DEVELOPING AND USING P4 AND P4FPGA

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Dhruv Singhal

December 2017

ABSTRACT

P4FPGA is a toolkit developed at Cornell University that allows network switch programmers to compile and test P4 programs on a variety of FPGA-based networking devices. The toolkit is open-source and the original P4FPGA paper provided a technical overview of the toolkit; this paper describes the more pragmatic aspects of the actual C++ program that constitutes the P4FPGA compiler as well presents guidance on how to use the P4FPGA compiler.

ACKNOWLEDGEMENTS

The present work is based on the programming toolkit, software, published work, and thesis developed by Dr. Han Wang; Han was instrumental in giving me both an overview and a deep dive into the code-base for this project as well as with help in setting up the toolkit.

I am just as grateful for all the help and guidance I received from Professor Hakim Weatherspoon, who is my primary adviser for this project and the main faculty supervising the P4FPGA project.

Jingbo Wang has been a close collaborator in this project. She was responsible for the work on the P4 reference Behavioral Model compiler and upgrading Whippersnapper Benchmark suite to use P4-16.

Vishal Shrivastav was very helpful in providing me with assistance with Sysadmin work, notably getting access to computational resources, licenses, and hardware for the project.

P4FPGA project itself has a few other collaborator as well as makes use of a few open source projects. I have personally interacted with and received guidance from Hyun Tu Dang, the author of the Whippersnapper Benchmark suite, and Jamey Hicks of the Connectal project. Among those I did not get a chance to interact with in connection with this project are Robert Soule and Ki Suh Lee; nonetheless, their contribution to the project is duly acknowledged.

Finally, I am very grateful to Professor Nate Foster, my Master of Science committee chair, and Professor Kevin Tang, my advisor for my minor in Electrical and Computer Engineering, for their guidance throughout my enrollment in the Masters program at Cornell as well as for valuable input on this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Overview

P4FPGA is a software toolkit that allows enables execution of packet processing programs written in the P4 language on a variety of FPGA-based networking devices. The toolkit is open-source and the original P4FPGA paper [1] provides a technical overview of the toolkit. Using P4FPGA requires a considerable background in computer networking, software defined networking, operating systems and reconfigurable hardware (i.e. FPGAs). This document provides a context behind the development of the P4 language, P4FPGA, and the Whippersnapper benchmark suite, as well as to introduce the more pragmatic aspects of using and developing the P4FPGA compiler. Also presented in this document is a brief proposal for improving and standardizing the collection of timestamps for benchmarks in P4-enabled devices that was developed while analyzing the work on Whippersnapper benchmark suite.

1.2 Historical Background

Traditional networking hardware has been quite rigid in that the hardware was designed to perform only a fixed set of functions; the protocols and interfaces it could use were set in stone by their designers, and once deployed, provided very few ways for a network administrator to adapt its functionality to changing needs and demands. Moreover, experimentation with new protocols and

technologies meant that those concept has to be defined and refined all the way down to logic circuits (i.e. actual hardware), before any data could be collected on their performance for analysis and comparison, leading to high costs of research and development. These limitations lead to the birth and popularization of Software Defined Networking (SDN, [2] provides a historical overview).

SDN is characterized by computer networking systems that is programmable or reconfigurable (to varying degrees) even after deployment, as opposed to being fixed function like the traditional networking hardware. An SDN separates the *control* and *data* planes. The control plane is responsible for defining the overall behavior of the network, and it typically manages multiple forwarding devices. The data plane, on the other hand, just forwards individual network packets it receives according to a set of rules dictated by the control plane, at a very fast rate. The control plane typically reacts to changes in the network, for example, addition, remover or reorganization of devices, exceptional activity on the network, and so on. The data plane handles the "regular" functioning of the network. The control plane, unlike the data plane, can perform general computations on the packet contents, but is several orders of magnitude slower than the data plane.

The control plane shapes the behavior of the data plane by filling in the *forwarding tables* of individual packet processors. Forwarding tables are a simple data structure that tell the packet processor how to process the packet. Managing the forwarding table requires a well-defined interface for the controller to program the packet processors. In the early days of SDN, this was done in proprietary ways by SDN component vendors - akin to how each operating system offers a different API, even though most operating systems perform a very

similar set of functions.

