Composing Dataplane Programs with μP4

Hardik Soni* Myriana Rifai† Praveen Kumar* Ryan Doenges* Nate Foster*

*Cornell University †Nokia Bell Labs

Abstract

Dataplane languages like P4 enable flexible and efficient packet-processing using domain-specific primitives such as programmable parsers and match-action tables. Unfortunately, P4 programs tend to be monolithic and tightly coupled to the hardware architecture, which makes it hard to write programs in a portable and modular way—e.g., by composing reusable libraries of standard protocols.

To address this challenge, we present the design and implementation of a novel framework (μP4) comprising a lightweight *logical architecture* that abstracts away from the structure of the underlying hardware pipelines and naturally supports powerful forms of program composition. Using examples, we show how μP4 enables modular programming. We present a prototype of the μP4 compiler that generates code for multiple lower-level architectures, including Barefoot's Tofino Native Architecture. We evaluate the overheads induced by our compiler on realistic examples.

CCS Concepts

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Domain specific languages**; **Retargetable compilers**; **Modules / packages**; *Source code generation.*

Keywords

Programmable dataplanes, P4, Modularity, Composition.

ACM Reference Format:

Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing Dataplane Programs with μP4. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3387514.3405872

1 Introduction

Over the past few years, the synergistic development of packet-processing hardware and software has changed how networks are built and run. Hardware models such as RMT [4] offer tremendous flexibility for customizing the dataplane without having to fabricate new chips, while languages such as P4 [3, 7] enable specifying rich packet-processing functions in terms of programmable parsers and reconfigurable match-action tables (MATs).

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405872

To support a variety of targets—e.g., software switches, ASICs, FPGAs, etc.—P4 allows programmable and fixed-function blocks to be arranged into different layouts as specified by an architecture declaration. For example, the Portable Switch Architecture (PSA) [10] models a switch with programmable parsers, programmable ingress and egress pipelines, as well as fixed-function schedulers and queues. However, while this design allows the language to flexibly accommodate a range of targets, it also creates a tight coupling between P4 programs and the underlying architectures, which makes it difficult to write, compose, and reuse common code fragments across different programs and architectures.

To illustrate these challenges, consider the two programs shown in Fig. 1, each written for a simple parser-control-deparser pipeline. The first program, `ether`, parses the Ethernet header, modifies the addresses using next hop (`nh`) which is supplied as an argument, and finally deparses the packet. The second program, `ipv4`, parses the IPv4 header, uses longest-prefix match to determine the next hop, decrements the `ttl` field, and deparses the packet. Note that neither of these is a complete program: `ether` is parameterized on the next hop and so it does not specify forwarding behavior, while `ipv4` does not generate a valid packet—at least, not one that is well-formed according to the standard networking stacks implemented on end hosts. To obtain a complete program, we must somehow combine the code in `ether` with the code in `ipv4`, so that `ether` can transfer execution control to `ipv4` appropriately—e.g., at ①—and also get back the value of `nh`. However, doing this correctly is non-trivial in P4 today.

In practice, large programs like `switch.p4` [6] are typically written in a monolithic style. To reuse the code in `switch.p4` to implement a different program, say a standalone Ethernet switch, one would need to somehow detangle the Ethernet-specific functionality from the rest of the program. Conversely, to add support for a new protocol such as SRv6 [21], one would need to make numerous changes to the program, including modifying header type declarations, extending parsers, adding tables, and updating the control flow. Without a detailed understanding of the entire program, doing this correctly is extremely difficult. Even worse, P4 programs rely on C preprocessor directives to enable and disable various inter-related features—an ad hoc and fragile approach that can easily result in errors [13]. Finally, to port the code from the V1Model to a new architecture, say PSA, one would need to replace the V1Model metadata and externs with PSA-specific ones and also restructure the program to conform to PSA's pipeline.

To address these challenges, this paper presents μP4—a new framework with a *logical* architecture that provides fine-grained abstractions for constructing and composing dataplane programs. Similar to Click [24], μP4 distills packet processing to its logical essence and abstracts away from hardware-level structures. This approach enables writing programs in a modular way, drawing on functions defined in simple programs and reusable libraries of code

```
parser P(packet_in pin, out hdr_t ph) {
  state start { pin.extract(ph.eth); }
}
control C(inout hdr_t ph, inout sm_t sm,
          in bit<16> nh ) { // Needs next-hop [nh]
  action drop() {}
  action forward(bit<48> dst_mac, bit<48> src_mac,
                 bit<8> port) {
    ph.eth.dstMac = dst_mac;
    ph.eth.srcMac = src_mac;
    sm.out_port = port;
  }
  table forward_tbl {
    key = { nh : exact; }
    actions = { forward; drop; }
  }
  apply { ① forward_tbl.apply(); }
}
control D(packet_out po, in hdr_t ph) {
  apply { po.emit(ph.eth); }
}
```

