P4FPGA : A Rapid Prototyping Framework for P4

Han Wang
Cornell University

Robert Soulé
Università della Svizzera italiana
Barefoot Networks

Huynh Tu Dang
Università della Svizzera italiana

Ki Suh Lee
Cornell University

Vishal Shrivastav
Cornell University

Nate Foster
Cornell University
Barefoot Networks

Hakim Weatherspoon
Cornell University

ABSTRACT

This paper presents P4FPGA, a new tool for developing and evaluating data plane applications. P4FPGA is an open-source compiler and runtime. The compiler extends the P4.org reference compiler with a custom backend that generates FPGA code. P4FPGA supports different architecture configurations, depending on the needs of the particular application.

We have benchmarked several representative P4 programs, and our experiments show that code generated by P4FPGA runs at line-rate at all packet sizes with latencies comparable to commercial ASICs. By combining high-level programming abstractions offered by P4 with a flexible and powerful hardware target, P4FPGA allows developers to rapidly prototype and deploy new data plane applications.

KEYWORDS

P4, FPGA, High-level synthesis

ACM Reference format:
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 ACM Symposium on SDN Research conference, Santa Clara, California USA, April 2017 (SOSR 2017),* 14 pages.
DOI: http://dx.doi.org/10.1145/3050220.3050234

1 INTRODUCTION

P4 [6] promises to have a profound impact on networking by making data planes programmable and enabling unprecedented levels of innovation. Programmers are already using the language's domain-specific abstractions to implement a variety of novel applications, including network diagnostics and telemetry tools [23, 38], advanced traffic engineering and load balancing systems [19], and even optimized consensus protocols [11, 25].

However, most current targets for P4 are implemented in software [32, 36]. To get the full performance benefits of a programmable data plane, developers need access to platforms that can execute their designs efficiently in hardware. In this respect, Field Programmable Gate Arrays (FPGAs) are an attractive target platform for P4 programs. As a form of re-programmable silicon, FPGAs offer the flexibility of software and the performance of hardware. Indeed, major cloud providers, such as Microsoft, Amazon and Baidu, already deploy FPGAs in their data centers to boost performance—e.g., to accelerate network encryption and decryption or implement custom transport layers [35].

There are several challenges in designing a compiler from P4 to FPGAs. First, FPGAs are typically programmed using low-level libraries that are not portable across devices. Moreover, communication between 3rd party processing elements is device-specific, adding an additional hurdle to portability. Second, generating an efficient implementation of a source P4 program is difficult since programs vary widely and architectures make different tradeoffs. Third, although the P4 language is target agnostic, it relies on a number of "extern" functions for critical functionality, such as checksums and encryption, complicating code generation.

This paper presents P4FPGA, an open-source P4-to-FPGA compiler and runtime that is designed to be flexible, efficient, and portable. To ensure that P4FPGA is flexible enough to implement many different network functions, the compiler allows users to incorporate arbitrary hardware modules written in the language of their choice. This approach offers a degree of flexibility that would be difficult to achieve on other targets, such as a switch ASIC. To ensure that the code generated by the compiler is efficient, P4FPGA supports datapaths with one [24] or more ports [17]. This approach allows users to

SOSR 2017, Santa Clara, California USA

DOI: http://dx.doi.org/10.1145/3050220.3050234

select the best design for their particular application. Finally, to ensure that programs are portable across different devices, P4FPGA provides a runtime with device-agnostic hardware abstractions. This runtime allows P4FPGA to support designs that can be synthesized to either Xilinx or Altera FPGAs.

We have evaluated our prototype implementation on a variety of representative P4 programs. Our experiments show that code generated by P4FPGA runs at line-rate throughput on all packet sizes up to MTU with latencies similar to off-the-shelf commodity switch ASICs. Moreover, P4FPGA is already being used by at least two research projects [11, 18] to deploy P4 programs on actual hardware.

Overall, this paper makes the following contributions:

- It presents the design of a P4-to-FPGA compiler and runtime system.
- It evaluates the performance of the generated code and backend on a variety of non-trivial P4 programs and demonstrates that performance is competitive with commercial switches—e.g., latencies are comparable to commercial cut-through switches.
- It develops a wide variety of standard and emerging network applications using P4FPGA, which demonstrates that the tool is broadly applicable.

The rest of this paper is organized as follows. We first provide background on the P4 language (§2). We then discuss the design of the code-generation (§3) and runtime (§4) components; and details of the implementation (§6). Next, we evaluate (§7) our prototype. Finally, we discuss related work (§8), and conclude (§9).

2 BACKGROUND AND OVERVIEW

Before describing the details of the P4FPGA design, we briefly present a high-level overview of the P4 language [6], and networking processing on FPGAs.

P4 Background. When describing a P4 compiler, it is important to clarify some terminology. A *target* is hardware that is capable of running a P4 program, such as FPGAs, ASICs, or CPUs. An *architecture* refers to the combination of the runtime as well as the processing pipeline specified in a P4 program. A P4 program is *target independent*, meaning the same program can be implemented on different hardware. A P4 compiler is responsible for mapping the abstract architecture specified by the P4 program to a particular target and architecture. FPGAs are uniquely able to map abstract architectures directly to hardware. In contrast, a single switch ASIC will only be able to faithfully implement a small subset of potential architectures and extern primitives.

As a language, P4 allows developers to specify how packets are processed in the data plane of network forwarding

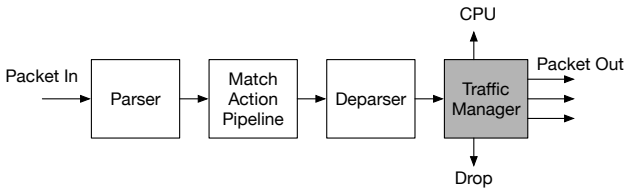

Figure 1: Example P4 Abstract Architecture

elements. P4 programs are written against an abstract architecture that hides the actual physical implementation. In the abstract architecture, packets are first parsed, and then processed by a sequence of match-action tables. Each table matches on specified packet header fields, and then performs a sequence of actions to modify, forward, or drop the packet. Additionally, the program collects packet metadata, such as ingress and egress port numbers, which flows through the pipeline. Implicitly, at the end of the pipeline, packets are reassembled for transmission in a deparser stage. Figure 1 illustrates a simple example with a single parser and match table pipeline. The P4 language provides syntax that mirrors this abstract model. For brevity, we do not describe the syntax in detail as the full language specification is available online [31]. We simply mention that programmers can declare packet headers, compose tables, and specify actions. Tables can be populated at runtime with flow rules via a control plane API.

The P4FPGA compiler supports both $P4_{14}$ and $P4_{16}$ syntax. The example code in Figure 2 shows a subset of a $P4_{14}$ program for counting UDP packets by destination port. Line 4 defines the layout for the UDP packet headers. Line 11 declares an instance of that header, named `udp`. Line 12 defines how to parse UDP packet headers. The `extract` keyword assigns values to the fields in the header instance. The `return` keyword returns the next parser stage, which could be `ingress`, indicating the start of the pipeline. Line 26 is the start of the flow control for the P4 program. It checks if the arriving packet is an Ethernet packet, then if it is an IPv4 packet, and finally if it is a UDP packet. If so, it passes the packet to the `table_count` table, defined on Line 23. The `table_count` table `reads` the destination port, and performs one of two possible actions: `count_c1` or `_drop` a packet. The action `count_c1` on Line 20 invokes a `count` function. The `count` function must be defined externally to the P4 program.