As SDN became prominent, OpenFlow [3] was proposed as a standard interfacing layer between the controllers and the packet processors. OpenFlow provides a set of primitives for packet processing, namely, *header field* - which the incoming packets are matched against, *counters* - which maintain some state for analysis, and *actions* - which define the fate of the packet. In addition to that, OpenFlow defines a programmatic interface for controllers to install these primitives on packet processors.

OpenFlow has become the *de facto* universal standard for SDN interfaces, but it has its limitation. Notably, OpenFlow supports a fixed set of protocols and has only a limited support for extending that functionality. This is where P4 comes in. P4 stands for **P**rogramming **P**rotocol-independent **P**acket **P**rocessors. The P4 language is a general, extensible, domain-specific language that can define the behavior of a wide range of packet processing systems - physical or virtual. P4 programs can be written by hand, but can also be generated by an SDN controller. The data-plane runtime is responsible for compiling and/or interpreting the P4 program and configuring the packet processor accordingly.

Since P4 is fairly new, not a lot of systems exist that can make use of P4 programs directly. In particular, at the time of inception of P4FPGA, there were no open platform that allowed one to experiment with P4 on any of the highly flexible FPGA-based networking boards. FPGA-networking systems differ from regular Network Interface Cards (NICs) in that they can be reprogrammed (a characteristic of FPGAs - Field Programmable Gate Arrays) to support a very wide range of characteristics, right from the level of physical wire protocol to application layer processing. Thus, Wang et al [1] created the P4FPGA toolkit to

enable compilation of P4 to a variety of hardware and software targets.

1.3 The Author's Contribution

Since P4 is presented as more of a *revolution* than just *evolution*, meaning that it does things that are either not possible or hard to do in traditional systems, the P4 community needs a way to showcase the performance of P4-enabled devices. Moreover, consumers evaluating P4 hardware need a way to compare the various individual target devices, as well as the more general categories of devices. Thus *benchmarking* is an important are of focus for the p4 community.The Whippersnapper benchmark suite [4] was developed for such needs. However, the authors of that suite realized that there was no standard and uniform way of obtaining timestamps within the P4 pipeline to compute time interval, and, thus, latencies; one had to resort to non-standard, makeshift techniques to extract that information.

Therefore, the **author's contribution** in this thesis is to propose a set of key points at which those implementing P4 support in their switches will have to add facility to collect timestamps and make them accessible to P4 programs via standard API calls. The timestamps would have a bounded precision, and so it would be possible to compare timestamps and, more importantly, the differences between timestamps, to obtains latencies in the pipeline with bounded precision.

CHAPTER 2

THE P4 LANGUAGE

This chapter elaborates on the brief introduction to the P4 language presented in the previous chapter and discusses the structure and interpretation of P4 programs [1].

2.1 Introduction

P4 stands for **P**rogramming **P**rotocol-independent **P**acket **P**rocessors. P4 is a language for expressing how packets are processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance. While P4 was initially designed for programming switches, its scope has been broadened to cover a large variety of devices. It should be noted, however, that P4 is designed to specify only the data plane functionality of the target.

2.2 Comparison With a Traditional Switch

A P4-programmable switch differs from a traditional switch in two essential ways [5]:

1. The data plane functionality is not fixed in advance but is defined by the a P4 program. The data plane is configured at initialization time to implement the functionality described by the P4 program and has no built-in

[1] Please note that this chapter is essentially an abbreviation and paraphrasing of [5].

knowledge of existing network protocols.

2. The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

2.3 Advantages Over Traditional Systems

P4 provides a number of significant advantages compared to standard packet processing systems:

- **Flexibility:** P4 makes many packet-forwarding policies expressible as programs, in contrast to traditional switches, which expose fixed-function forwarding engines to their users.
- **Expressiveness:** P4 can express sophisticated, hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures (assuming sufficient resources are available).
- **Resource mapping and management:** P4 programs describe storage resources abstractly (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.