(a) `ether.p4`: Parses and processes L2 Ethernet headers using next-hop `nh`.

```
// Expects a (partial) packet starting at IPv4 header
// Sets out param [nh] based on dstAddr
parser P(packet_in pin, out hdr_t ph) {
  state start { pin.extract(ph.ipv4); }
}
control C(inout hdr_t ph, out bit<16> nh,
          inout sm_t sm) {    // sets [nh]
  action process(bit<16> next_hop) {
    ph.ipv4.ttl = ph.ipv4.ttl - 1;
    nh = next_hop;   // set out param [nh]
  }
  table ipv4_lpm_tbl {
    key = { ph.ipv4.dstAddr : lpm; }
    actions = { process; }
  }
  apply { ipv4_lpm_tbl.apply(); }
}
control D(packet_out po, in hdr_t ph) {
  apply { po.emit(ph.ipv4); }
}
```

(b) `ipv4.p4`: Parses and processes L3 IPv4 headers to identify next-hop.

Figure 1: Example P4 code snippets to illustrate the need for composing dataplane programs.

to specify rich packet-processing functionality—e.g., it enables the kind of composition needed above for `ether` and `ipv4` to build a modular router (§4). The μP4 compiler maps composite programs onto standard P4 architectures, composing parsers and MATs from multiple programs in complex and user-defined ways. To do this, it generates the code needed to emulate the logical packet-processing behavior using the available elements in the underlying pipeline, making use of static analysis and optimizations to partition the functionality while conserving hardware-level resources such as storage for headers and metadata.

Prior work on languages such as Pyretic [26] offer composable abstractions for OpenFlow switches, while systems such as HyPer4 [18], HyperV [37], and P4Visor [38] focus on merging target-specific P4 programs to execute on a single device. To construct practical dataplanes, we need to support more powerful forms of composition while retaining the ability to express complex packet processing–e.g., cloning and replication. μP4 enables these using logical buffers that interact along well-defined interfaces.

Overall, this paper makes the following contributions:

- We identify challenges in making dataplane programming modular, composable and portable (§2).
- We design a new framework, μP4, that enables fine-grained composition of dataplane programs in a portable manner, and we develop a case study using μP4 to build a modular router (§3-4).
- We develop techniques for compiling a μP4 program to multiple P4 targets, including merging program pieces and scheduling them on a packet-processing pipeline (§5).
- We prototype a μP4 compiler which targets two different architectures, including Barefoot's Tofino, showing that it is feasible to build and run complex dataplanes with μP4 within practical hardware resource constraints (§6-7).

While much work remains to fully achieve the vision of modular dataplane programming—e.g., developing libraries and optimizations, etc.—we believe μP4 presents a promising first step (§8).

Ethical concerns. This work does not raise any ethical issues.

2 Goals, Challenges and Insights

Goals. Our overall goal is to enable dataplane programming in a modular, composable, and portable manner, as we define in the next few paragraphs.

Modular: It should be possible to write individual packet-processing functions in an independent manner agnostic of other dataplane functions. For example, one should be able to define Ethernet and IPv4 packet-processing functionality as separate modules.

Composable: It should be easy to flexibly compose individual functions to construct larger dataplane programs. For example, in Fig. 1, imagine combining `ether` with `ipv4`, or any other routing scheme (e.g., IPv6, MPLS etc.) with a compatible interface and semantics, to obtain a modular router.

Portable: It should be easy to port programs to other architectures, say PSA, V1Model, or NetFPGA's SimpleSUMESwitch [19], without having to modify the source code. Following the "write-once, compile-anywhere" philosophy, programs should be loosely coupled to architectures and be based on general constructs that a compiler maps to architecture-specific constructs.

Challenges. We identify three main challenges in achieving these goals with the current P4 programming model.