FPGA Background. FPGAs are widely used to implement network appliances [10, 16] and accelerators [3], as applications implemented on FPGAs typically achieve higher throughput, lower latency, and reduced power consumption compared to implementations with general-purpose CPUs.

Development on an FPGA typically involves using a low-level hardware description languages (i.e., Verilog, VHDL)

```
// We have elided eth and ipv4 headers,
// and the extern declarations for
    brevity

header_type udp_t {
   fields {
      srcPort : 16;
      dstPort : 16;
      length : 16;
      checksum : 16;
}}
header udp_t udp;
parser parse_udp {
   extract(udp);
   return ingress;
}
counter c1 {
   type: packet;
   numPackets : 32;
}
action count_c1() {
   count(c1, 1);
}
table table_count {
   reads { udp.dstPort : exact; }
   actions { count_c1; _drop; } }
control ingress {
   if (valid(eth)) {
     if (valid(ipv4)) {
        if (valid(udp)) {
          apply(table_count);
}}}}
```

Figure 2: Subset of a $P4_{14}$ program to count UDP packets.

to statically specify a hardware circuit for a single application [40]. However, these languages are widely regarded as difficult to use, and consequently, there has been significant research in high-level synthesis [2, 9, 37].

P4FPGA uses one of these languages, Bluespec System Verilog [27], both as the target for compiler generation and to implement the runtime. Bluespec is a strongly typed functional language, similar in many respect to Haskell. Users write hardware operations as guarded rules [27]. The language includes a large set of libraries for common hardware constructs such as registers, FIFO queues and state machines. Moreover, P4FPGA uses Bluespec code from the Connectal project [21] to implement the control plane channel. Using Bluespec simplifies FPGA development by providing high-level language constructs, and is more expressive than Verilog.

P4FPGA Overview. A compiler for P4 is responsible for two main tasks: generating the configuration to implement a data plane on a target platform at compile time and generating an application programming interface (API) to populate tables and other programmable elements at run time.

Figure 3 presents a high-level overview of the P4FPGA framework and compilation strategy. The components inside the dashed-line were developed specifically for P4FPGA. The components outside the dashed-line are existing open-source tools or commercial products for FPGA synthesis that are re-used by P4FPGA.

P4FPGA builds on the reference P4 compiler implementation provided by the P4 organization [34]. The reference compiler parses P4 source, and produces a standard intermediate representation (IR) [34]. We chose to build on the reference front end for practical reasons. It both reduces the required engineering effort, and ensures that FPGA conforms to the latest P4 syntax standards.

P4FPGA includes three main components: (i) a code generator, (ii) a runtime system, and (iii) optimizers implemented as IR-to-IR transformers. The code generator produces a packet-processing pipeline inspired by the model proposed by Bosshart et al. [7]. The runtime provides hardware-independent abstractions for basic functionality including memory management, transceiver management, host/control plane communication. Moreover, it specifies the layout of the packet processing pipeline (e.g. for full packet switching, or to support network function virtualization (NFV). The optimizers leveraging hardware parallelism to increase throughput and reduce latency.

The compiler produces Bluespec code as output, which is compiled to Verilog. The Verilog code is further synthesized by downstream FPGA tool chains. The output bitstream can then be used to configure an FPGA.

In the following sections, we present the code generator, runtime system, and optimizers in full detail.

3 CODE GENERATION

The core job of the P4FPGA compiler is to map logical packet-processing constructs expressed in P4 into physical packet-processing constructs expressed in a hardware description language. We organize the generated physical constructs into *basic blocks*. As in most standard compilers, a basic block is a sequence of instructions (e.g., table lookups, packet-manipulation primitives, etc.). We implement basic blocks in P4FPGA using parameterized templates. When instantiated, the templates are hardware modules that realize the logic for packet parsers, tables, actions, and deparsers.

There are two motivations behind our use of basic blocks. First, it reduces the complexity of the compiler, since code generation simply becomes the composition of modules that implement standard interfaces. Second, the modular design

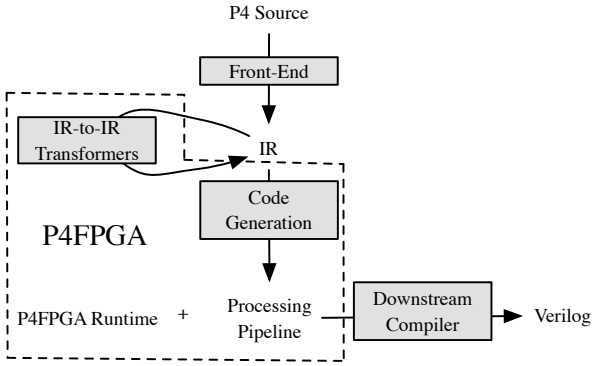

Figure 3: P4FPGA Framework Overview.

enables extensibility in two ways: (i) programmers can easily add externally-defined functionality via a foreign-function interface (e.g., to implement a custom hash function), and (ii) programmers can modify the compiler by replacing one basic block with another that implements the same functionality (e.g., to modify the memory storage to use DRAM, SRAM, or an SSD).

The control flow constructs from the P4 source program dictate the composition of the basic blocks. We refer to this composition of blocks as the *programmable packet-processing pipeline*. This is in contrast to the fixed-function pipeline that is realized by the P4FPGA runtime system. In other words, the programmable packet-processing pipeline is specified by the logic of a particular P4 source program, whereas the fixed-function pipeline is determined by the target platform, and is fixed for all input source programs.

3.1 Programmable Pipeline

The programmable packet-processing pipeline realizes the programmable logic of a P4 source program on an FPGA hardware implementation. It consists of a composition of basic blocks to parse, deparse, match, or perform an action.

Parsing. Parsing is the process of identifying headers and extracting relevant fields for processing by subsequent stages of the device. Abstractly, the process of parsing can be expressed as a finite state machine (FSM) comprising a set of states and transitions. From a given state, the FSM transitions to the next state based on a given input from a header or metadata. A subset of the states identifies and extracts header fields. The FSM graphs may be acyclic, as is the case with Ethernet/IPv4 parsers, or cyclic—e.g., to parse TCP options. P4FPGA adopts a streaming approach in which the packet byte stream is fed into the FSM and processed as soon as there is enough data to extract a header or execute a FSM transition.

The implementation of the parser basic block includes code that is common to all parser instances and generated code that is customized for each specific parser. The common code includes state variables (e.g., header buffer, parse state, and offset), and a circuit that manages incoming bytes. The generated portion of a parser implements the application-specific FSM.

Deparsing. As shown in Figure 3, there are two sources of input data to the deparser stage. One is the packet data stored in memory (§4), and one is the modified packet header processed by the programmable pipeline. The deparser reassembles the packet for transmission from these two input sources.

