P4 Weaver: Supporting Modular and Incremental Programming in P4

Ali Fattaholmanan
Università della Svizzera italiana

Mario Baldi
Politecnico di Torino

Antonio Carzaniga
Università della Svizzera italiana

Robert Soulé
Yale University

ABSTRACT

In this paper, we introduce P4 Weaver as an approach towards bringing modularity into the P4 language. P4 Weaver is designed to merge new data plane features into a base program in a principled and controlled way, so as to preserve the reliability of the switch. We also present an architecture for an integrated development environment that supports modular P4 programming while also safeguarding the intellectual property of the vendor code. We demonstrate the utility of P4 Weaver by adding three popular but non-trivial protocols to a P4 switch. We show that modularity is indeed beneficial and that P4 Weaver supports modularity efficiently and reliably.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Software and its engineering** → *Integrated and visual development environments*; *Source code generation*.

KEYWORDS

modular data-plane programming, source-to-source P4 compiler

ACM Reference Format:
Ali Fattaholmanan, Mario Baldi, Antonio Carzaniga, and Robert Soulé. 2021. P4 Weaver: Supporting Modular and Incremental Programming in P4. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), September 20–21, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3482898.3483353

1 INTRODUCTION

P4 programmable devices have the potential to bring a major paradigm shift in the way computer networks are designed and deployed. Traditionally, network devices have been built around a fixed set of data plane functions. Users were forced to find creative engineering solutions using available features to address their use cases. Now network administrators can follow a different approach: starting from their use case, they identify the data plane behavior that addresses it in the best way and implement such behavior (or have it implemented by a third party) on a programmable network device. However, network dataplane programming also introduces new problems and challenges.

First, network programming is *programming*, which is never easy, and network programming is relatively new and lacks mature tools to guide and support the development process. This kind of support is especially important because network operators, although technologically savvy, are not necessarily software developers. Moreover, network programming languages such as P4 [7] are intentionally computationally restricted, which can make it cumbersome to express intended algorithms.

Second, there is the question of who will do the programming. Some vendors might develop their platforms around a programmable ASIC without exposing programmability to the user [2, 11, 28, 29]. At the other end of the spectrum, some operators might choose bare-metal programmable switches [14] to then develop the whole data plane and control plane stack from scratch. In these cases it is conceivable that a single development team would own a monolithic data-plane program.

However, arguably the most widely beneficial and effective case is somewhere in the middle. A vendor would sell a switch with a pre-programmed data plane profile with standard features (e.g., L2/L3 forwarding, tunneling, and ACLs), as is the case with existing Arista [2] and Cisco [11] products, and the customer would program the switch (or deploy third-party programs) to extend standard features or to implement new protocols. In other words, network programming is likely to be *incremental* and therefore modular.

This immediately leads to integration problems. The vendor would want to be very careful in exposing their code to the customer, and the customer should be careful in integrating new code within the vendor code. This is for business reasons, for example because the vendor might want to protect their intellectual property, but also for reliability, since both the vendor and the customer would want to guarantee the correct functionality of the switch.

In this paper, we propose P4 Weaver, a new environment for incremental P4 programming. P4 Weaver allows switch customers to implement and deploy their new ideas and algorithms while ensuring that the switch remains reliable in its basic traffic-handling functionality.

P4 Weaver introduces annotations in P4 programs for both the vendor code and the extension code. Annotations in the vendor code provide fine-grained access control over the core functionality of the switch. For example, the vendor might annotate their code to expose TCP headers and to use a new forwarding scheme for a particular application's traffic, while at the same time preventing any change in the headers and the handling of ARP packets. Annotations within the extension code describe how the extension code should be

SOSR '21, September 20–21, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9084-2/21/09…$15.00
https://doi.org/10.1145/3482898.3483353

integrated within the base code of the switch. For example, the customer might want/be able to add telemetry headers and the corresponding processing before UDP or TCP processing, but after IP forwarding.

In modern systems, incremental and modular programming is supported by language constructs and by system features, such as method invocation and dynamic linking. However, P4 does not provide such constructs and features. This restriction is a deliberate design choice motivated by efficiency and by the nature of the target platform. We therefore design the core of P4 Weaver as a source-to-source compiler that combines the customer and vendor code.

P4 Weaver is implemented within a client-server architecture that supports the programmer through a modern integrated development environment (IDE). This architecture hides and protects the vendor code, but at the same time supports the customer in programming against the headers and controls that are annotated as accessible by the vendor. For example, the IDE supports the customer adding a telemetry header to TCP packets even without giving the customer access to the TCP header definitions. P4 Weaver then integrates the extension code within the vendor code, compiles it, and delivers the resulting binary code and other artifacts to the customer for deployment onto the switch. If the extension code violates the integration rules mandated by the vendor or if the integrated code would exhaust the resources of the switch, then P4 Weaver provides appropriate feedback to the customer (through the IDE) with error messages or resource allocation reports, respectively.

We evaluate P4 Weaver with some micro-benchmarks and primarily through significant case studies. For example, we use P4 Weaver to integrate new functions into a switch, including the GPRS Tunneling Protocol (GTP) with ECMP routing, and in-band network telemetry (INT). This diversity of applications demonstrates the flexibility and expressiveness of P4 Weaver.

In summary, we make the following contributions:

- We design annotations for the P4 language that support the principled and controlled extension of P4 programs;
- We implement a source-to-source compiler that integrates new code within an existing P4 programs following proper syntactical structures and abiding by the access-control policies defined through annotations;
- We implement a development environment that supports incremental P4 programming using annotations and the source-to-source code weaver;
- We demonstrate the use and utility of the annotations and the development environment through a series of practical examples in which we extend a switch with real features and protocols that are of interest to network operators.

The rest of this paper is organized as follows. We first discuss the background and provide a motivating example (§2). Then, we present the design of the P4 Weaver annotations and the weaving process (§3), followed by a discussion of the system architecture (§4). Next, we evaluate P4 Weaver with a set of micro-benchmarks and case studies (§6). Finally, we discuss related work (§7) and conclude (§8).

2 MOTIVATION AND BACKGROUND

Modular design is crucial for code re-usability, as it allows developers to change different system units independently. Indeed, modular design and information hiding are the bedrocks of modern software development [27].

In many software frameworks, modularity is a feature provided by the programming language and the compiler. However, today, the support for modular development in P4 is limited. In this section, we motivate the need for modularity in P4, discuss the challenges for supporting modularity in the language, and consider aspect-oriented programming techniques as a possible approach to address them.

Who is the P4 Programmer? A natural question to ask is who, exactly, will be programming in P4? One can imagine several alternative scenarios.