C1. P4 programs are monolithic. P4 programs tend to be written in a monolithic style, with a "flat" top-level structure and a single collection of parsed headers and metadata that are threaded through the entire program. For example, in Fig. 2, PSA's ingress parser initializes headers and metadata that are subsequently processed by the ingress control. To reuse the code for ingress control block in a different context, the new parser would have to use the same headers and metadata as the original parser. While it is possible to restrict parameters and variables using lexical scope, the same is not true of types, parsers, and controls. In practice, programmers often use headers and metadata in unrestricted ways [6]—i.e., their programs lack the forms of abstraction and encapsulation needed

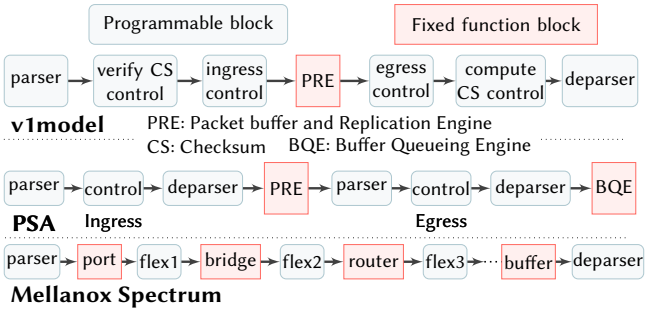


Figure 2: Architectures vary in their arrangement of dataplane pipelines.

to write truly modular programs. Even minor modifications to one part of a program can induce changes in the rest of the program.

C2. Heterogeneous programming model. Different programmable blocks in P4 are expressed in different sub-languages with incompatible abstract machines—e.g., while parsers are written in a sub-language based on finite state machines (FSMs), control blocks are written in a more standard imperative sub-language [7]. Due to this heterogeneity, P4 lacks a uniform interface for packet-processing modules; this prevents modules from being composed together to build larger programs. For instance, returning to Fig. 1, we might like to execute the parser, control, and deparser blocks of `ipv4` module at ① just before applying `forward_tbl` in `ether`'s control block so that `nh` is defined. However, this would require somehow executing part of `ether`'s control block after `ipv4`'s deparser, which would not be possible if the architecture exposed only a single deparser or control block.

C3. P4 programs are architecture-specific. A P4 program is dependent on the pipeline model of the architecture for which it is written. As architectures differ vastly in the set of programmable blocks, their arrangement in the pipeline, and the set of available externs and metadata (see Fig. 2), it is difficult to port a program to another architecture without a significant rewrite. For example, consider porting a program from PSA to V1Model architecture. Mapping the functionality implemented for PSA's ingress deparser block to any programmable block in V1Model's pipeline requires rewriting the program based on a semantic understanding of the blocks. In addition, P4 programs usually rely on architecture-specific metadata and associated primitives such as packet `recirculate`, `clone` etc. [8, 10], which further tightens the coupling with the architecture and undermines portability.

To summarize, current P4 programs are (i) monolithic as headers and metadata are generally shared across the entire program, (ii) not composable because of heterogeneous models for different processing blocks, and (iii) not portable because of their tight coupling to target architectures.

Insights. Based on these observations, we identify the following two insights that enable us to achieve the aforementioned goals.

I1. Homogenize abstract machines for flexibly composing and mapping modules to targets. Although primitives such as parsers and MATs are expressed using different sub-languages of P4, we find that fundamentally these primitives can be implemented by a common abstract machine based on MATs [17]. Homogenizing the abstract machines for all processing enables (i) powerful forms of composition by allowing unconstrained transfer of execution control between modules and (ii) flexibly mapping processing logic on to programmable blocks of a target.

I2. A general abstraction of dataplane architectures for target-agnostic and modular programming. In addition to common domain-specific primitives like MATs, the current P4 programming model relies on target-specific operations using fixed-function blocks; this results in P4 programs being architecture-specific (C1). By decoupling P4 programs from such target-specific constructs and using a general dataplane abstraction, we can express packet-processing in a way that is portable across architectures while supporting target-specific functions—imagine a compiler which links an architecture-agnostic program with architecture-specific libraries to build the dataplane. Such an approach also enables writing and composing independent modules while encapsulating implementation details.

Of course, introducing yet another architecture to unify existing architectures would add significant complexity to the P4 ecosystem. Instead, we identify a *logical* architecture that acts as high-level description against which programs are written. This architecture is logical in the sense that target devices do not explicitly implement it. Rather, it is designed to capture the essence of packet processing including: (i) simple linear pipelines that are expressive enough to encode a wide range of packet-processing functions, (ii) common interfaces for composing modules, and (iii) generic constructs for creating non-linear pipelines based on packet cloning and replication. We propose a compiler that maps programs written against this logical architecture to the architectures supported by P4 targets.