Like the parser, the deparser is implemented as a FSM. However, the design of deparser is more complicated, since it may add or remove headers during packet assembly.

The deparser consists of three modules: packet extender, packet merger, and packet compressor. The packet extender supports the addition of headers by inserting empty bytes at a designated offset. The packet merger writes modified packet fields, including fields added by the extender module. The packet compressor marks bytes to be removed by writing to a bit mask.

Note that the deparsing stage is responsible for modifying packets. Packet modification could be performed inline one-by-one (i.e., after every table), or all together at the end of the pipeline. P4FPGA takes the latter approach, since it reduces latency. In other words, the pipeline modifies a copy of the header, and changes are merged with the actual header in the deparser stage.

Matching. In P4FPGA, basic blocks for tables are implemented as FPGA hardware modules that support get/put operations via a streaming interface. P4 allows users to specify the algorithm used to match packets. Our P4FPGA prototype supports two matching algorithms: ternary and exact-match. The ternary match uses a third-party library. We implemented two versions of exact match ourselves, one using a fully-associative content addressable memory (CAM) [1], and the other using a hash-based lookup table. Users can choose the implementation strategy by using a command line option when invoking the compiler. Note that because of the "programming with tables" abstraction that P4 provides, some programs include tables without lookup keys, whose purpose is solely to trigger an action upon every packet processed by the pipeline stage. P4FPGA handles this corner case by not allocating any table resources to this stage.

Actions. P4 actions can modify a field value; remove/add a header; or modify packet metadata. Conceptually, each action operates on one packet at any given time, with all temporary variables stored in metadata on the target. P4FPGA performs

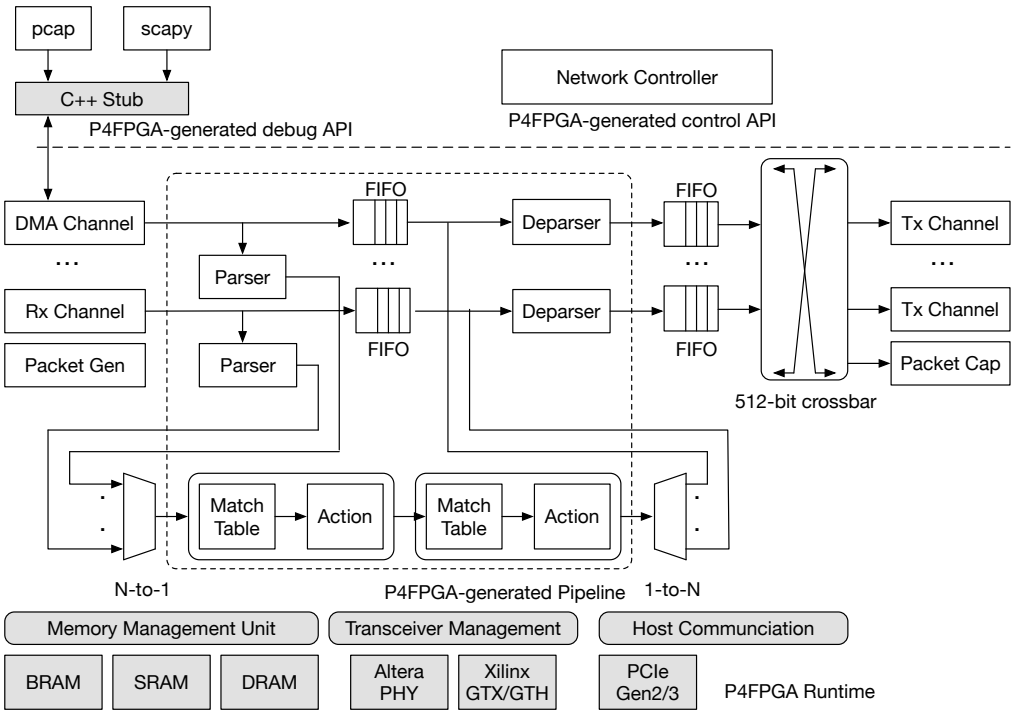

Figure 4: P4FPGA Runtime and Pipeline.

inline editing to packet metadata and post-pipeline editing to packet headers. Modify actions create a copy of the updated value stored in a memory that is merged with the original packet header field in the deparser block. For actions that change a packet header length, basic blocks are created before and after the deparser block, to perform re-alignment. For example, to remove a packet header, the deparser marks the header as invalid in a bit mask. The realignment block then shifts the remaining bytes forward to cover the gap created by the removed header.

Control Flow. P4 control flow constructs compose tables and actions into an acyclic graph. A naïve implementation would be to use a fixed pipeline. In such a design, the runtime would use extra metadata to implement the logic of the source program. However, because we target an FPGA, P4FPGA can map the control flow directly onto the generated hardware design. Each node corresponds to a basic block followed by a branch condition. We note that this is much more flexible than implementing control flow on an ASIC. During program execution, the parsed packet and metadata is passed along the tree structure. At each node, the runtime evaluates the conditional and passes the data to the next node along the appropriate branch, or performs a table lookup depending on the rules specified in the control plane API. P4FPGA relies on pipeline parallelism to achieve high throughput. In other words, at any given time, different nodes in the tree can process different packets in parallel.

3.2 Control Plane API

In addition to generating code that implements the application-specific logic in the data plane, P4FPGA also generates a control plane API that exposes a set of C++ functions that allow users to insert/delete/modify match table entries and read/write stateful memory. Moreover, the generated interface includes functions to support debugging. Users can inject packets via a packet capture (`PCAP`) trace, or can enable/disable an on-chip packet generator and capturer.

3.3 External Functions

One of the advantages of FPGAs over ASICs are that they are more flexible and programmable. P4 offers a relatively restrictive programming interface that is targeted for network applications and is platform-agnostic by design. Therefore, it is sometimes necessary to execute additional functionality via externally defined functions. An externally defined function could be used to access a state in a register or to execute custom logic, such as a checksum computation or encryption/decryption. In P4, these are declared using `extern` blocks, and the implementations of these calls are target specific.

P4FPGA allows users to implement externally defined functions in the hardware description language of their choice. However, such functions pose a challenge for efficient code generation, since they may have high latency. For example, an external function that accesses persistent state or requires complex logic may take a long time to complete. If the processing pipeline were to block while waiting for the function to return, it could significantly impact throughput. P4FPGA provides an asynchronous implementation, so that the processing of other packets can continue in parallel. This is roughly analogous to multi-threading, but without the associated cost of context switching.

4 FIXED-FUNCTION RUNTIME

The P4FPGA fixed-function runtime provides the execution environment for packet processing algorithms specified in P4. It defines an API that allows generated code to access common functionality through a set of target-agnostic abstractions. Consequently, the runtime system plays a crucial role in providing an efficient, flexible and portable environment for packet processing applications. It must provide an abstract architecture that is uniform across many different hardware platforms. It must also provide an efficient medium to transport data across processing elements. Finally, the runtime system must provide auxiliary functionalities to support control, monitoring, and debugging.