The "whitebox" deployment of a switching system with a programmable data plane requires the device user to implement from scratch a data plane program and the corresponding control plane software (possibly starting from a publicly available or proprietary code base). This represents a significant shift from the traditional, "turn-key" deployment of networking equipment: devices come with pre-defined data plane functions and a network operating system controlling them, and their users are concerned only with configuring such functions using well-known, possibly standard, interfaces such as a CLI (command line interface), SNMP (simple network management protocol), and NETCONF.

To both benefit from a programmable data plane, and also offer a more familiar deployment model, some equipment vendors, such as Cisco Systems with their Nexus 3400 switches [11] and Arista Networks with their 7170 Series switches [2], have marketed products that lend themselves to the traditional turn-key deployment. Their switches, built around the Barefoot Networks P4 programmable Tofino chip [4], are released with a working data plane program controlled by their signature operating system (NXOS and EOS, respectively). Similarly, in the Data Processing Unit (DPU) world, Pensando Systems has developed a full software stack implementing distributed services for the Enterprise [29] and for Cloud Providers [28] that runs on a programmable ASIC designed in house.

For vendors, programmable data plane chips offer several advantages in terms of reduced development time and cost over traditional designs based on fixed-function ASICs. However, turn-key deployment of programmable data plane chips does not allow their customers or third parties to take advantage of the programmability of the data plane.

Ideally, then, there should be support for an incremental extension and composition of functions in a basic data plane program. In other words, there should be support for *modularity* in P4 programs.

Motivating Example. As a motivating example, imagine that a network operator would like their network to use Generic Network Virtualization Encapsulation (GENEVE) as their virtualization protocol. They purchase a switch from a vendor that supports a variety of L2/L3 forwarding features, including two virtualization protocols, Virtual Extensible LAN (VXLAN) and Network Virtualization using Generic Routing Encapsulation (NVGRE), but not GENEVE. The user might modify the vendor's base P4 program to add support for the GENEVE protocol. This would require that they:

(1) Add a new header definition for GENEVE.
(2) Modify the parser finite state machine.
(3) Add a new table to match on GENEVE header fields.
(4) Add new actions for processing and possibly remove the GENEVE header.
(5) Add a new table to match on various header fields to identify packets that need to be encapsulated in a GENEVE header.
(6) Add new actions to build and add a GENEVE header.
(7) Modify the composition of the table pipeline.
(8) Update the deparser to emit the new header.

While perhaps conceptually straightforward, making these changes is not a trivial task, even if the amount of code to be written is possibly small. The open source *switch.p4* program, which is comparable to a vendor base program, is roughly 5600 lines of code and implements essentially all of the functionality found on a modern data center switch, including L2 switching, L3 routing, multicast, LAG, ECMP, ACLs, MPLS, multi-device fabrics, and mirroring. Even just identifying the spots where the program must be modified in order to merge the new code within it is a daunting task. Moreover, a small change in this complex code base can inadvertently break some other functionality.

Requirements. In a word, the example above suggests that P4 programming should be *modular.* The context of the example is an incremental deployment, where a customer adds a function to a base vendor program, but the idea of modularity applies more generally, for example when an organization divides the development of a P4 program between two teams. We therefore refer generically to the addition of *extension* code to a *base* data plane program, and we use the terms *vendor* and *customer* to refer generically to the developers of the base and extension code, respectively. We now articulate specific modularity requirements.

- *Do not break what works.* The base data plane program of a switch usually implements protocols that are essential for the correct operation of the switch. The addition of new extension code should not affect the correct operation of the base program.
- *Do not break the control plane.* The operating system of a network device typically relies on the data plane offering a specific set of functions and a well-known interface to control these functions. The incremental addition of new code should not disable the operating system by changing or otherwise affecting the interface between data plane and control plane.
- *Do not show, do not see.* The base program is often developed by a switch vendor who might not want to share it with their customers. In all the above examples of turn-key systems [2, 11, 28, 29] and even in the case of features developed on stand-alone chips [4, 12], vendors are not willing to let go of the intellectual property on which they have heavily invested by open-sourcing their code.[1] On their part, customers would most likely want to develop extensions without having to study or even see the base code. In any case, hiding code is essential for modularity, so that the extension would not rely on the implementation details of the base program, and vice-versa. Being able to extend a base code without needing to know its details, significantly reduces the complexity of implementing such extension, which is beneficial even when the base code is actually available. This is the case, for example, of a developer adding new features to a well tested P4 program previously developed within their own company.
- *Resource sharing and isolation.* Programmable chips have a limited amount of resources (e.g. RAM, T-CAM, ALUs, registers, etc.) that must be shared between the base and incremental code. Such resources are used to store both the data plane program and the parameter data provided by the control plane. Since the base and incremental programs share resources, mechanisms must be in place to ensure that the extension would not modify the functionality and the parameters of the base program or exhaust resources needed by the base program.

[1]Although Intel/Barefoot has released P4 code in the public domain, it is only a subset of their full code base.

2.1 Modularity Challenges

Today, in P4, there are three main ways to support modular development. First, actions group together a sequence of instructions that may be parameterized. In $P4_{16}$, actions can be called directly, like a function invocation. Second, both $P4_{14}$ and $P4_{16}$ use the C preprocessor to allow for direct inclusion of programs or declarations, macros, and static conditionals. Third, $P4_{16}$ introduced the notion of an *architecture.* The architecture describes common capabilities of a network switch. The architecture allows developers to separate target-specific details from the core logic of their program. The architecture also allows developers to create different instances of a forwarding device with different implementation details.

Unfortunately, these features are not sufficient to extend the base P4 program with the GENEVE protocol. We could use file inclusion (i.e., CPP macros) to include the new header definition. But, our example also requires that we re-write the parser and modify the program control flow—which is not possible using the existing support for modular development.

The P4 Language Design Working Group has done work to add a certain degree of modularity in the language, but finding a solution to the problem is not straightforward. Defining new language constructs is relatively easy. What remains fundamentally difficult is mapping such constructs onto some specific hardware targets.

In a typical software development environment, systems might come in binary form but users would still be able to augment the code with static and dynamic linking. However, such an approach would not easily work for ASICs or FPGAs. On a general-purpose instruction set architecture such as x86, the output of the compilation is an object file that contains data as well as code objects identified by symbols. Some of the instructions may reference *external* symbols (functions, global objects, etc.) defined in other object files. Later a static or dynamic *linker* loads all the necessary object files and links every symbol to its implementation, resolving each reference to a symbol into a concrete memory address. This creates the final executable. Then, at runtime, the stack-based call and return mechanism implemented by the CPU ultimately realizes the seamless integration of modular components.