Summary. With μP4, users write programs in a variant of the P4 language that has been extended with new constructs to support modularity and composition. Each μP4 module processes a complete or partial packet along with associated metadata and generates one or more packets along with possibly modified metadata. μP4 modules communicate via a well-defined uniform interface—a *logical buffer*. Further, to express special processing such as packet replication, μP4 provides generic logical externs. Hence, μP4 abstracts away the details of target pipelines. The compiler (μP4C) maps target-agnostic μP4 programs with logical externs to target-specific realizations using a sequence of transformations. First, μP4C translates a program to a μP4-specific Intermediate Representation (IR). It then homogenizes the abstract machine for each block by transforming all programmable blocks into match-action units. This enables natural transfer of execution control across modules as well as code reuse and composition. Finally, μP4C partitions the IR into a configuration specific to the target architecture and schedules processing onto the available packet-processing blocks.

3 The μP4 Framework

"An abstraction is one thing that represents several real things equally well."

—Edsger W. Dijkstra

We now describe the key components of the μP4 framework. A μP4 dataplane is based on an abstract programming model in which modules correspond to fine-grained functions that hide implementation detail and communicate along clearly-defined interfaces.

μP4 dataplane model. Each packet-processing module is a self-contained execution unit and is structured according to the abstract

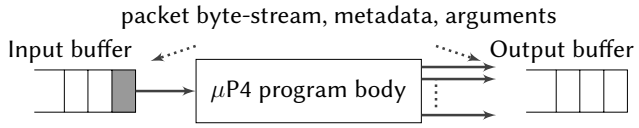


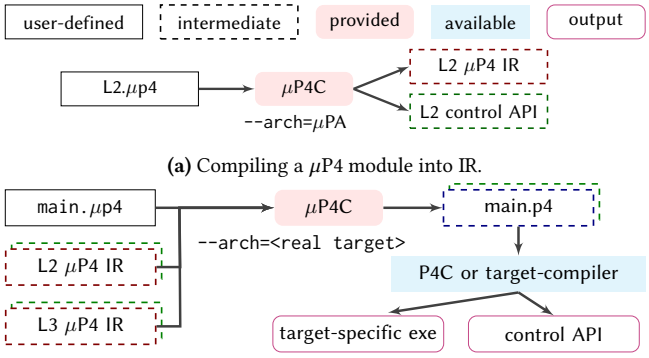
Figure 3: μP4's abstract dataplane model.

model shown in Fig. 3. A μP4 program takes as input a packet byte-stream, metadata and arguments for user-defined parameters and it generates byte-streams, metadata, and return values corresponding to one or more packets as output. Such a model encapsulates several important implementation details—e.g., header types, local state, user-defined metadata and MATs—of a μP4 program and hides them from other μP4 programs. This notion of encapsulation is key to enabling modular programming with μP4. Operationally, the model fetches an element from a logical input buffer, executes the μP4 program, and writes one or more elements to the logical output buffer, which typically acts as input for another μP4 program.[1] Note that these buffers exist only in the abstraction and do not represent buffers in actual targets. Encapsulation of packet-processing functions in modules that communicate via logical buffers form the basis of μP4's abstract programming model.

μP4 Architecture (μPA). Next, we expose μP4's dataplane model through a logical architecture (μPA) which provides concrete interfaces to write μP4 programs (§4). For this, our approach is inspired by the Click modular router [24]. In μPA, the packet-processing pipeline is specified as a composition of *logical μP4 programs*. Hence, each pipeline has one or more μP4 program bodies and exposes an interface to specify run-time parameters in a generic way. To abstract away from target-specific constructs, μPA provides logical externs and metadata—they facilitate the use of features such as packet replication which are not part of the core P4 language but are supported by common architectures.

μP4 Compiler (μP4C). μP4 extends P4 by allowing users to write libraries of independent packet-processing modules that are executed via a built-in `apply` method. Fig. 4 illustrates building a dataplane with μP4 and μP4C in two stages: (i) compiling individual target-agnostic modules and (ii) composing modules and linking them with a target-specific backend to build an executable dataplane. Users write individual modules for specific processing—e.g., `L2.μp4` in Fig. 4 performs only Ethernet processing similar to that in Fig. 1. Fig. 4a shows the first stage where μP4C translates a module into μPA-specific IR. Next, users link different modules together to build a dataplane—e.g., in Fig. 4b, `main.μp4` composes Ethernet (L2) and IPv4 (L3) modules to build a modular router. In during this, μP4C transforms the parser and deparser blocks of each module into MATs. This transformation unifies the abstract machines used for parsing and packet-processing, and allows seamless transfer of execution within and across modules, thus enabling flexible composition. Further, based on the specified target device, the compiler also generates the dataplane configuration specific to the device through a sequence of intermediate steps (§5).

[1]Packets marked with drop are not inserted into the output buffer.



(a) Compiling a μP4 module into IR.

(b) Compiling a composed μP4 program for a target.