Note that P4 developers can create a variety of potential applications, ranging from packet switching to NFV style hardware offloading for packet processing. These applications have different requirements from the architecture that the fixed function runtime provides. To support these different use-cases, P4FPGA allows P4 developers to choose either of two architectures: *multi-port switching* or *bump-in-the-wire*. The multi-port switching architecture is suitable for network forwarding elements, such as switches and routers, and for experimenting with new network routing protocols. It includes an output cross-bar, as shown in Figure 4, to arbitrate packets to the appropriate egress port. The bump-in-the-wire architecture is suitable for network functions and network acceleration. It receives packets from a single input port, and forwards to a single output [24].

Below, we describe the design of the major components of the P4FPGA fixed function runtime. These components, indicated as grey boxes in Figure 4, include memory management, transceiver management, and host communication.

Memory Management. As packets arrive at the FPGA, they must be stored in memory for processing. This memory can be designed in two ways. A straight-forward approach is to use FIFO queues, which forward packets through processing elements in the order in which they are received. However, simple FIFO queues are not sufficient for implementing more advanced packet-processing features, such as quality-of-service guarantees. In particular, such features require re-ordering packets as they are processed.

Therefore, P4FPGA includes an optional memory buffer managed by a hardware memory management unit (MMU). The MMU interface defines two functions: `malloc` and `free`. The `malloc` function takes one parameter, the size of packet buffer to be allocated rounded up to 256-byte boundary, and returns a unique packet identifier (PID). The PID is similar to a pointer in C, and is used throughout the lifetime of the packet in the pipeline. Upon the completion of packet transmission, the PID (and associated memory) is returned to the MMU to be reused for future packets, via a call to `free`. Users can configure the amount of memory used for storing packets. By default, P4FPGA allocates 65,536 bytes of on-chip block RAM (BRAM) per packet buffer.

Transceiver Management. P4FPGA is portable across many hardware platforms. As a result, it provides a transceiver management unit that enables it to use the media access control (MAC) and physical (PHY) layers specific to a target platform. For instance, the P4FPGA transceiver management unit uses vendor-specific protocols without requiring changes to the P4 program.

Host Communication. P4FPGA integrates a host communication channel between the FPGA and host CPU. This is useful for implementing the control channel and for debugging. The host communication channel is built on top of the PCI express protocol, which is the de-facto protocol for internal communication within network devices, such as switches and routers. We provide both blocking and non-blocking remote procedure calls (RPC) between software and hardware. For example, it is possible for a host program to issue an non-blocking call to read hardware registers by registering a callback function to receive the returned value. Similarly, a controller can program match tables by issuing a function call with an encoded table entry as a parameter.

Timing Closure. Our general approach to the timing closure problem is as follows: First, we use pipeline FIFOs to ensure the inputs and outputs of parser, table and action blocks are registered. Second, we optimized the design of action engine and control flow logic to perform simple combinatorial logic in each cycle. If the logic is too complex to perform within a clock cycle and causes timing closure failure, we decompose the logic across multiple clock cycles. Third, the generated pipeline is constructed by template instantiation. These templates were designed to minimize timing issues by construction.

5 OPTIMIZATION

To ensure that the code generated by P4FPGA is efficient, we implemented a number of optimizations at both the compiler and micro-architectural level. Based on our experience, we have identified a few principles that we followed to improve the throughput and latency of the packet processing pipeline. Below, we describe the optimizations in the context of the NetFPGA SUME platform, but the same principles should apply to other platforms such as Altera DE5. For clarity, we present these principles in order of importance, not novelty.

Leverage hardware parallelism in space and time to increase throughput.

FPGAs provide ample opportunities to improve system throughput by leveraging parallelism in space, e.g., by increasing the width of the datapath. The throughput of a streaming pipeline, r, is determined the datapath width, w and the clock frequency, f ($r = w \times f$). The maximum clock frequency for an FPGA is typically 100s of MHz (a mid-end FPGA ranges from 200 to 400 Mhz). Therefore, in order to reach a throughput of 40 to 100 Gbps, it is necessary to use a datapath width in the range of 100s of bits to a few thousand bits.

On the NetFPGA SUME platform, we target an overall system throughput of 40Gbps on the four available 10Gbps Ethernet ports at 250 MHz. We used 128-bits for the parser datapath and 512-bits for the forwarding pipeline datapath. The theoretical throughput for the parser is 128 bits $\times$ 250 Mhz, or 32 Gbps. As a result, we replicate the parser at each port to support parsing packets at 10 Gbps.

Another important form of hardware parallelism is pipeline parallelism. We clock the P4 programmable pipeline at 250 MHz. If we process a single packet in every clock cycle, we would be able to process 250 Mpps (million packet per second). At 10 Gbps, the maximum packet arrival rate is 14.4 Mpps for 64 byte packets. At 250 Mpps, we should be able to handle more than sixteen 10 Gbps ports simultaneously with one P4 programmable pipeline. Of course, the theoretical maximum rate does not directly translate to actual system performance. Nonetheless, we conducted extensive pipelining optimizations to ensure that all generated constructs are fully pipelined. In other words, control flow, match tables and action engines are all fully pipelined.

Transform sequential semantics to parallel semantics to reduce latency.

The P4 language enforces sequential semantics among actions in the same action block, meaning that side effects of a prior action must be visible to the next. A conservative compilation strategy that respects the sequential semantics would allocate a pipeline stage for each action. Unfortunately, this strategy results in sub-optimal latency, since each stage would add one additional clock cycle to the end-to-end latency. P4FPGA optimizes latency by leveraging the fact that hardware inherently supports parallel semantics. As a result, we opportunistically co-locate independent actions in the same pipeline stage to reduce the overall latency of an action block.

Select the right architecture for the job.

Network functions can be broadly divided into two subcategories: those that need switching and those that do not. For example, network encryption, filtering, firewalling can be enforced on a per-port basis. This is especially true if interface speeds increase to 50 or 100Gbps, when CPUs barely have enough cycles to keep up with data coming in from one interface. On the other hand, prototyping network forwarding elements on FPGAs requires switching capability. As mentioned in Section 4, P4FPGA allows users to select the architecture most appropriate for their needs.

Use a resource-efficient components to implement match tables.

In P4FPGA generated pipelines, match tables dominate FPGA resource consumption. This is because FPGAs lack hardened content-addressable memory (CAM), an unfortunate reality of using FPGAs for network processing. Although one can implement CAM using existing resources on FPGAs, such as Block RAMs or LUTs, it is not efficient. High-end FPGAs have more resources on-chip to implement CAMs, but they also come at a premium price. To alleviate the situation, P4FPGA uses hash-based methods for table lookup. The compiler uses these more efficient implementation techniques by default. But, users may choose to use more expensive CAM implementations by specifying a compiler flag.

Eliminate dead metadata

A naïve P4 parser implementation would extract full header and metadata from packets by default. This can be wasteful if the extracted headers are not used in the subsequent pipeline. P4FPGA analyzes all match and action stages, and eliminates unused header fields and metadata from the extracted packet representation.