The problem is that these same mechanisms can not be mapped onto ASICs where the logic connections between code and data elements are hard-wired as a result of the compilation of the P4 source. In other words, the notion of a linker and a run-time stack for P4 is not feasible. This leads us to a different approach, one that works via source-code manipulation.

2.2 Aspect-Oriented Programming in P4?

Today, one of the most popular paradigms for modular programming is object-oriented programming (OOP). There are many variations—from Javascript's prototype based inheritance to C++, Java, and Python's class based approach—but all of them are based on the concept of an "object" with data fields and methods. This abstraction enables many of the desirable features of modularity, such as encapsulation, composition, inheritance, and delegation.

OOP is not without limitations. Given a break-down of functionality and data into objects, there might still be common "aspects" of the behavior of many of those objects that could not be easily modularized into one or more separate objects. To address this problem, Gregor Kiczales and colleagues at Xerox Park proposed the concept of aspect-oriented programming (AOP) [19]. Aspect-oriented programming provides a way to modularize cross-cutting concerns, that is, units of functionality that span multiple classes, methods, procedures, etc. The basic idea of AOP is that developers can modify the behavior of some base code by adding some code called *advice*. An advice may take control at specific *join points* in the base code, for example before or after the invocation of a function. A *pointcut* is an expression that specifies one or more such join points for a given advice. The program that joins the base program with the advice at the specified join points is referred to as a *weaver*.

In this paper, we propose to use techniques from AOP to provide modularity to P4. In the programming languages community, this is likely a somewhat controversial proposition. One of the major criticisms of AOP is that the implicit transfer of control from the base program to the advice code makes the program flow hard to understand. This is, essentially, the same argument that Dijkstra famously made against GOTO statements.

Given this valid criticism, it is natural to ask, "should AOP be used in P4?" We argue that what we propose *uses* AOP techniques, but is in fact a more *principled and controlled approach* that provides proper code isolation and sequencing. In AOP, the extension code would define an "aspect" consisting of both an advice, meaning the code of the extension, and a pointcut, meaning the definition of the points in which that code takes control. By contrast, in our design of P4 Weaver there are two clearly distinct roles, namely the base code and the extension, with a proper separation of concerns. In particular, it is the base code that defines appropriate pointcuts through which the extension code can take control and also how and where control returns to the base code. In fact, the base code can constrain the extension code within a strict sequencing or more generally limit the use of both control and data structures of the base code.

3 P4 WEAVER DESIGN

We now detail the design of our P4 Weaver. We start by discussing the type of changes we propose to support with P4 Weaver. P4 Weaver realizes such changes by integrating the extension code within the base code. This integration is controlled by annotations in both the extension and base code. We present the syntax and semantics of these annotations. Then we present the design of the weaver, including some technical details of the weaving process.

3.1 Modular Extensions in P4

P4 Weaver is intended to support customers in customizing the behavior of their switches. However, a customer may not make arbitrary modifications, as these could introduce potentially unsafe behavior. Below, we discuss the types of code modifications that are permissible.

Only Additive Modifications. P4 Weaver only allows users to *add* functionality to a base program. This is because removing or changing existing functionalities and parameters can violate the contract between the control plane and the data plane (e.g., the PD-API in case of P4 on Tofino).

As a concrete example, a user might want to remove an unused protocol. If they know that they will never use IPv6 in their network, they might want to remove the related table to recover switch resources. However, the control plane may use or otherwise refer to that data-plane table, which would then break the contract provided by the API.

Even simple modifications can be problematic. A user might want to route packets also based on the value of the differentiated services field (DS). To do that, a user could add a key to the IPv4 forwarding table to represent the DS field, so that different entries for the same destination could be associated with different DS values. However, this change would break the operating system because the operating system would not know of the additional key in the table.

Reconciling violations of a data-plane API is possible, but requires coordinated changes in the control plane and possibly the switch operating system. In practice, this is often done using preprocessor flags to selectively include feature-specific definitions in P4 programs [17], which in turn are carefully coupled with a corresponding configuration of the control-plane code, possibly through the same preprocessor flags. Such coordinated feature selection is perfectly compatible with P4 Weaver.

However, P4 Weaver is intended to support modularity and a separation of concerns between the base code and extension code. We assume that the developer of the data-plane extension would not have access to the control-plane or operating-system code, and therefore we choose to prohibit altogether changes that may create inconsistencies. In this respect, our design is quite conservative. We know that if no tables and corresponding actions are removed or changed, the API stays consistent with the control plane and the operating system. One could imagine a more permissive design in which users may modify a base program but only to a limited extent so as to preserve correctness. For example, it might be possible to safely reduce the size of a table. However, determining which modifications are safe requires a careful analysis of the expectations of the control plane over the data plane parameters (such as table sizes) which are platform dependent.

Possible Additions. Even considering only additive changes, our design limits the type of such additions. The choice of allowable additions stems from a careful analysis of the P4 language specification [25]. P4 Weaver allows users to add new constants, errors, actions, parser state changes, type declarations (e.g., headers, meta data), control constructs (e.g., tables), and functions. Table 1 details the addition of each language construct.

Construct	Description
constant	Declaration of a new constant value
error	Declaration of a new error type
function	Declaration of a new function
type	Declaration of a new derived type or use of *typedef* to define an alias for another type declaration. This addition covers the addition of a *new header,* since header definitions and structs are considered type declarations.
parser	Extension of the finite state machine of the parser: addition of a new parser state together with the connections to and from this new state.
control	Declaration of a new control block, or addition of modules to the switch's previously defined *ingress* and *egress* blocks. This addition covers the addition of a *new table,* since control blocks include table definitions, constant values, variables and action declarations.
action	Declaration of a new action outside of a control block. The action then can be invoked from all the control blocks.

Table 1: Additive changes supported by P4 Weaver

P4 Weaver does not allow users to add new *match kind* or *extern* declarations. These declarations depend on the architecture, and therefore are only valid and meaningful if supplied by the vendor. Users are also not allowed to instantiate a new switch. Such a change does not make sense in our context and would likely violate the control-plane API.

3.2 Annotations in the Base Code

The vendor annotates the base code to define allowable extensions for the customer. In particular, annotations in the base code (i) control the visibility of header types and identifiers, (ii) indicate places where new parser state transitions may be added, and (iii) indicate places in the source code where the composition of tables can be modified. Each annotation is associated with a specific language construct and includes a set of attributes consisting of key-value pairs.