Figure 4: Steps in compiling a μP4 program

4 μP4 Architecture (μPA)

This section presents μP4's logical architecture (μPA) in terms of: (i) *μP4 pipelines* and interfaces for writing μP4 programs (§4.1), and (ii) *logical externs* for expressing special packet-processing operations in a portable manner (§4.2).

4.1 μP4 Pipelines and Interfaces

A μP4 dataplane can be built out of two kinds of pipelines: *linear* and *orchestration* (Fig. 5). A linear pipeline models processing an input packet using a μP4 program in three distinct stages: parser, control, and deparser. The pipeline may generate multiple outputs for each input packet, but each packet is processed using the same logic. In contrast, an orchestration pipeline allows different copies of a packet to be processed in different ways—e.g., using conditionals on metadata and invoking different programs. A combination of these two kinds of pipelines enables expressing a wide range of packet-processing behaviors.

Interfaces. Recall the earlier example of composing L2 and L3 modules (Fig. 1)—if we composed L2 and L3 sequentially, L3 would have to parse the entire packet output by the L2 module. This introduces a tight coupling between them as the L3 parser has to parse the L2 header also in order to reach the L3 header. Ideally, we want the L3 module to process only the part of the packet starting at the L3 header. To enable this, μPA introduces three interfaces that allow one module to invoke another: `Unicast`, `Multicast` and `Orchestration`.

The `Unicast` and `Multicast` interfaces are implemented by linear pipelines, while `Orchestration` is implemented by orchestration pipelines. Essentially, these interfaces refine the notion of a logical buffer (Fig. 5)—e.g., a linear pipeline that writes multiple packets would use the `Multicast` interface to populate its output buffer. The following snippet shows the signatures for these interfaces while §A provides detailed declarations. Fig. 6 shows declarations of types, such as `pkt`, used here and Fig. 8 illustrates the usage of a `Unicast` interface.

```
Unicast<I,O,IO> (pkt p, im_t im, in I i_param,
                 out O o_param, inout IO io_param);
Multicast<I,O> (pkt p, im_t im, in I i_param,
                out_buf<O> ob);
Orchestration<I,O> (in_buf<I> ib, out_buf<O> ob);
```

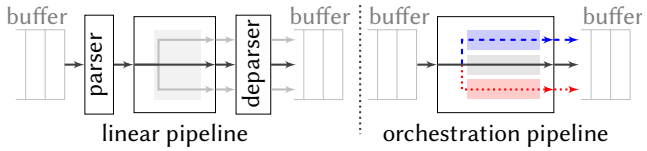


Figure 5: Linear and orchestration μP4 pipelines.

To implement the model in Fig. 3, μP4 implicitly performs a few operations while invoking μP4 programs—e.g., note that unlike `Orchestration`, `Unicast` is not parameterized on any buffer. So, the model fetches an element from an implicit logical input buffer for the `Unicast` interface, while for the `Orchestration` interface, it fetches the elements from a program-specified input buffer.

4.2 Logical Externs

μP4's logical externs model packet-processing on headers and metadata in a target-agnostic manner. The μP4C compiler maps these generic constructs to target-specific constructs, as shown in Fig. 6.

Packet extern. A packet is represented as a byte-array using the `pkt` extern. Instances of `pkt` can be created using the `copy_from` method and modified using the `extractor` and `emitter` externs.

Intrinsic metadata and constraints. Target architectures usually expose certain per-packet intrinsic metadata with some constraints on their usage—e.g., PSA's `output_port` cannot be updated in the egress pipeline [10]. To use such metadata and capture their constraints, μPA provides (i) an extern, `im_t` and (ii) an enumerator, `meta_t` (see Fig. 6). The extern `im_t` provides generic methods to access metadata, while `meta_t` maps μPA's metadata to target-specific metadata as defined in μP4C's target-specific backend (§5). Each value in the enumerator maps to an immutable intrinsic metadata field for the target—e.g., `ingress_timestamp`. Note that `meta_t` must be defined for each target in the μP4C backend. To access values set by the target, `im_t` provides a method `get_value`.

Input and output buffers. Corresponding to the logical buffers in the model (Fig. 3), μPA provides `in_buf` and `out_buf` externs. These are used to invoke other modules from within the control block of a μP4 pipeline. As the model restricts fetching multiple packets simultaneously from an input buffer, `in_buf`'s dequeue method is not exposed to the user. However, users can use the buffers to pass arguments to modules. `out_buf` exposes `enqueue` and `merge` methods to store packets processed by a callee. To move elements to an `in_buf` instance, which is passed as the argument, users can use `to_in_buf`. Unlike `in_buf` and `out_buf`, the `mc_buf` extern allows storing replicated headers for multicast.