Use non-blocking access for external modules.

Stateful processing is expensive on high-performance packet-processing pipelines. Complex operations may require multiple clock cycles to finish, which can negatively affect performance if pipelining is only performed at the function level. P4FPGA implements fine-grained pipelining on stateful elements to maintain high throughput. For example, a memory read operation requires issuing a read request to memory and waiting for the corresponding response. Due to the high latency of memory, the response may only come after multiple cycles of delay. In P4FPGA, we support split-phase reads such that a read request and response can happen at different clock cycles. Meanwhile, the pipeline can continue processing other packets.

6 IMPLEMENTATION

Our prototype P4FPGA implementation consists of a C++-based compiler along with a Bluespec-based runtime system. For the frontend, we reused P4.org's C++ compiler frontend to parse P4 source code and generate an intermediate representation [30]. We designed a custom backend for FPGAs, which consists of 5000 lines of C++ code. The runtime is developed in a high-level hardware description language, Bluespec [27]. Bluespec provides many higher level hardware abstractions (e.g., FIFO with back-pressure) and the language includes a rich library of components, which makes development easier. The runtime is approximately 10,000 lines of Bluespec. We relied on Bluespec code from the Connectal project [21] to implement the control plane channel. We also implemented mechanisms to replay `pcap` traces, access control registers, and program dataplane tables. All code is publicly available under an open-source license.[1]

Complex FPGA-based systems often require integration with existing intellectual property (IP) components from other vendors and P4FPGA is no exception. We allow third-party IPs to be integrated with the existing P4FPGA runtime system as long as those components conform to the interfaces exposed by P4FPGA runtime. For example, we currently support IP cores such as MAC/PHY and Ternary CAM (TCAM) provided by FPGA vendors and commercial IP vendors [5].

7 EVALUATION

In this section, we explore the performance of the P4FPGA. Our evaluation is divided into two main sections. First, we evaluate the ability of P4FPGA to handle a diverse set of P4 applications. Then, we use a set of microbenchmarks to evaluate the individual components of P4FPGA in isolation.

Toolchain and hardware setup. We evaluate the performance of P4FPGA generated designs against a set of representative P4 programs. Each program in our benchmark suite is compiled with the P4FPGA compiler into Bluespec source code, which is then processed by a commercial compiler from Bluespec Inc. to generate Verilog source code. Next, the Verilog source code is processed by the standard Vivado 2015.4 tool from Xilinx, which performs synthesis, placement, routing and bitstream generation. The compilation framework supports both the Altera tool suite, Quartus, and Xilinx tool suite, Vivado. For this evaluation, we only used Vivado. We deployed the compiled bitstream on a NetFPGA SUME platform with a Xilinx Virtex-7 XC7V690T FPGA, with 32 high-speed serial transceivers to provide PCIe (Gen3 x8) communication and 4 SFP+ ports (10Gbps Ethernet).

For packet generation, we built a custom packet generator that is included as part of the P4FPGA runtime. It generates packets at a user-specified rate. We also provide a utility to program the packet generator with a packet trace supplied in the `PCAP` format or to configure/control the packet generator from userspace. Similarly, we provide a built-in packet capture tool to collect output packets and various statistics.

[1] http://www.p4fpga.org

7.1 Case Studies

To illustrate the broad applicability of P4FPGA, we implemented three representative P4 applications as case studies. We chose these examples because (i) they represent non-trivial, substantial applications, (ii) they illustrate functionality at different layers of the network stack, and (iii) they implement diverse functionality and highlight P4's potential.

Table 1 shows the lines of code in P4 for each of these applications. As a point of comparison, we also report the lines of code for the generated Bluespec code. While lines of code is not an ideal metric, it does help illustrate the benefit of high-level languages like P4, which requires orders-of-magnitude fewer lines of code. Below, we describe each of these applications in detail.

L2/L3 Forwarding. P4 was designed around the needs of networking applications that match on packet headers and either forward out a specific port, or drop a packet. Therefore, our first example application performs Layer 2 / Layer 3 forwarding. It uses a switching architecture and routes on the IP destination field.

Paxos. Paxos [22] is one of the most widely used protocols for solving the problem of *consensus*, i.e., getting a group of participants to reliably agree on some value used for computation. The protocol is the foundation for building many fault-tolerant distributed systems and services. While Paxos is traditionally implemented as an application-level service, recent work demonstrates that significant performance benefits can by achieved by leveraging programmable data planes to move consensus logic in to network devices [11, 12, 25]. The P4 implementation [11] defines a custom header for Paxos messages that is encapsulated inside a UDP packet. The program keeps a bounded history of Paxos packets in registers, and makes stateful routing decisions based on comparing the contents of arriving packets to stored values. Paxos uses the switch architecture with one input and one output port, essentially a bump-in-the-wire.

Market Data Protocol. Many financial trading strategies critically depend on the ability to react quickly to changing market condition, and to place orders at high speeds and frequencies. Platforms that implement these trading algorithms would therefore benefit by offloading computations into hardware using custom packet headers and processors. As a proof-of-concept for how P4 could be used for financial applications, we implemented a commonly used protocol, the Market Data Protocol (MDP). MDP is used by the Chicago

Table 1: Example applications compiled by P4FPGA and lines of code (LoC) in P4 and Bluespec. The framework includes P4FPGA runtime and control plane support.

Name	Description	LoC in P4	LoC in Bluespec	Framework
l2l3.p4	L2/L3 router	170	1281	33295
mdp.p4	variable packet length, financial trading protocol	205	1812	33295
paxos.p4	stateful processing, consensus protocol	385	3306	33295

Table 2: Latency breakdown, cycles @ 250MHz. Note that memory is only accessed in the shared memory configuration.

App	Size	Parser	Table	Memory	Deparser
l2l3	64	2	31	21	11
	256	2	31	23	32
	512	2	31	24	66
	1024	2	31	23	130
mdp	256	15	9	23	34
	512	35	9	24	68
	1024	88	9	23	130
paxos	144	6	42	21	12

Mercantile Exchange. Essentially, MDP is a L7 load balancer. An MDP P4 implementation is complicated by the fact that the protocol header is variable length. Figure 5 shows the header definitions for a book refresh message. A "book" is an entity that keeps the most recent stock price. A book refresh message has a fixed header *mdp_t* that is common to all MDP protocol messages, as well as a variable length header, *refreshBook*, with one or more entries *refreshBookEntry*. A field *numEntries* in *refreshBook* dictates how many entries must be extraced by the parser. Our P4 implementation of MDP can address the header variable length and also parse the input packet stream, filter duplicated messages, and extract important packet fields for additional processing.