Listing 1 exemplifies the annotations that P4 Weaver supports for the base code. We now document each annotation type in detail.

Header Annotations. The *@Header* annotation is associated with a header definition and is defined by the *access* attribute. This annotation specifies the permission granted to the extension code in accessing the referenced header. Possible values for the *access* attribute are *name*, *read,* and *write.* A *name* access means that the extension code may validate the existence of this header (using the `isValid()` method); *read* means that the extension code may read the value of all the fields in this header; *read* implies *name*; *write* means that the customer code may modify the value of all the fields in this header; *write* implies *read* and *name.*

```
@Header (access="read")
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}
// ...
@State (enabled="true",
  positions="before,after",
  name="Parser_UDP")
state parse_udp {
  packet.extract(hdr.udp);
  transition accept;
}
// ...
@Table (enabled="true",
  positions="before/after/hit/miss",
  name="Ingress_IPv4_Forward")
table ipv4_forward {
  key = {
    meta.routing_metadata.nhgrp : exact;
  }
  actions = {pkt_send; drop; }
  size = 64;
  default_action = drop;
}
```

Listing 1: Annotations in the base code for the GTP use case (described in Section 6). IPv4 forwarding table. According to the values of `valid_positions`, the customer can inject new code, *before, after* or in case of *miss* or *hit* of this table. As in *@StateConfig*, the vendor can use an alias, to avoid revealing the real name of the table.

State Annotations. The *@State* annotation is associated with a parser state and is defined by attributes *name, enabled,* and *positions.* This annotation allows the extension code to connect the annotated parser state with a new parser state of the extension. The *name* attribute defines an alias that the extension code must use to refer to the annotated parser state. The *enabled* attribute simply makes this annotation active or dormant. The value of the *positions* attribute may be *after*, *before*, or both, indicating that the extension may add an outgoing branch, an incoming branch, or both, respectively.

Table Annotations. The *@Table* annotation is associated with a table definition and is defined by attributes *name, enabled,* and *positions.* This annotation allows a user to add code before or after the application of the annotated table. The *name* attribute defines an alias that the extension code must use to refer to the annotated table. The *enabled* attribute simply makes this annotation active or dormant. The value of the *positions* attribute may be *before*, *after*, *hit*, *miss*, or any combination of these values. These values indicate that the extension code may take control before, after, in case of a miss (no match), or hit (match), or in any combination of these positions.

3.3 Annotations in the Extension Code

A developer creates an extension by writing P4 code and by annotating that code to properly connect it to the base code. The annotations refer to the headers, parser states, and tables of the base code as defined by the annotations in the base code. In other words, the annotations in the base code define the interface between the extension and the base code.

Annotations in the extension code: (i) define new state-transitions in the parser state machine, (ii) attach control logic to, or return control to the base code, and (iii) modify the deparser to emit any new headers. Listing 2 shows an example of the annotations that P4 Weaver supports.

```
@Parser(after="Parser_UDP",
  condition="hdr.udp.dstPort == GTP_UDP_PORT")
state parse_gtp {
  packet.extract(hdr.gtp_common);
  transition select(hdr.gtp_common.version, hdr.gtp_common.tFlag) {
    (1,0) : parse_teid;
    (1,1) : parse_teid;
    (2,1) : parse_teid;
    (2,0) : parse_gtpv2;
    default : accept;
  }
}
// ...
@Control(table="Ingress_IPv4_Forward",
   condition="before")
control GTP_contrl(inout my_headers hdr,
      inout my_metadata meta,
      inout standard_metadata_t standard_metadata) {

  meter(256, MeterType.bytes) teid_meters;

  action drop() {
    mark_to_drop(standard_metadata);
  }
  ...
}
// ...
@Deparser(after="hdr.udp")
control MyDeparser(packet_out packet, in headers hdr) {
  apply {
    packet.emit(hdr.gtp_common);
    packet.emit(hdr.gtp_teid);
    packet.emit(hdr.inner_ipv4);
    packet.emit(hdr.inner_icmp);
    packet.emit(hdr.inner_udp);
  }
}
```

Listing 2: *@Parser* annotation that adds the GTP parser state after the state that processes the UDP header. In this example, `condition` specifies that only those UDP packets with destination port equal to `GTP_UDP_PORT` are considered to be GTP packets.

Notice that apart from the annotations, the extension code is pure P4 code. In fact, only a few elements of that code need to be annotated, and many other additions, of the kinds listed in Table 1, are simply defined by the structural elements of the extension code. For example, the code may define new headers and new tables as usual.

We now document each type of annotation in detail.

Parser State Annotations. The *@Parser* annotation is associated with a parser state and is defined by attributes *after, before,* and *condition.* This annotation connects one or more source states in the base code to this annotated state. Attributes *after* and *before* identify one or more source states: *after* names the source state directly as shown in the example of Listing 2, which creates a branch from the `Parser_UDP` state in the base code to the new `parse_gtp` state defined in the extension code. Notice that `Parser_UDP` is the alias for the state `parse_udp` as annotated in the base code shown in Listing 1. Instead, *before* identifies one or more source states indirectly. For example, `before="Parser_UDP"` indicates as sources the states that connect *into* state `Parser_UDP`, which in this case might be states `IPv4` and `IPv6`. The *condition* attribute defines the branching condition to go from the source states to the new, annotated state.

Control Block Annotations. The *@Control* annotation is associated with a control block and is defined by attributes *table* and *condition.* This annotation attaches the annotated control block to the application of a table in the base code. Attribute *table* identifies the table. Attribute *condition* may be either *before, after, miss,* or *hit,* which determines whether the annotated control block takes control before, after, or only in case of a miss or a hit, respectively. Notice that *after* means in case of *either* miss or hit.

Deparser Annotations. The *@Deparser* annotation is associated with a deparser control block and is defined by attributes *after* and *before.* This annotation determines the sequencing of the annotated deparser as part of the overall deparser. More specifically, this annotation determines the position of the headers indicated in the annotated deparser block within the deparser sequence in the base code. The *after* and *before* attributes refer to the names of the headers that immediately precede and follow the annotated headers, respectively. For example, the *@Deparser* annotation in Listing 2 adds the GTP headers immediately after UDP.

3.4 Weaver

The weaving process takes the annotated base code and the annotated extension code, and combines them into a single P4 source file. As a first step, the weaver parses the base code and the extension code and builds an abstract syntax tree (AST) for each. The weaving then proceeds as a merge operation on the two trees.