Multicast extern. For programs that need packet replication, μPA provides the `mc_engine` extern. It can be instantiated within a control block while implementing the `Multicast` interface. To store copies of parsed headers, unparsed payloads, and metadata, `Multicast` provides a buffer of type `mc_buf` as a parameter. Users can create copies of a packet by invoking `mc_engine`'s `apply` method. Logically, this can be thought of as spawning multiple pipelines, each processing a replicated packet. Within each pipeline, all statements reachable from the method call are executed (see §B).

***Example**: Modular router.* We demonstrate how we can compose packet-processing modules to build a modular router using μP4's interfaces and externs. Fig. 8a shows `l3.`μ`p4`, which implements the `Unicast` interface and processes IPv4 and IPv6 headers. Both `ipv4` and `ipv6` ① set a user-defined parameter, `nh` (next-hop), based on an input packet `p` and intrinsic metadata `im`. Similarly, `L3` exposes a parameter, `type` ③. Such user-defined parameters allow flexible passing of data across modules for composition. In Fig. 8b, `ModularRouter` parses the Ethernet header and invokes an instance of L3 via `l3_i.apply()` ④. It passes a partial packet `p`, without the Ethernet header, to `L3`. `L3`, in turn, uses the `type` argument to invoke an instance of the appropriate protocol—IPv4 or IPv6—which parses the L3 header, sets the value of `nh`, deparses, and returns the execution control ②. `ModularRouter` uses the value of `nh` to continue L2 processing by applying `forward_tbl` table ⑤.

```
extern pkt { /* packet representation */
  byte[] packet;
  unsigned length;
  void copy_from(pkt pa);
}
extern emitter { /* packet assembler */
  void emit<H>(pkt p, in H hdr);
}
enum meta_t { /* enum mapping target metadata */
  IN_TIMESTAMP, OUT_TIMESTAMP,
  IN_PORT, PKT_LEN, ...
}
extern mc_engine { /* multicast extern */
  mc_engine();
  void set_mc_group(GroupId_t gid);
  apply(im_t, out PktInstId_t);
  set_buf(out_buf<O>);
  apply(pkt, im_t, out O);
}
extern extractor { /* header extraction extern */
  void extract<H>(pkt p, out H hdr);
  void extract<H>(pkt p, out H hdr, in bit<32> size);
  H lookahead<H>();
}
extern im_t { /* intrinsic metadata */
  void set_out_port(in bit<8>);
  bit<8> get_out_port();
  bit<32> get_value(in meta_t ft);
  void copy_from(im_t im);
}
/* used only by the architecture */
extern in_buf<I> {
  dequeue(pkt, im_t, out I);
}
extern out_buf<O> {
  enqueue(pkt p, im_t im, in O out_args);
  void to_in_buf(in_buf<O>);
  void merge(out_buf<O>);
}
extern mc_buf<H, O> {
  enqueue(pkt, in H, im_t, in O);
}
extern void recirculate<D>(in D data);
```

Figure 6: Type and extern declarations in μPA.

5 μP4 Compiler (μP4C)

Now, we present the design of μP4C—a compiler that transforms μP4 programs to target-specific P4 code.

[1]For brevity, we elide unused parameters in parsers, controls, and declarations.

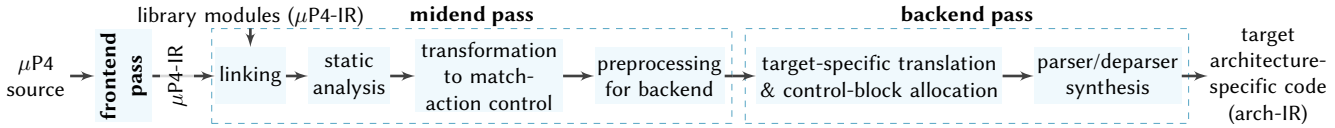


Figure 7: Overview of transformations performed in different passes in μP4C: frontend, midend, and backend.

```
ipv4(pkt p, im_t im, out bit<16> nh);
ipv6(pkt p, im_t im, out bit<16> nh);    ①
// implement unicast interface for L3 program
program L3 : implements Unicast<> {
parser P(extractor ex, pkt p, out empty_ht h,
         inout data_t d) {
  state start { transition accept; }  }
control C(pkt p, inout empty_ht h, im_t im,
          out bit<16> nh, inout bit<16> type) {
  ipv4() ipv4_i; // instantiation
  ipv6() ipv6_i;
  apply { switch (type) { // out arg: nh    ②
      0x0800: ipv4_i.apply(p, im, nh);
      0x86DD: ipv6_i.apply(p, im, nh); } } }
control D(emitter em, pkt p, in empty_ht h) {
    apply { } }
}
```