Table 3: Latency comparing to vendors. The latency of cut-through switch (Arista 7050QX) is from [4]

Mode	Packet Size			
	64	**256**	**1024**	**1518**
Arista 7050QX	550ns	550ns	550ns	550ns
P4FPGA (L2/L3)	340ns	420ns	810ns	1050ns

```
header_type mdp_t {
    fields {
        msgSeqNum : 32;
        sendingTime : 64;
        msgSize : 16;
        blockLength : 16;
        templateID : 16;
        schemaID : 16;
        version : 16;
}}
header_type event_metadata_t {
    fields {
        group_size : 16;
}}
header_type refreshBook {
    fields {
        transactTime : 64;
        matchEventIndicator : 16;
        blockLength: 16;
        numEntries: 16;
}}
header_type refreshBookEntry {
    fields {
        mdEntryPx : 64;
        mdEntrySize : 32;
        securityID : 32;
        rptReq : 32;
        numberOfOrders : 32;
        mdPriceLevel : 8;
        mdUpdateAction : 8;
        mdEntryType : 8;
        padding : 40;
}}
```

Figure 5: Header definitions for MDP.p4.

Processing time and latency. Our evaluation focuses on two metrics: processing time and latency. Table 2 shows the processing time for each application on a single FPGA. Note that memory is only access in a shared memory architecture configuration. All latency measurements are taken from a

cycle-accurate simulation, which is as precise as measurement on actual FPGA hardware. The numbers are in term of cycles running at 250MHz, where each cycle is 4 nanoseconds. We measured the packet-processing time of each application on small and large packets. Since the L2/L3 application only parses Ethernet and IP headers, parsing only takes 2 cycles, or 8 ns. On the contrary, the MDP application spends more time parsing because it performs variable-length header processing and inspects packet payload for market data. Match and action stages are constant time for each application For example, L2/L3 spends 31 cycles or 124 ns in match and action stage. The time is spent on table look-up, packet field modification and packet propagation through multiple pipeline stages. The amount of time spent in a match and action stage depends on the number of pipeline stages and the complexity of actions performed on a packet. Memory access accounts for time taken to access a shared memory buffer, and therefore is always a constant overhead among all packets. The time required for the deparser, which must reassemble and transmit the packet, is proportional to the packet size. Even though the latency for a single packet may be 65 cycles or longer (e.g. L2L3 with 64 byte packets), a pipeline has a lot of parallelism and a piplined stage may take as much as 10 to 20 cycles for a table access with or without memory.

We define the pipeline latency as the time from when the first bit of packet enters the P4 pipeline (right after the RX channel in Figure 4) until the first bit of packet exits the pipeline (right before the TX channel 4). In all three cases, P4FPGA processes packets with low latency. The additional latency in the program generated by P4FPGA in Table 3 is caused by serializing and deserializing packets to and from the packet buffer (store-and-forward). To place the latency numbers in context, we report the performance results from Arista 7050QX cut-through switch in Table 3. As we can see, P4FPGA is able to offer latency comparable to commercial off-the-shelf switches.

Packet processing is heavily pipelined and we can sustain 10Gbit/s line rate at all packet sizes for all our test applications. We note that the shared memory buffer architecture imposes some overhead due to the memory management unit. Specifically, the malloc and free operations do not support pipelining. Currently, the shared memory buffer implementation supports up to 10Mpps, which is less than line rate for packets smaller than 125 bytes. We expect that this performance could be further optimized with additional engineering.

7.2 Microbenchmarks

The next part of our evaluation focuses on a set of microbenchmarks that evaluate different aspects of P4FPGA in isolation. We investigate the following questions:

- How does the runtime perform?
- How does overall pipeline perform?
- How much of the FPGA resources are required for the pipeline and runtime?

We focus on three metrics for evaluation: throughput, latency and resource utilization on the FPGA. We present the details of these microbenchmarks below.

7.2.1 Fixed-Function Runtime. The target FPGA board consists of 4x 10Gbps ports. As a result, the runtime system must sustain line-rate forwarding at 40Gbps to avoid being a bottleneck to overall system performance. To verify that P4FPGA is able to satisfy this requirement, we measured the raw throughput of the fixed-function runtime with an empty packet processing pipeline (no table, no action in ingress or egress pipeline).

In this experiment, we used a runtime configured with six input and output ports. Each input port receives traffic from a built-in packet generator at full 10Gbps line rate. The two additional ports in the runtime runtime can be used to send packets to the host CPU through a DMA engine or to recirculate packets from the egress pipeline to the ingress pipeline. However, these operations are out-of-the-scope for this paper. We loaded a packet trace with packet sizes ranging from 64 to 1516 bytes and replayed the packet trace a million times. The fixed function runtime is able to sustain between 53.3Gbps for 64 bytes packets and 59.5Gbps for 1518 bytes packets, which is well above the required 40Gbps throughput requirement.

7.2.2 Programmable pipeline. We evaluated the performance of a generated P4 pipeline with a set of microbenchmarks that focused on each key language construct in isolation: parsers, tables, and actions. As a point of comparison, we also report results for running the same experiments with the PISCES [36] software switch. PISCES extends Open vSwitch [29] with a protocol independent design. In all cases, PISCES uses DPDK [15] to avoid the overhead of the kernel network stack. Note, to make the comparison equal, we used only two ports for PISCES.

Parser. We used the packet generator to send 256-byte packets with an increasing number of 16-bit custom packet headers. We measured both latency and throughput, and the results are shown in Figures 6 and 7. As expected, we see that parsing latency increases as we increase the number of extracted headers. In terms of absolute latency, P4FPGA is able to leverage the performance of FPGAs to significantly reduce latency. P4FPGA took less than 450 ns to parse 16 headers, whereas PISCES took 6.5us. The results for throughput are similar. For both P4FPGA and PISCES, the parser throughput decreases as the number of headers increases. As expected, P4FPGA significantly outperforms PISCES in terms of absolute throughput as well as the number of headers that parse without performance degradation.