For some parts of the code, such as constants, errors, functions, types (including header definitions), and actions, the merge operation is very straightforward. The output is simply the union of the two parts. More specifically, if subtrees $t_1^b, t_2^b, \ldots$ and $t_1^e, t_2^e, \ldots$ are the two sets of subtrees of the base and extension ASTs, respectively, representing, say, the header definitions, then the output AST will contain a concatenation of all those subtrees: $t_1^b, t_2^b, \ldots t_1^e, t_2^e, \ldots$.

The other parts, namely parser and control (see Table 1), require a more involved merging process. In both cases, the merging process requires changes in the base code and, to a lesser extent, in the extension code. Also, other general processing steps are needed to properly handle identifiers and to check that the merging is consistent with the policies defined within the base code. We now describe these processing steps at least at a high level.

Code Generation for the Parser. The combined parser consists of the finite state machine of the parser of the base code properly extended with the new states defined by the extension code. Therefore, the crucial change amounts to adding the necessary transitions between source states (base) and new states (extension) within the `transition` blocks of the source states.

Algorithm 1 shows the process to merge the parser of the extension into the parser of the base code. Lines 8 and 13 ensure that the extension is allowed according to the access-control policies determined by the annotations in the base code and similarly line 5 ensures no policy violation exists in accessing base headers. Then, the merge procedure branches depending on whether the parser extension is to be inserted *after* or *before* a particular parser state in the base code.

Algorithm 1 `WeaveParser` function

```
Input: base_AST, ext_AST  // AST of base and extension code
Output: merged_AST or error message
 1: merged_AST $\leftarrow$ copy of base_AST
 2: Try:
 3:   for all parser states ext_s in ext_AST do
 4:     h $\leftarrow$ headers accessed in ext_s
 5:     assert h are accessible according to base_AST
 6:     if ext_s.after then
 7:       base_s $\leftarrow$ ext_s.after  // base-code parser state
 8:       assert “after” $\in$ base_s.positions
 9:       cond $\leftarrow$ ext_s.condition
10:       update base_s.select with cond and ext_s
11:     else if ext_s.before then
12:       base_s $\leftarrow$ ext_s.before  // base-code parser state
13:       assert “before” $\in$ base_s.positions
14:       for all parser states s in base_AST do
15:         if s has a transition to base_s then
16:           update that transition to ext_s instead
17:         end if
18:       end for
19:     end if
20:     append ext_s to merged_AST
21:   end for
22:   return merged_AST
23: Catch: assertion violations
24: return assertion violations
```

In the *after* case, the `select` statement in the parser state of the base code (*base_s*) must be modified to include a transition to the parser state of the extension (line 10). At a high-level, there are two cases: if the transition condition (line 9) refers to the same selection variable used in base-code state, then P4 Weaver adds the given condition as the first `case` statement within the selection. Otherwise, if the transition condition refers to a different variable, then P4 Weaver adds the new condition variable to the `select` statement of the base-code state, and correspondingly adds the selection expressions for what is now a *pair* of keys for all the `case` statements in that state. If the select statement already refers to two keys, then P4 Weaver changes it to three, etc.

In the *before* case, P4 Weaver identifies the target base-code parser state (*base_s*) and all the other states in the base-code parser that have a transition into that target state. For each of those states, P4 Weaver changes their transition into the target state to instead go to the state of the extension code (see line 14).

Code Generation for Control Blocks. Merging control blocks means inserting into the base code the application of the control blocks defined and annotated within the extension code. The annotation of the extension control blocks determines the target table and also a condition. In the simplest case, the condition is either *before* or *after,* in which case P4 Weaver simply adds an invocation of the extension control block before or after the application of the target table in the same code-block within the base code. For example, if the base code applies `IP` table within, say, an `if` statement (i.e., the *then* code-block), and the extension code defines a control block `B` annotated as *after* table `IP`, then P4 Weaver inserts the statement `B.apply(...)` right after the statement `IP.apply()` within the same `if` statement.

Notice that in generating the `B.apply(...)` statement, P4 Weaver must also define the proper parameters for the `apply` call. This is because the extension code (the `B` block) might refer to headers that are defined in the base code but not in the extension code. Thus, P4 Weaver outputs a completely new `struct` declaration that contains the union of the headers available from the input parameters of the control block in both the base and extension code.

The *miss* and *hit* conditions are a bit more complex. In these cases, P4 Weaver must generate code to store the result of the application of the associated table and then wrap the `B.apply(...)` call in a corresponding `if` statement. This may be necessary because the P4 compiler does not allow multiple applications of the same table.

Handling Identifiers. Even with P4 Weaver's conservative approach to modifications, introducing new code has the potential to introduce unexpected behaviors and bugs. The extension code might inadvertedly use identifiers that are already used in the base code. To avoid duplicate identifiers, P4 Weaver generates a unique alias for all the identifiers defined in the extension code. This solves the problem without revealing the presence of any specific identifier in the base code. The aliases are defined using a special prefix in such a way that they can be distinguished as belonging to the extension code, and furthermore that they can be mapped to the corresponding original identifiers in the extension code. This allows the weaver to correctly refer to the original extension code when reporting feedback to the customer (for example, for error messages). We discuss the feedback channels in greater detail in Section 4.

Checking for Unauthorized Access to the Packet. The extension code might erroneously read from or write into headers that are not declared accessible by the vendor. P4 Weaver parses and analyzes the extension code, and therefore can identify the *lvalue* and *rvalue* expressions in the extension code to then determine which objects (e.g., headers) are accessed by the extension code. P4 Weaver then checks that the extension code would not violate the access policies annotated within the base code. As an example, see lines 4 and 5 of Algorithm 1.

Checking for Invalid Logic Paths. The extension code must also be consistent with the sequencing of the various processing phases as determined by the base code. For example, the customer might extend the finite-state machine of the parser by adding new code to the control blocks. However, those additions have to be explicitly allowed with annotations within the base code. In general, P4 Weaver makes sure that the flow of the program will always satisfy the constraints of the vendor.

4 ARCHITECTURE

The design of the annotations and weaver mechanism described above provides a principled and controlled approach to ensuring code isolation and enabling incremental development. However, this design does not in and of itself hide the base code from the customer. Nor does it offer any support to the customer in developing their extension code.

