework for Validating Data Planes. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 1–3.

[7] Broadcom. 2018. Jerico2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690

[8] Cavium. 2018. XPliant Ethernet Switch Product Family. https://cavium.com/xpliant-ethernet-switch-xp60-and-xp70-family.html

[9] P4 Language Consortium. 2018. *Education Workgroup*. Repository, https://github.com/p4lang/education/wiki.

[10] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, H Weatherspoon, M Canini, N Zilberman, F Pedone, and R Soulé. 2018. *P4xos: Consensus as a Network Service*. Technical Report. Research Report 2018-01, USI.

[11] Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. 2018. Consensus for Non-Volatile Main Memory. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 406–411.

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA*.

[13] Stephen Ibanez. 2018. CS344 - Build an Internet Router. https://build-a-router-instructors.github.io/

[14] Stephen Ibanez. 2018. P4->NetFPGA Workflow. https://github.com/NetFPGA/P4-NetFPGA-public/wiki

[15] Stephen Ibanez. 2018. Tutorial Assignments. https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Tutorial-Assignments

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[17] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. 2015. High speed networks need proactive congestion control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM, 14.

[18] Jehandad Khan and Peter Athanas. 2017. Creating Custom Network Packet Processing Pipelines on HMC-Enabled FPGAs. In *Proceedings of the third workshop on Networking and Programming Languages (NetPL)*.

[19] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.

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

[21] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[22] NetFPGA.org. 2018. NetFPGA. https://netfpga.org/

[23] Netronome. 2018. About Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/

[24] Netronome. 2018. P4 Introduction. https://www.netronome.com/technology/p4/

[25] Barefoot Networks. 2018. Tofino. https://www.barefootnetworks.com/products/brief-tofino/

[26] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 69–70.

[27] P4.org. 2018. Announcing P4Runtime. https://p4.org/api/announcing-p4runtime-a-contribution-by-the-p4-api-working-group.html

[28] P4.org. 2018. behavioral-model. https://github.com/p4lang/behavioral-model

[29] P4.org. 2018. Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html

[30] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch.. In *NSDI*, Vol. 15. 117–130.

[31] Benoît Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas Clausen. 2018. Stateless Load-Aware Load Balancing in P4. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 418–423.

[32] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Langlois. 2018. P4-compatible High-level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 147–152.

[33] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 525–538.

[34] Satnam Singh and David J Greaves. 2008. Kiwi: Synthesis of FPGA circuits from parallel programs. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*. IEEE, 3–12.

[35] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 15–28.

[36] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, et al. 2017. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 459–471.

[37] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. ACM, 122–135.

[38] Xilinx. 2018. SDNet. https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

[39] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE micro* 34, 5 (2014), 32–41.