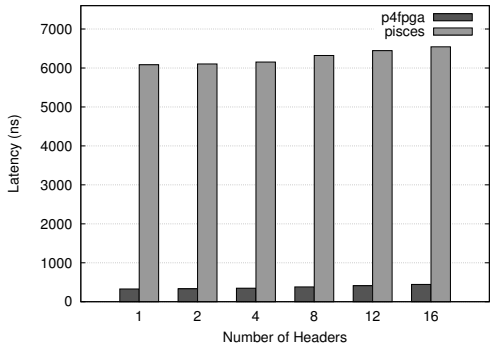

Figure 6: Parser latency v.s. number of headers parsed

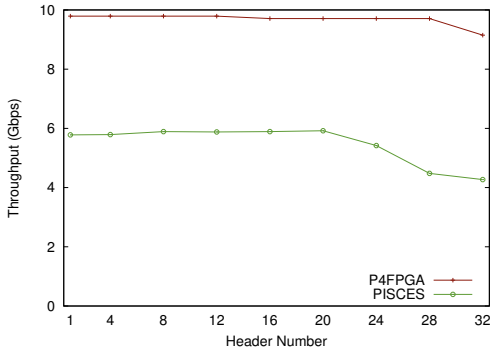

Figure 7: Parser throughput v.s. number of headers parsed

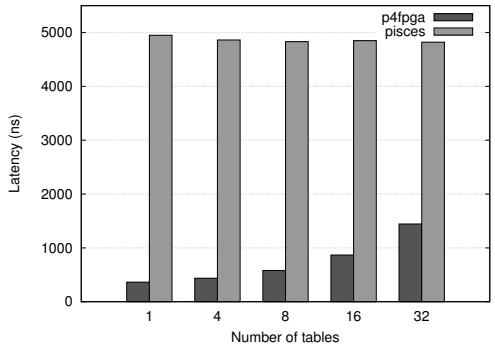

Figure 8: Processing latency v.s. number of tables

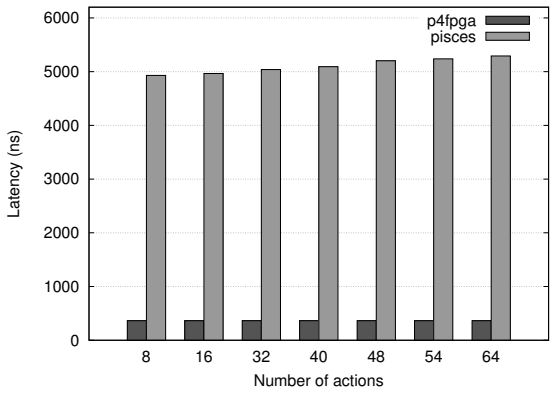

Figure 9: Pipeline latency v.s. number of actions

Table. In this experiment, we compiled a set of synthetic programs with an increasing number of pipeline stages (1 to 32). We measured the end-to-end latency from the entry of the ingress pipeline to the exit of the egress pipeline. The result is shown in figure 8. Although the absolute latency is much better for P4FPGA, the trend shows that the processing latency increases with the number of tables. In contrast, the latency for PISCES remains constant. This is because PISCES implements an optimization that fuses multiple match-action pipeline stages into a single match-action rule. We have not yet implemented this optimization for P4FPGA.

Action. In this experiment, we evaluate how the action complexity can affect throughput. We vary the number of header field writes from 8 to 64. All field write operations are independent, meaning that they write to different fields in the packet header. Hence, P4FPGA is able to leverage hardware parallelism to perform all write operations within the same clock cycle, as there is no dependency between any operation. Note that this faithfully implements the sequential semantics of the original P4 source program, even though all actions are performed in parallel. As shown in Figure 9, the end-to-end packet processing latency in P4FPGA remains the same at 364 ns. This is in contrast to PISCES, which consumes more CPU cycles to process write operations, as the operations are performed in sequence on a CPU target [36]. In other words, the absolute latency is much higher on a software target, and it also increases with the number of write operations. In contrast, with P4FPGA, the latency remains low and constant independent of the number of writes in a stage.

7.2.3 Resource Utilization. We report the resource utilization of the FPGA in two parts: the resource consumed by the fixed function runtime which is common to all P4 programs; and the resource consumed by individual P4 constructs which is variable depending on parameters specified by P4 program. We quantify resource consumption with the number and percentage of look-up tables (LUTs) and memory consumed by each block.

The runtime subsystem implements PCIe host communication, and the MAC and PHY layers of the Ethernet protocol. As shown in Table 4, the total resource consumption of the runtime is about 7.5% of total available LUTs and 2.3% of available memory blocks, which leaves many of the resources available to implement the actual logic of a P4 program.

Table 4: Area and frequency of fixed function runtime

	Slice LUTs	Block RAMs	MHz
PCIe	6377	9	250
10G MAC x4	8174	0	156
10G PHY x4	10422	0	644.5
Connectal	7867	25	250
Area Used	32700 (7.5%)	34 (2.3%)	-

Next, we profile resource consumption of major P4 constructs: match table, parser, deparser and action. Match tables are implemented with content-addressable memory (CAM) to perform key lookup, and regular memory to corresponding action for a matched key entry. Unlike ASICs, FPGAs lack native support for CAM, and as a result, we had to emulate CAM by implementing it with regular memory blocks. We evaluated three different CAM implementations on the FPGA: binary CAM for implementing exact match, ternary CAM for implementing ternary and longest prefix match, and hash-based CAM for exact match.

As shown in Table 5, we can implement up to a 288-bit key binary CAM (BCAM), ternary CAM (TCAM), or a hash-based associative memory with minimum resource utilization. The commercial-grade TCAM implementation is more efficient than our BCAM. We suspect that the difference is due to both implementation efficiency and internal architecture of these two CAM technologies. But, the hash-based associative memory implementation is the most efficient among all three implementations [14]. If we were to use the whole FPGA for only a CAM with a 288-bit key, then a BCAM, TCAM, and hash-based associative memory can fit up to 6K, 53K, 93K entries on a Virtex-7 FPGA, respectively. To put these numbers into context, a Mellanox Spectrum ASIC allows 9K entries of 288 bit rules in a common TCAM table shared between ingress and egress pipeline.

8 RELATED WORK

We briefly survey related work on P4 compilers, use of FPGAs in networking, and FPGA synthesis.

P4 Compilers. Given the significant interest in P4 as a development platform, there are several efforts underway to implement P4 compilers and tools. Our microbenchmarks compare against PISCES [36], which is a software hypervisor switch that extends Open vSwitch [29] with a protocol-independent design. The Open-NFP [28] organization provides a set of tools for developing network function processing logic, including a P4 compiler that targets 10, 40 and 100GbE Intelligent Server Adapters (ISAs) manufactured by Netronome. These devices are network processing units (NPUs), while P4FPGA targets FPGAs. The Open-NFP compiler currently does not support register related operations and cannot parse header fields larger than 32 bits. Users implement actions in MicroC code external to the P4 program. P4c [33] is a retargetable compiler for the P4 language which generates high performance network switch code in C, linking against DPDK [15] libraries. DPDK provides a set of user-space libraries, which bypass the Linux kernel. P4c does not yet support P4 applications that require register uses to store state. P4.org provides a reference compiler [34] that generates a software target, and can be executed in a simulated environment (i.e., Mininet [26] and P4 Behavioral Model switch [32]). P4FPGA shares the same compiler front-end, but provides a different back-end.

A P4 compiler backend targeting a programmable ASIC [20] must deal with resource constraints. The major challenge arises from mapping logical lookup tables to physical tables on an ASIC. In contrast, FPGAs can directly map logical tables into the physical substrate without the complexity of logical-to-physical table mapping, thanks to the flexible and programmable nature of FPGAs.

Perhaps the most closely related effort is Xilinx's SDNet [39]. SDNet compiles programs from the high-level PX [8] language to a data plane implementation on a Xilinx FPGA target, at selectable line rates from 1G to 100G. A Xilinx Labs prototype P4 compiler works by translating from P4 to PX, and then using SDNet to map this PX to a target FPGA. As the compiler implementation is not yet publicly available, we cannot comment on how the design or architecture compares to P4FPGA.

FPGAs for networking. The NetFPGA project [40] is another open-source framework that researchers frequently use to prototype networking ideas. P4FPGA shares the same vision with NetFPGA to provide an open framework for network researchers. Furthermore, P4FPGA can support not only NetFPGA-specific hardware platforms, but also many other existing FPGA platforms on the market.

A unified software-hardware co-design framework simplifies the FPGA development process [21]. P4FPGA leveraged the idea of generating SW/HW interfaces from a interface definition file from Connectal [21], which has greatly simplified the generation of a control-plane interface for P4 prototyping on FPGAs.