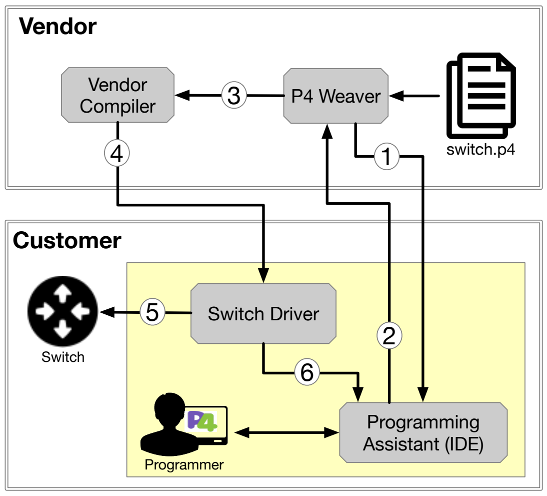

Figure 1: The components of the P4 Weaver client-server architecture along with the communication among them. (1) The IDE receives the interface of the base code and other diagnostics, and uses that to guide and support the developer. (2) The extension code is merged with the base code (*switch.p4*) by the weaver, and then (3) compiled to obtain a binary file that (4) is then sent to the client. If the compilation is successful, then (5) the binary image can be deployed onto the switch and/or (6) more diagnostics and statistics can be sent back to the developer.

For these requirements—i.e., to limit the visibility of the base code and to support developers—P4 Weaver uses a client-server architecture (Figure 1) that realizes its full functionality without giving the customer direct access to the vendor base-program, and at the same time providing focused support for the developer.

Server Side. The base P4 program is stored on a server that may be administered by the vendor or a trusted third party. In the figure, the base code is named *switch.p4*. Thus, the server side is responsible for all the activities that require access to the base code. In particular, the server side runs a compiler for the annotations in the base code, the weaver, and the vendor compiler.

The annotation compiler reads and extracts from the base code and its annotations the information needed by the developer. This compiler is part of the P4 Weaver system and does not need to be able to generate code for the vendor target architecture (or any other real target). In our implementation this is a modified version of the open-source P4 compiler that produces structured information on the interface of the base code in the form of a JSON file.

The weaver then merges the base code with the extension code also performing all the necessary validity checks as described in Section 3. The weaver takes the base code from the local repository, the extension code from the client side, and may also return diagnostic information back to the client.

The vendor compiler takes the merged P4 program and produces a binary image, possibly including diagnostic information, which is then sent over to the client.

The client-side component of P4 Weaver is a programming assistant integrated into an IDE. The assistant receives information about the base program and exposes it to the customer. In our implementation, this information is provided via a GUI in the IDE. The IDE then supports the developer by providing autocompletion, basic checks, and error reporting using a local P4 compiler. The IDE also relays diagnostic information from the vendor compiler on the server side.

The workflow within this client-server architecture is as follows. First, the P4 compiler on the server side processes the base code and extracts its structured interface (e.g., public headers, parser states, etc.) and communicates this information to the programming assistant on the client side (arrow 1). Second, the customer uses the programming assistant (IDE) to code their extensions. When the developer is ready to compile their code, the client software sends the extension code to the server component (arrow 2).

The weaver program then merges the extension and base code, combining them into a single P4 switch program. If everything goes well, the server component then invokes the vendor-supplied P4 compiler on the merged program (arrow 3). For example, for Tofino-based switches, this compiler would be the Barefoot Networks compiler provided with their SDE. If the weaver detects errors (e.g., policy violations) the server component then relays those error messages back to the IDE on the client side (arrow 1). Once the compilation is complete, the server component sends the image back to the client (arrow 4). Finally, on the client side, the switch driver installs the new binary files on the switch device (arrow 5) and/or may return diagnostic to the developer (arrow 6).

Note that this client-server architecture and this data flow allows for the best level of privacy protection afforded by the vendor compiler and hardware. However, P4 Weaver is not intended to protect the base code in a cryptographic sense, nor to protect the functionality of the switch against the intentions of the customer. With or without P4 Weaver, having physical access to the device and being able to inspect registers and table entries, the customer might still be able to reverse engineer some information about the vendor code, in particular information on resource usage. Also, it is of course possible for a customer to disrupt the intended functionality of the switch by misconfiguring or modifying table entries. Similarly, a P4 extension can explicitly drop all incoming traffic. P4 Weaver is not intended to prevent such scenarios, also because they might be exactly what the customer wants to obtain.

5 LIMITATIONS AND FUTURE WORK

Below, we discuss some of the limitations of the current P4 Weaver design, and directions for future work.

Incremental Control Plane Development. P4 Weaver is designed to support incremental development of the data plane program. As we already discussed in Section 3, P4 Weaver only allows additive modifications, so as not to break the API between the data plane and the control plane. In the future, it would be useful to extend P4 Weaver with support for incremental extensions to the control plane component. However, this is a challenging problem. There are several layers of abstraction in the switch software stack (e.g., hardware abstraction, generic switch hardware abstraction, switch state abstraction, etc.), and each of these layers has complex internal and external (intra-layer) dependencies. Moreover, none of these abstractions are truly program independent. For example, even with the P4Runtime API, which is the control plane specification for a P4-programmable device, although the API calls are program independent, the parameters are not.

Resource Allocation. One of the biggest challenges for P4 developers is managing resource consumption. Adding relatively minor changes to an existing program can alter the table dependency graph and dramatically impact how a data plane program is laid out in memory. In the future, we plan to add some support to P4 Weaver to help developers with the resource allocation challenge. One approach would be to give estimates of resource usage via static analysis of the extension code. Another approach would be to be less conservative and allow users to modify resource allocation (e.g., reduce the number of table entries) so as to free up resources without breaking the API contract.

Dynamic Access Control. Currently, access to the header fields is considered to be static and independent of the type of extension applied by the customer. However, one could imagine a more flexible design. For example, bounding access to the vendor TCP header definition in case of a certain destination port address. Even though some of these features can be expressed with more complex annotations, not all of them can be realized due to limitations in the P4 language itself.

6 EVALUATION

To demonstrate the utility of the annotations and system architecture, we used P4 Weaver to implement a series of practical examples in which we extend a switch with real features and protocols that are of interest to network operators. We report on the details of the implementations and our experience in these use cases. As a sanity check, we also report on the running time for the weaver tool, which is negligible. Overall, we found P4 Weaver to be an effective incremental development environment for P4.

6.1 Prototype Implementation

We have implemented a prototype of P4 Weaver. All source code is publicly available with an open-source license[2]. The P4 Weaver annotations use P4's existing support for annotations, allowing us to extend the language without changing the grammar [26]. The prototype implementation includes two major components: the client and server, which we discuss in more detail below.

P4 Weaver Client. The client side component is implemented as a plugin for the Microsoft VSCode IDE. The plugin is written in TypeScript, a type-safe extension to Javascript. The client includes a P4 parser, written using the ANTLR parser generator tool [1]. The client-side parser allows the IDE to check for syntactic errors and perform some basic semantic checks, without communicating to the server.