- **Software engineering:** P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- **Component libraries:** Component libraries supplied by manufacturers can be used to wrap hardware-specific functions into portable high-level P4 constructs.
- **Decoupling hardware and software evolution:** Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.
- **Debugging:** Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

2.4 P4 Language Versions

The P4 language has two major versions in prevalence - P4-14, which is the original P4 language, and P4-16, which was released to the public in May 2017. P4-16 makes major changes to the language, making it syntactically more consistent, smaller, and more explicit. While P4FPGA supports both versions of the language, **this document focuses primarily on P4-16** since the author considers it to be more relevant in the future.

2.5 Components of a P4 Program

The core abstractions provided by the P4 language are [5]:

1. **Header** types describe the format (the set of fields and their sizes) of each header within a packet.
2. **Parsers** describe the permitted sequences of headers within received packets, how to identify those header sequences, and the headers and fields to extract from packets.
3. **Tables** associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.
4. **Actions** are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control-plane at runtime.
5. **Match-action** units perform the following sequence of operations:
   (a) Construct lookup keys from packet fields or computed metadata,
   (b) Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
   (c) Finally, execute the selected action.
6. **Control flow** expresses an imperative program that describes packet-processing on a target, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can also be performed using a control flow.
7. **Extern** objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4.

8. **User-defined metadata** are user-defined data structures associated with each packet.
9. **Intrinsic metadata** is provided by the architecture associated with each packete.g., the input port where a packet has been received.

2.6 Example P4 Program: A Simple Switch

Presented below is a very simple switching program in P4 adapted from section 5.1 of [5]. The comments explain the purpose of each statement. A switch was chosen as an example because it represents one of the most typical use-cases of P4 and because it is sufficiently complex to demonstrate a wide variety of features of the language.

```
// File "very_simple_switch_model.p4"
// Very Simple Switch P4 declaration
// core library needed for packet_in and packet_out definitions
# include <core.p4>
/* Various constants and structure declarations */
/* ports are represented using 4-bit values */
typedef bit<4> PortId;
/* only 8 ports are "real" */
const PortId REAL_PORT_COUNT = 4w8;  // number 8 in 4 bits
/* metadata accompanying an input packet */
struct InControl {
    PortId inputPort;
}
/* special input port values */
const PortId RECIRCULATE_IN_PORT = 0xD;
```

```
const PortId CPU_IN_PORT = 0xE;
/* metadata that must be computed for outgoing packets */
struct OutControl {
    PortId outputPort;
}
/* special output port values for outgoing packet */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Prototypes for all programmable blocks */
/**
 * Programmable parser.
 * @param <H> type of headers; defined by user
 * @param b input packet
 * @param parsedHeaders headers constructed by parser
 */
parser Parser<H>(packet_in b,
                 out H parsedHeaders);
/**
 * Match-action pipeline
 * @param <H> type of input and output headers
 * @param headers headers received from the parser
                  and sent to the deparser
 * @param parseError error that may have surfaced
                     during parsing
 * @param inCtrl information from architecture,
                 accompanying input packet
 * @param outCtrl information for architecture,
                  accompanying output packet
```

```
 */
control Pipe<H>(inout H headers,
                in error parseError,// parser error
                in InControl inCtrl,// input port
                out OutControl outCtrl); // output port
/**
 * VSS deparser.
 * @param <H> type of headers; defined by user
 * @param b output packet
 * @param outputHeaders headers for output packet
 */
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
/**
 * Top-level package declaration - must be
 * instantiated by user.
 * The arguments to the package indicate blocks that
 * must be instantiated by the user.
 * @param <H> user-defined type of the headers processed.
 */
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
// Architecture-specific objects that can be instantiated
// Checksum unit
extern Checksum16 {
    Checksum16(); // constructor
    void clear(); // prepare unit for computation
    void update<T>(in T data); // add data to checksum
```

```
    void remove<T>(in T data); // remove data from
                               // existing checksum
    bit<16> get(); // get the checksum for the data
                   // added since last clear
}
```

CHAPTER 3

THE P4FPGA TOOLKIT

3.1 Background

Field Programmable Gate Arrays (FPGAs) are electronic devices that perform functions very similar to regular silicon-based integrated circuits (ICs), but unlike ICs, FPGAs can be *reprogrammed*. They enable circuit designers to quickly test out the viability and characteristics of arbitrary logic circuits, without having to go through the process of fabricating an IC. And owing to the fact that FPGAs still are hardware components, not emulators, the performance of FPGAs is very much comparable to what one would except with fabricated ICs. Due to this characteristic, FPGAs are often used as "accelerators" for computations that require high performance.