(a) L3 μP4 program implements IPv4 and IPv6 processing.

```
L3(pkt p, im_t im, out bit<16> nh,
    inout bit<16> type);    ③
program ModularRouter : implements Unicast<> {
parser P(extractor ex, pkt p, out hdr_t h) {
  state start {
    ex.extract(p, h.eth);
    transition accept; }
}
control C(pkt p, inout hdr_t h, im_t im) {
  // p is partial pkt without L2 header
  bit<16> nh;
  L3() l3_i;
  action drop () {}
  action forward(bit<48> dmac, bit<48> smac,
                 bit<8> port) {
    h.eth.dstMac = dmac;
    h.eth.srcMac = smac;
    im.set_out_port(port); // setting metadata
  }
  table forward_tbl { // L2 forwarding, needs nh
    key = { nh : exact; }
    actions = { forward; drop; } }
  apply { // invoke L3 to get nh
    l3_i.apply(p, im, nh, h.eth.etherType);    ④
    forward_tbl.apply(); // use nh in forward_tbl    ⑤
} }
control D(emitter em, pkt p, in hdr_t h) {
  apply { em.emit(p, h.eth); } }
}
ModularRouter(P, C, D) main;
```

(b) ModularRouter implements L2 processing and invokes L3 μP4 program.

Figure 8: Composing L2 and L3 processing to build a modular router.

5.1 Design Overview

μP4C builds on the P4 reference compiler, `p4c` [9], and has a modular design consisting of three passes: *frontend*, *midend* and *backend*. Fig. 7 shows an overview of these passes.

Frontend. The frontend transforms a μP4 programs into an intermediate representation (μP4-IR). It performs basic checks at the source level and serializes the μP4-IR to JSON.

Midend. The midend pass is target-agnostic and focuses on composition and preprocessing for the backend pass. It performs four main transformations: (i) linking μP4-IR for modules, (ii) static analysis (§5.2), (iii) homogenizing parser and control blocks to enable composition (§5.3), and (iv) preprocessing any packet copying constructs—e.g., `copy_from`—for the backend pass (§5.4). After this pass, the composed μP4-IR contains only control blocks with μPA-specific constructs.

The midend pass begins by linking the μP4-IRs of all the μP4 programs that need to be composed. It then performs a static analysis of the μP4 programs to compute its "operational-region"—i.e., the region of a packet's byte-stream that the program needs to access—and synthesizes a stack of one-byte headers, called a *byte-stack*, large enough to store the operational-region. Next, considering this byte-stack as a packet, it transforms all the parsers and deparsers into MAT control blocks, thus homogenizing the processing blocks. This step simplifies the μP4-IRs of μP4 programs and composes them. Naturally, the μP4-IR of the "main" μP4 program—i.e., the one composing together other μP4 programs—is also transformed to a MAT control block that invokes others. Optionally, in the case of packet replication, μP4C's midend preprocesses `copy_from` of `pkt`. It extracts packet-processing code for every `pkt` instance and prepares a processing schedule for the backend.

Backend. This pass is specific to a target architecture and has two goals: (i) to allocate control blocks on to the target's dataplane pipeline while respecting any constraints on metadata (§5.5) and (ii) to synthesize required target-specific parser and deparser blocks needed to parse and deparse packets into μP4-specific byte-stack.

5.2 Static Analysis

μP4C performs static analysis to compute the operational-region, which is quantified by: (i) *extract-length*: the maximum number of bytes that need to be extracted from a packet to execute the composed μP4 program, (ii) size of the byte-stack needed to store new header instances that may be added during processing, and (iii) *min-packet-size*, the minimum size of packets that may be accepted. μP4C computes these values recursively for each μP4 program involved in composition as the programming model allows nested invocations of μP4 programs.

To compute these values for a μP4 program, ψ, we first analyze the parse graph of its parser and compute the extract-length for the parser, $El_p(\psi)$, as the maximum number of bytes extracted by the

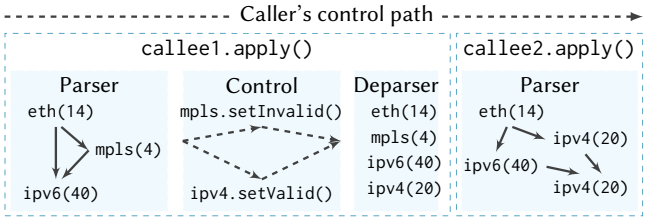


Figure 9: Static analysis: multiple callees in a control path.

parser to reach the *accept* state. ψ's extract-length also depends on its callees. So, for a control block, c, we define the extract-length, $El_c(\psi)$, as the maximum number of bytes extracted during execution of any of its control paths. In a control path, multiple callees may be invoked. We define the extract-length, $lc_\psi(x)$, of a control path x as the maximum number of bytes required to process all the callees in x. The μP4 program's extract-length, $El(\psi)$, is the sum of extract-lengths of its parser ($El_p(\psi)$) and control blocks ($El_c(\psi)$). Note that the size of the byte-stack needed for a μP4 program may differ from its extract-length as a packet's size may change during processing.