P4 Weaver Server. The server-side software is composed of several independent modules. The main server interface is a REST Web API implemented with Ruby on Rails. The web server invokes the P4 Weaver core, which merges the base and extension code. The weaver module is written in Java and includes a P4 parser, again based on ANTLR, which constructs the two ASTs that are then merged into a single program. To compile the merged program, the server invokes an external P4 compiler. P4 Weaver is agnostic to which P4 compiler is used. We have used P4 Weaver with both the open-source P4c compiler that targets the behavioral model software switch, and the proprietary P4 compiler included in the Barefoot Networks SDE.

[2]https://github.com/usi-systems/p4weaver

6.2 RTP Timestamp Switching

Real-time Transport Protocol (RTP) timestamp switching is a functionality not offered by any commercial switch—but actually needed by the broadcast media industry—and is perfect in showing the benefits stemming from the hybrid deployment of a P4 programmable switch.

Media flows transported within multicast IP packets must be switched based on the value of the timestamp contained in the RTP header and the multicast destination address in the IP header. A device implementing timestamp switching must rewrite the destination IP address based on the value of the timestamp, so that different portions of a flow get delivered to different sets of destinations by the subsequent switches. Once the destination address is rewritten, the switch must be able to properly replicate and forward the resulting IP packet based on the new IP address, possibly using routing information collected by a multicast routing protocol running in the control plane.

If a P4 programmable device is deployed in whitebox mode, the user must write the P4 code implementing not only timestamp switching, but also multicast IP packet forwarding. Moreover, when developing the control plane, the user must implement not only the control for timestamp switching, but also for IP multicast, which might include support for multicast routing protocols.

When deploying a P4 programmable device in hybrid mode, the incremental programmer only needs to implement the timestamp switching function, while multicast IP packet forwarding is already implemented within the base code and its control logic is readily available within the network operating system.

To add support for RTP, a user of P4 Weaver must implement the RTP header; specify how the RTP header is parsed, branching on the parsing of a UDP header; add a new table that uses the destination IP address and RTP timestamp as keys; define a new action to modify the destination IP address in case of match; and indicate where the new table should be matched to the pipeline.

6.3 GPRS Tunneling Protocol (GTP)

The General Packet Radio Service (GPRS) is used by some mobile carriers for communication between cell tower base stations and mobile phones. For some of these sessions, cell tower base station *A* with IPv4 address *X* will have an IPv4 tunnel configured between itself and another network device (e.g., perhaps another cell tower base station *B* with IPv4 address *Y*). This enables the mobile phone data to be carried over an IPv4 network.

GTP has a signaling component for establishing new sessions; we will not discuss it here because it is implemented in the control plane. Once sessions are established, there could be thousands of individual handset-to-other-IP-addressable-device sessions sending data simultaneously, all having the same Ethernet/IPv4/UDP/GTP encapsulation.

The P4$_{16}$ header type definition `gtp_v1_t` in Listing 3 defines the format of a GTP version 1 header. There are optional fields in a GTP version 1 header that can appear after the `teid` field. For this use case, we only need the portion of the GTP header defined here. We do not need access to any optional field or headers after the GTP header.

```
header gtp_v1_t {
    bit<3>  version;
    bit<1>  protocol_type;
    bit<1>  reserved1;
    bit<1>  extension_header_flag;
    bit<1>  sequence_number_flag;
    bit<1>  npdu_number_flag;
    bit<8>  message_type;
    bit<16> message_length;
    bit<32> teid;   /* Tunnel Endpoint ID */
}
```

Listing 3: GTP Header

It is fairly common in many network switches to extract fields like Ethernet MAC addresses, IPv4 addresses, IPv4 protocol, and/or layer-4 ports, and then hash those fields to select among one of several equal-cost paths through the network (ECMP) as a form of load balancing.

For all GTP version 1 encapsulated traffic from IPv4 address X to IPv4 address Y, all of those mentioned fields in the headers up to and including the UDP header will have the same field values, and thus existing ECMP implementations will send the packets for many otherwise independent GTP sessions over the same path. While the data rates of individual mobile phone data sessions might be fairly low, the aggregate rate of thousands of such data sessions can be a significant fraction of the network link capacities.

In this use case, we implement logic to recognize packets that contain a GTP version 1 header, extract the Tunnel Endpoint Identifier (TEID) field, and include the `teid` field in the hash calculation used for ECMP path selection.

To correctly recognize which packets contain a valid GTP version 1 header, the program must first check that there is an IPv4 header with fragment offset equal to 0, followed by a UDP header with destination port 2152 decimal. The program then must check that the GTP `version` field is set to 1. If the version field is not 1, then the program should not extract a GTP version 1 header from the packet. This is implemented using the lookahead feature of P4$_{16}$ parsers.

6.4 In-band Network Telemetry (INT)

One recent approach to network monitoring is to collect, directly within the packet, fine grained statistics on every switch a packet passes through [16, 18]. A "sink" switch, possibly located at the egress of the monitored network, extracts the data collected in the packet and sends it to an analytics system such as Barefoot Networks Deep Insight [5] or Broadcom's BroadView Analytics [9] for processing.

We used P4 Weaver to add support for providing telemetry data by populating fields in the in-band network telemetry (INT) header. This required new P4 code that defines the INT header and how it is parsed; adding actions to populate the INT header fields with the appropriate data; and a table to check if the switch is the sink; if it is, the program removes the INT header stack and forwards the information out to the collector in a new report packet. INT header processing was our most complex use case, requiring 813 new lines of P4 code and 13 client-side annotations.

Example	LoC	Annotations	Weaving (ms)	P4c (ms)
RTP	76	3	608 ± 48	3433 ± 63
GTP	263	3	641 ± 43	4628 ± 112
INT	813	13	657 ± 65	9402 ± 615

Table 2: A comparison among the running time of P4 Weaver for different case studies. It can be seen that running time is slightly longer for larger extension and higher number of annotations, while being always insignificant relative to the compilation time using Barefoot Networks' P4c. In all the cases, the base LoC is 537 and it contains 20 annotations.

6.5 Running Time

Table 2 shows the number of lines of code and number of annotations for the base and extension programs from all three of our case studies. It also shows the running time for weaving the base and extension programs, which are computed as an average of 15 independent runs. We also report the running time for the Barefoot Networks' P4c compiler included in SDE 9.0.0 for compiling the merged programs.

We ran all experiments on a Ubuntu 18.04.1 LTS virtual machine with a 2.8GHz Intel i7 processor with 32KB L1 data, 32KB L1 instruction, 256KB L2 (per core), 6MB L3 (shared) cache, and 8GB of main memory. The processor has 2 cores and our implementation is parallelized to use both.