FPGAs, when combined with network interfaces, allow for rapid prototyping of new wire protocols. In addition to that, when used as accelerators, network FPGA devices can be used to implement and test higher layer protocols. Since the P4 language is designed to express the behavior of a forwarding device, it serves as an ideal candidate for configuring a networking FPGA device. This is what P4FPGA is for.

3.2 Components of the P4FPGA System

P4FPGA itself consists of a compiler, written in C++, that transforms a P4 program into a Bluespec specification of a forwarding device logic.

3.2.1 P4C: Compiler Frontend and Midend

The P4 consortium provides P4C [6], which is composed of the following:

1. **Frontend:** A compiler frontend that transforms P4 programs into an Abstract Syntax Tree (AST) representation for use by other stages of the compiler. The frontend supports both P4-14 and P4-16, but the support for P4-14 is winding down.
2. **Midend libraries:** A set of libraries that can be used to perform common transformations, such as basic optimizations, on the AST produced by the frontend. A target compiler is free to use or not use the midend functionality.
3. **Sample Backends:** P4C currently provides two targets as examples for developers, namely, the Behavioral Model Version 2 (BMv2) and the Extended Berkeley Packet Filter (EBPF). BMv2 is a software switch emulator, while EBPF is a low-level programming language (similar to machine code) executed by a virtual machine that resides in the Linux kernel.
4. **Development Tools:** Apart from the actual libraries, P4C also has some supporting scripts and tools that simplify development with P4C. These include several CMake build files that are extensible and can be adapted to compile and use external modules.

P4FPGA relies on P4C for the frontend, some midend libraries, and the supporting build tools. It supplies its own backend that transforms the AST into a set of Bluespec files that specify the hardware structure that would implement the P4 program.

3.2.2 P4FPGA Compiler Backend [1]

As states previously, the P4PFGA compiler transforms the AST produced by the P4C frontend to a Bluespec program. The following text provides a brief tour of the main components of the compiler, written in C++.

The C++ portion of P4FPGA is made up of `cpp` files, which roughly correspond to "modules" - collections of related functions and classes.

- **`backend.cpp`**: This module defines the entry point into the P4FPGA backend. It translates the `IR::TopLevelBlock`, provided by the P4C frontend, to `FPGAProgram`, which is P4FPGA's internal representation of an entire Bluespec system that implements a certain P4 program. In terms of Bluespec modules, `backend.cpp` generates `Top.bsv`, which lays out the platform, runtime, pipeline structure, and the API - everything that is needed to *support* the P4 program in an FPGA environment (as opposed to implement the P4 program itself).
- **`program.cpp`**: This module is delegated the task of generating the Bluespec code that actually implements the P4 program fed into the compiler. It generates the `Program.bsv` module, which describes the program architecture and its interface with the runtime.
- **`pipeline.cpp`**: This module generates `Pipeline.bsv` based on the P4 architecture specification. The architecture specification defines the sequence of parsers, deparsers, and control blocks. Note that currently[2], P4FPGA assumes a V1 model, wherein the order of blocks is as follows:

[1] Adapted from and elaboration of `https://github.com/p4fpga/p4fpga#generated-bluespec-organization`

[2] July 2017

Parser → Ingress → Egress → Deparser

- **`control.cpp`**: This module generates `Control.bsv`, which contains that *ingress* and *Egress* logic. Ingress and egress implement the control flow for tables and actions.
- **`table.cpp`**: This module generates the implementation of a P4 table specification using either a Binary Content Addressable Memory (BCAM) or a Ternary Content Addressable Memory (TCAM).
- **`action.cpp`**: This module generates the implementation of a P4 action block. It implements a few basic arithmetic and logic operation optimizations using certain Digital Signal Processing (DSP) transformations. It uses Bluespec's built-in operators to implement boolean operations.
- **`channel.cpp`**: This module generates `Channel.bsv`, which defines the interfaces to various communication channels to interact with the controlleras well as the I/O ports.