High-level Synthesis. FPGAs are typically programmed using hardware description languages such as Verilog. Many developers find working with these languages challenging, as they expose low-level hardware details to the programmer. Consequently, there has been significant research in high-level synthesis and programming language support for FPGAs. Some well-known examples include CASH [9], which compiles C to FPGAs; Kiwi [37], which transforms .NET

Table 5: BCAM and TCAM Resource Utilization on Virtex 7 XCVX690T, which has 1470 BRAM blocks, 866400 Flip-flops and 433200 LUTs. We show resource utilization as percentage as well as actual amount of resource used.

Hardware	Key Size (Bits)	#Entries	% BRAM	#Flip-Flops	#LUTs
BCAM	36	1024	2.2% (32/1470)	0.26% (2280 / 866400)	0.59% (2552 / 433200)
	72	1024	4.4% (64/1470)	0.46% (3992 / 866400)	1.1% (4642 / 433200)
	144	1024	8.7% (128/1470)	0.92% (7976 / 866400)	2.2% (9589 / 433200)
	288	1024	17.4% (256/1470)	1.8% (15944 / 866400)	4.5% (19350 / 433200)
TCAM	72	2048	2.7% (40/1470)	0.56% (4900 / 866400)	2.1% (9100 / 433200)
	144	2048	3.2% (48/1470)	0.63% (5482 / 866400)	2.8% (12033 / 433200)
	288	2048	3.9% (58/1470)	0.85% (7430 / 866400)	4.4% (18977 / 433200)
HASH	72	1024	0.7% (10.5/1470)	0.12% (1053 / 866400)	0.27% (1185/433200)
	144	1024	0.8% (12.5/1470)	0.16% (1440/866400)	0.32% (1395/433200)
	288	1024	1.1% (16.5/1470)	0.25% (2232/866400)	0.46% (2030/433200)

progrrams into FPGA circuits; and Xilinx's AccelDSP [2], which performs synthesis from MATLAB code.

P4FPGA notably relies on Bluespec [27] as a target language, and re-uses the associated compiler and libraries to provide platform independence. As already mentioned, P4FPGA uses Connectal [21] libraries, which are also written in Bluespec, for common hardware features.

9 CONCLUSION

FPGAs offer performance that far exceeds software running on general purpose CPUs, while offering a degree of flexibility that would be difficult to achieve on other targets, such as ASICs. At the same time, they are also notoriously difficult to program. P4FPGA lowers the barrier to entry, giving programmers a programmable substrate for creating innovative new protocols and applications.

P4FPGA provides a P4-to-FPGA compiler and runtime that is flexible, portable, and efficient. It supports multiple architectures, generates code that runs on Xilinx or Altera FPGAs and runs at line-rate with latencies comparable to commercial ASICs. P4FPGA is open source and publicly available for use. Indeed, it has already been used by two research projects to evaluate P4 programs on hardware. We hope that this research will help other users in real environments or to support systems and networking research.

10 AVAILABILITY

P4FPGA is publicly available under an open-source license. All source code, as well as example P4 source programs and their generated Bluespec counterparts are available at http://p4fpga.org. Furthermore, benchmarks are available via the P4 Whippersnapper Benchmark Suite [13].

ACKNOWLEDGEMENTS

This research is partially supported by Swiss NSF (166132 and 159537), European Union's Horizon 2020 research and innovation programme under the SSICLOPS project (agreement No. 644866), DARPA CSSG (D11AP00266), NSF (1053757, 1440744, and 1422544), and with gifts from Cisco, Xilinx, Altera and Bluespec. We thank Jamey Hicks and John Ankcorn for their help with Connectal, Nagase for providing the TCAM IP core, our shepherd Luigi Rizzo, and the SOSR reviewers for helpful comments.

REFERENCES

[1] A. Abdelhadi and G. Lemieux. Modular SRAM-Based Binary Content-Addressable Memories. In *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2015.

[2] AccelDSP Synthesis Tool. http://www.xilinx.com/tools/acceldsp.htm.

[3] Algorithms in Logic. www.algo-logic.com.

[4] Arista 7050X Switch Architecture. https://solutions.arista.com/hubfs/Arista/Datasheets/Arista_7050X_Switch_Architecture_V0.51_2.pdf.

[5] Axonerve. Axonerve Low Latency Matching Engine Synthesizable IP Core.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[7] 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 *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, Aug. 2013.

[8] G. Brebner and W. Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE/ACM International Symposium on Microarchitecture*, Jan. 2014.

[9] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[10] Corsa. Corsa DP6420 OpenFlow data plane. http://www.corsa.com/products/dp6420.

[11] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, Apr. 2016.

[12] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM SOSR*, pages 59–73, June 2015.

[13] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soul, and H. Weatherspoon. Whippersnapper: A P4 Language Benchmark Suite. In *ACM SIGCOMM SOSR*, 2017.

[14] U. Dhawan and A. Dehon. Area-Efficient Near-Associative Memories on FPGAs. *ACM Transactions on Reconfigurable Technology System*, Jan. 2015.

[15] DPDK. http://dpdk.org/.

[16] ExaBlaze. Exalink Fusion. https://exablaze.com/exalink-fusion.

[17] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. H. Dave, A. W. Moore, and P. G. Neumann. Blueswitch: Enabling Provably Consistent Configuration of Network Switches. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2015.

[18] T. Jepsen, L. P. de Sousa, H. T. Dang, F. Pedone, and R. Soulé. Optimistic aborts for geo-distributed transactions. *CoRR*, abs/1610.07459, 2016.

[19] J. Z. J.K Lee. LBSwitch: Your Switch is Your Server Load-Balancer . http://p4.org/p4-workshop-2016/, May 2016.

[20] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, May 2015.

[21] M. King, J. Hicks, and J. Ankcorn. Software-Driven Hardware Development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2015.

[22] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, May 1998.

[23] P. Lapukhov. Data-plane probe for in-band telemetry collection. https://tools.ietf.org/html/draft-lapukhov-dataplane-probe-00.

[24] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2016.

[25] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI16*, Nov. 2016.

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

[27] R. Nikhil and K. Czeck. BSV by Example. CreateSpace, 2010.

[28] Open-NFP. http://open-nfp.org/.

[29] Open vSwitch. http://www.openvswitch.org.

[30] P4. P4 Behavioral Model. https://github.com/p4lang/p4c-bm.

[31] P4. P4 Specification. http://p4.org/spec/.

[32] P4 Behavioral Model. https://github.com/p4lang.

[33] P4@ELTE. http://p4.elte.hu/.

[34] P4.org. http://p4.org.

[35] A. Putnam, A. Caulfield, E. Chung, and D. Chiou. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, June 2014.

[36] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2016.

[37] S. Singh and D. J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[38] T. Tofigh. Dynamic Analytics for Programmable NICs Utilizing P4. http://p4.org/p4-workshop-2016/, May 2016.

[39] Xilinx. SDNet. http://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[40] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, Sept. 2014.