These experiments show that the overhead of P4 Weaver is negligible when compared to overall compilation time.

7 RELATED WORK

Incremental Programming for P4. P4 Weaver builds on earlier work in daPIPE [3]. Both projects use a client/server architecture to control access to the vendor code. Moreover, both projects identify "hook points" in P4 code where user and vendor code can be merged. P4 Weaver offers a more principled and controlled approach to providing code isolation and sequencing. With daPIPE, the hook points are exposed through a GUI. The P4 Weaver development environment is implemented as a plug-in to an existing IDE. P4-Ansible [21] is a tool to merge customer and vendor P4 programs, but we were unable to find additional details beyond the feature table on the product website. P4 Visor [35] merges multiple independent P4 programs deployed on the same device to optimize resource consumption. It provides algorithms to efficiently merge two P4 programs, but does not addressing the issue of modular development.

Data Plane Programming Languages. Several recent projects have proposed domain-specific languages for dataplane programming [8, 10, 31]. Notably, μP4 [32] and Lyra [15] allow for portable and modular development. With μP4, a large P4 program is decomposed into smaller, target-agnostic, reusable modules. Lyra targets data plane programming across multiple heterogeneous programmable switches. Both μP4 and Lyra require that a vendor rewrite their existing data plane codebase using the new language. By contrast, P4 Weaver uses annotations that must be applied to existing P4 code. Moreover, P4 Weaver provides fine-grained access control on the variables and code blocks which safeguards the base functionality of the switch.

P4 Compilers. Given the significant interest in P4 as a development platform, there are several efforts underway to implement P4 compilers and tools for a variety of hardware. P4.org provides a reference compiler [6] with back-ends that target the behavioral model software switch and Berkeley Packet Filters. Barefoot Networks develops a compiler that targets their Tofino chips [4]. There are a few projects that target FPGAs, including SDNet from Xilinx [34] and P4FPGA [33]. P4c [20] translates P4 to DPDK [13]. PISCES [30] targets a modified version of Open vSwitch [24]. P4gpu [22] generates code for GPUs. Open-NFP [23] supports a compiler that translates P4 to network processing units (NPUs). None of these projects offer support for modular development.

8 CONCLUSION

This paper has presented P4 Weaver, a development environment to enable incremental programming on P4-based switches. The design of P4 Weaver is inspired by prior work on Aspect-Oriented programming. But, in contrast to traditional AOP, P4 Weaver adopts a principled and controlled approach to code isolation, by assigning clearly distinct roles, vendor and customer, with a proper separation of concerns. Moreover, P4 Weaver provides controlled visibility via a client-server architecture that prevents physical access to the vendor base-program. Using three real-world inspired case studies, we have demonstrated how P4 Weaver allows users to address their use cases in the most suitable way by adding their own custom features, while also leveraging typical switching and routing functions. Overall, P4 Weaver enables the incremental programmer to leverage the base code without needing to cope with its complexity and while keeping the base code confidential.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Jedidiah McClurg, who helped us improve this paper. We also thank Andy Fingerhut who provided us with the GPRS example and technical feedback. We acknowledge the support from the Swiss National Science Foundation (SNF) (project 407540_167173) and an award from Cisco Systems.

REFERENCES

[1] ANTLR 2021. ANTLR: ANother Tool for Language Recognition. https://www.antlr.org.
[2] Arista 7170 Series [n.d.]. Arista 7170 Series. https://www.arista.com/en/products/7170-series.
[3] Mario Baldi. 2019. daPIPE a Data Plane Incremental Programming Environment. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '19)*. IEEE, 1–6.
[4] barefoot [n.d.]. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.
[5] Barefoot Networks Deep Insight 2021. Barefoot Networks Deep Insight. https://www.barefootnetworks.com/products/brief-deep-insight/.
[6] Behavioral Model (bmv2) 2021. The P4 Reference Switch. https://github.com/p4lang/behavioral-model.
[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[8] G. Brebner and Weirong Jiang. 2014. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro* 34 (Jan. 2014), 8–18.
[9] Broadcom BroadView Analytics 2021. Broadcom BroadView Analytics. https://www.broadcom.com/products/ethernet-connectivity/software/broadview-analytics.
[10] broadcom-csg-mktg @github. 2021. Introduction to NPLSpec. https://github.com/nplang/NPL-Spec.
[11] Cisco Nexus 3400 series [n.d.]. Cisco Nexus 3400 series. https://www.cisco.com/c/en/us/products/switches/nexus-3000-series-switches/models-comparison.html#~tab-nexus3400.
[12] Cisco Silicon One [n.d.]. Cisco Silicon One. https://www.cisco.com/c/en/us/solutions/silicon-one.html.
[13] DPDK [n.d.]. DPDK. http://dpdk.org/.
[14] Edgecore Wedge100BF-32X [n.d.]. Edgecore Wedge100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.
[15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. 435–450.
[16] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. 71–85.
[17] header.p4 2021. header.p4. https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/include/header.p4.
[18] int 2021. Inband Network Telemetry (INT). https://github.com/p4lang/p4factory/tree/master/apps/int.
[19] Gregor Kiczales et al. 1997. Aspect-Oriented Programming. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '97)*. 220–242.
[20] Sándor Laki. 2016. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. In *P4 Workshop (P4WS)*.
[21] MNK Lans and Consulting. 2021. P4-Ansible. https://mnkcg.com/products/p4-ansible.
[22] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation) (NSDI '16)*. 31–44.
[23] Open-NFP [n.d.]. Open-NFP. http://open-nfp.org/.
[24] Open vSwitch [n.d.]. Open vSwitch. http://www.openvswitch.org.
[25] P4 16 Language Specification 2021. P4_{16} Language Specification Version 1.1.0. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.
[26] p4org [n.d.]. P4 Language Consortium. https://p4.org.
[27] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.
[28] Pensando Distributed Services for Cloud Providers [n.d.]. Pensando Distributed Services for Cloud Providers. https://pensando.io/wp-content/uploads/2020/03/Pensando-Distributed-Services-for-Cloud-Providers.pdf.
[29] Pensando Distributed Services for the Enterprise [n.d.]. Pensando Distributed Services for the Enterprise. https://pensando.io/wp-content/uploads/2020/03/Pensando-Distributed-Services-for-the-Enterprise.pdf.
[30] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 525–538.
[31] Haoyu Song. 2013. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. 127–132.
[32] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing dataplane programs with μP4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. 329–343.
[33] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. 122–135.

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

[35] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, 98–111.