3.2.3 The Connectal Control-Plane System

Connectal provides a hardware-software interface for applications split between user mode code and custom hardware in an FPGA. It can automatically build the software and hardware glue for a message based interface and also provides for configuring and using shared memory between applications and hardware. Communications between hardware and software are provided by a bidirectional flow of events and regions of memory shared between hardware and software.

3.3 Areas of Improvement to P4FPGA

1. **Code Generation**:

   Problem: The current method of code generation is quite brittle, and consists of simply concatenating strings to compose the output. As the P4 specification and the FPGA APIs rapidly evolve, maintaining this method of code generation becomes increasingly difficult.

   Solution: An effective way of maintaining the compiler would be to divide the current P4FPGA backend into a midend and a backend. The midend will transform the P4 intermediate representation into a different abstract representation that is closer to the Bluespec output. The final backend will convert this abstract representation into actual Bluespec code. This has a number of advantages as a side effect. A more abstract representation opens way to generation of code directly to Verilog or VHDL, or to any of the numerous prototyping languages like PyMTL. Furthermore, even if the project focuses solely on Bluespec, it insulates the users from breaking changes in the language specifications, transferring that responsibility to the final backend programmer.

2. **Software Quality & Necessity of Manual Inspection**:

   Problem: P4FPGA, currently[3] is quite unstable, due to the fact that P4C, which P4FPGA vitally depends upon, is evolving rapidly with breaking changes, thus, demanding a lot of developer time to simply maintain a status quo. P4FPGA, currently, does not work end-to-end: almost every major and minor stage of compilation requires manual inspection and fixing for all but the most trivial P4 program. One particular example is that

[3] As of July 2017

the symbol name generation in P4FPGA is inconsistent, and relies on unstable interfaces exported by P4C. This results in generation of Bluespec symbol names that are either invalid, or that are duplicate or incorrectly scoped.

Solution: P4FPGA needs to track a known a stable version of P4C to avoid having to deal with breaking changes. In addition to that, higher level improvements at the level of code structuring and compiler layout will certainly reduce the amount of hand-holding the compiler needs.

3. **Very Specific Operating Environment**:

   Problem: P4FPGA depends on a large number of external libraries and toolkits, all of which have their own graph of dependencies. While the project and its dependencies try hard to be explicit about their dependencies, very often there exist implicit dependencies on common things like the C++ compiler, the operating system kernel interfaces, and pre-installed programs. Due to its dependency on Connectal as well as due to the limitations of the targets that are supported by P4FPGA, the P4FPGA toolkit is currently only known to run on a very specific combination of hardware and software.

   Solution: The solution here is more of a software engineering research problem, but is stated nevertheless. Containerization, i.e. encapsulating programs in virtual machines, or lightweight containers (such as Docker and LXC) will reduce the conflict between library versions across different tools. As a bonus, it would make it easy to distribute the toolchain to a wider audience that otherwise lacks the expertise, time, or resources to set their environment up themselves.

CHAPTER 4

PERFORMANCE MEASUREMENT

This chapter provides an overview of benchmarking as it would apply to P4.

4.1 Overview

P4 is a relatively new language with a rapidly evolving ecosystem of targets and toolkits. Every new architecture, switching program, and compiler either has no way to test its performance or has a custom-designed set of performance tests that usually paint the system in a positive light. Even if the authors make a conscious effort to maintain objectivity, there is always a concern that one might not be measuring the right metric and thus potentially optimizing for something that doesn't matter to the community. Therefore, a set of objective, peer-reviewed benchmarks is essential for the success of the community as a whole. The Whippersnapper benchmark suite [4] is an attempt at creating one such suite. This chapter will be on the topic of uniform and easy measurement of time within a P4 pipeline - one of the issues that is briefly discussed in [4].

4.2 Uniform Timestamping

4.2.1 Background

The P4 language, in its present form, does not provide a standard API to access a periodic counter, or a timestamp, in a P4 program. The current way to implement the collection of timestamps at various points in the pipeline is to

use vendor-specific `extern` functions - which are black-box interfaces to non-standard hardware functions. This leads to the problem of non-uniformity in the precision of data collected across a variety of targets, since every target would potentially implement timestamping with a different degree of precision, as well as different exact locations in the pipelines.

4.2.2 Proposal: Standard API

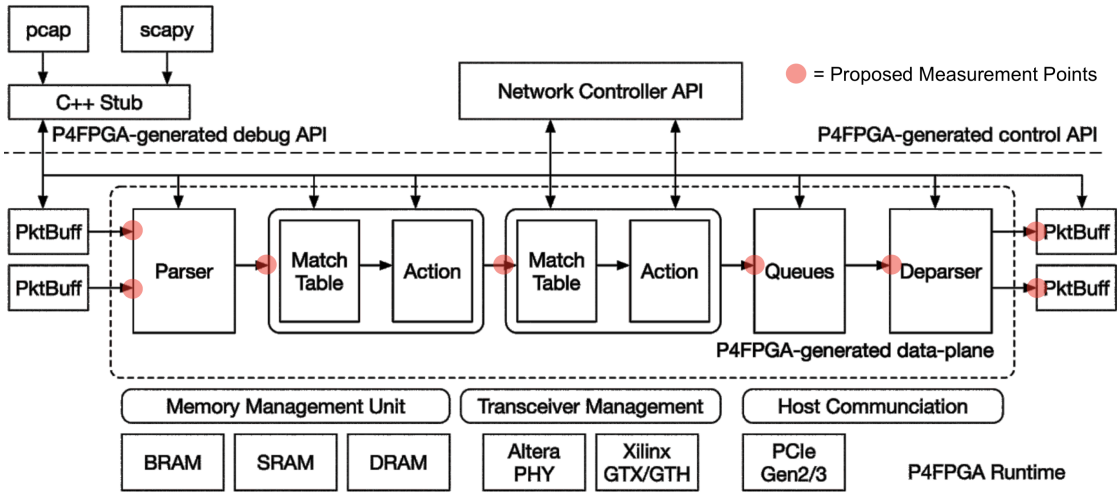
The author of this thesis, and collaborators, presented in [7] one approach to resolving the issue discussed above. The proposal is to three-fold:

- Define the **exact points** in the pipeline at which a timestamp can be *potentially* collected. In the context of P4FPGA,
- Allow for **storage and retrieval** of the collected timestamp from the *previous* packet, and writing into the current packet.
- Define a **standard library function** that provides access to timestamp at a specified point. The function will define a certain bounded precision, which must be adhered to by all implementations.

The specifics of the collection points and the function prototype are not defined here since these will require deliberation and input from the P4 target vendors as well as users. But the idea is more or less concrete in that it requires a standard method of data collection.

The proposal is viewed by the author as a standard that would be widely adopted by many, if not most, targets; its formal induction into the declarations

Figure 4.1: Proposed collection points of the timestamps in the P4FPGA pipeline.



of the P4 consortium is not relevant to its success. In fact, the proposal would serve its purpose if and only if it becomes the *de facto* standard for collection of timing data; it would not be considered useful if it merely remains a standard (official or otherwise), without any implementer or users.

4.2.3 The Proposal in the Context of P4FPGA

Having defined the three portions of the proposal, it is now possible to discuss their implementation in the context of a concrete P4 implementation, such as the P4FPGA toolkit.

- The proposed specific points are labelled in Figure 4.1.
- The timestamps would be stored in the same memory and registers and the intrinsic packet metadata.

- A proposed API is in Figure 4.2.

Figure 4.2: A proposed API for timestamp collection.

```
extern Timestamp {
    Timestamp();              // constructor
    bit<64> getTimestamp00(); // First timestamp
    bit<64> getTimestamp01(); // Second timestamp
    // ... and so on, for as many collection points as there are.
}
```

4.2.4 Materializing the Proposal

The proposal discussed above is currently just a proposal that requires deliberation with vendors and implementation in terms of actual computer programs. Speaking specifically of having a proof-of-concept implementation for P4FPGA, this does not have a concrete implementation yet as it is, first, necessary for P4FPGA itself to work as a coherent compiler for usefully complex P4 programs. To someone interested in taking the work further, it is recommended to focus on the P4 Behavioral Model software switch for exploratory work, and only then attempt at a P4FPGA implementation.

4.2.5 Limitations of the Scope of the Proposal

The proposal has a few drawbacks, which are discussed briefly in the following text.

- **Strictly Intra-Pipeline**: The API proposed above is aimed strictly for measurements *within* within a single pipeline; it does not concern itself with measurement of a collection of packet processors working together in a network.
- **Potential Waste of Resources**: One might argue that production system might not need the timestamping functionality for regular use, and, thus, it would be a waste of hardware and software resources, as well as designer time, to factor in the timestamping API.
- **Disagreement in the Exact Interface**: Different groups may have different priorities with respect to defining and refining the exact protocol for timestamp collection, and, thus, may never reach a consensus.

CHAPTER 5

CONCLUSION AND FUTURE DIRECTION

P4 is a rapidly evolving platform, which is gaining mainstream acceptance by a number of software and hardware vendors, their customers, as well as academicians. Like any new technology, it has its rough edges, which are getting polished as time passes. P4FPGA is a project with a lot of untapped potential, since it is, even now, somewhat unique in its ability to generate firmware for targets by different vendors. The deep layering also uniquely gives it the ability to rapidly add support for more platforms. It, however, cannot be denied that P4FPGA needs a fair amount of development before it is ready for widespread use. The following are two projects that can further the development of P4FPGA, and, thus, increase its appeal.

- **Stabilizing the P4FPGA Compiler**: A prior chapter discussed the shortcomings of the P4FPGA toolkit, and had emphasized the fact that P4FPGA is fairly unstable, and unsuitable for regular use. Therefore, effort must be put into fixing the problems that arise as P4FPGA is used on increasingly complex P4 programs.
- **Explore/Develop Open-Source FPGA Architectures**: Mentioned previously, as well, is the fact that P4FPGA's reliance on proprietary systems complicates the workflow as well as places sometimes unfeasible requirements on potential users. Making use of open systems would alleviate this, and can be a potentially transforming change for the toolkit.

APPENDIX A

GLOSSARY OF TERMS

- **Architecture:** A set of P4-programmable components and the data plane interfaces between them.
- **Content Addressable Memory (CAM):** It is a hardware structure that stores data in an associative array (i.e. key-value pairs), and provides the ability to look up the value, given a particular key (also know as a "tag"). CAMs operate in base-2 (i.e. binary), like most of the computer hardware.
  - **Binary (BCAM):** These match the tag *exactly*.
  - **Ternary (TCAM):** These allow for "don't care" bits in the tags.
- **Control plane:** A class of algorithms and the corresponding input and output data that are concerned with the provisioning and configuration of the data plane.
- **Data plane:** A class of algorithms that describe transformations on packets by packet-processing systems.
- **Metadata:** Intermediate data generated during execution of a P4 program.
- **Packet:** A network packet is a formatted unit of data carried by a packet-switched network.
- **Packet header:** Formatted data at the beginning of a packet. A given packet may contain a sequence of packet headers representing different network protocols.
- **Packet payload:** Packet data that follows the packet headers.

- **Packet-processing system:** A data-processing system designed for processing network packets. In general, packet-processing systems implement control plane and data plane algorithms.
- **Target:** A packet-processing system capable of executing a P4 program.

BIBLIOGRAPHY

[1] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A rapid prototyping framework for P4," in *Proceedings of the Symposium on SDN Research*, pp. 122–135, ACM, 2017.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[3] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Thow-publishedetti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 1617–1634, July 2014.

[4] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 language benchmark suite," in *Proceedings of the Symposium on SDN Research*, pp. 95–101, ACM, 2017.

[5] The P4.org language consortium, *The P4-16 Language Specification*, May 2017.

[6] The P4.org language consortium, "P4C." `https://github.com/p4lang/p4c`.

[7] D. Singhal, J. Wang, and H. Weatherspoon, "Uniform timestamping in p4." P4 Workshop 2017, 2017.