To estimate the maximum decrease in packet size in a control path, we consider all the header instances which are set to valid or invalid in the control path. The rationale for this is that if a packet already has a header instance, setting it to invalid will decrease the packet size, but setting it to valid will not. Similarly, for maximum increase, we consider an input packet that does not contain any header instance that is set to valid or invalid in the path. Then, we perform static analysis of the control block by evaluating all the `setValid` and `setInvalid` invocations. We denote increase and decrease in packet size on a control path x with $i_\psi(x)$ and $d_\psi(x)$, respectively. Similarly, $\Delta(\psi)$ and $\delta(\psi)$ denote increase and decrease for the entire μP4 program. Eq. (1) expresses $i_\psi(x)$ as the sum of (i) sizes of all the header instances on which `setValid` is done and (ii) maximum increase in packet size by every callee μP4 program in the path. We compute $d_\psi(x)$ in a similar way as show in Eq. (2).

$$i_\psi(x) = \sum_{\texttt{H.setValid}\in x} sizeof(H) + \sum_{\texttt{callee.apply()}\in x} \Delta(\texttt{callee}) \quad (1)$$

$$d_\psi(x) = \sum_{\texttt{H.setInValid}\in x} sizeof(H) + \sum_{\texttt{callee.apply()}\in x} \delta(\texttt{callee}) \quad (2)$$

Header instances that are not emitted by the deparser but are extracted by the parser also decrease a packet's size. So, we add the size of such header instances to $d_\psi(x)$ for every path x. For ψ, we compute the maximum increase $\Delta(\psi) = \max_x \{i_\psi(x)\}$ and decrease $\delta(\psi) = \max_x \{d_\psi(x)\}$.

Now, we estimate the extract-length, $lc_\psi(x)$, for a path x in ψ's control block as a function of the extract-length, $El(\texttt{callee})$, and maximum decrease in packet size, $\delta(\texttt{callee})$, of callees. Assume that a control path x invokes N callees, as shown in Fig. 9. The i^{th} callee may decrease a packet's size—e.g., a control path of `callee1` removes the `mpls` header, decreasing the size by 4 bytes. `callee2` may extract `eth`, `ipv6` and `ipv4` headers from the packet. So, to process (i) removal of 4-byte `mpls` header by `callee1` and (ii) extraction of maximum bytes (74-byte `eth-ipv6-ipv4`) in `callee2`'s parser, 4+74=78 bytes are required. We take into account the extract-length of every callee's parser along with the maximum decrease in packet size by the callee's predecessors in the control path x to compute $lc_\psi(x)$, as shown in Eq. (3).

$$lc_\psi(x) = \max_{cp} \left\{ \left(\sum_{i=0}^{i<cp} \delta(i) \right) + El(cp) \right\}, \qquad cp \in [0, N) \quad (3)$$

$$\mathcal{B}s_\psi = El(\psi) + \Delta(\psi) \quad (4)$$

Finally, we compute the byte-stack size for ψ, $\mathcal{B}s_\psi$, as the sum of its extract-length and maximum increase in packet size as shown in Eq. (4). For the example in Fig. 9, the byte-stack size for the caller is 98 ($El($`caller`$) = 78$, and $\Delta($`caller`$) = 20$ for increase in `callee1`) bytes. We perform a similar analysis for *min-packet-size* but elide the details for brevity.

Scalability. Although μP4C performs static analysis of the parse graph and control flow graph by exploring the different paths, it does not face the usual scalability issues with symbolic execution of dataplane programs [34]. This is because μP4C's static analysis of a parse graph to compute the operational region can be reduced to finding the longest path in a directed acyclic graph, which can be done in linear time. For the control flow graph, general symbolic execution does not scale as each table entry introduces a potential branch. This is not the case with μP4C as the static analysis does not depend on the table entries. Instead, μP4C needs to consider only the branches in the structure of program—e.g., due to conditionals and number of actions per MAT. Therefore, μP4C's static analysis scales well to large programs in practice.

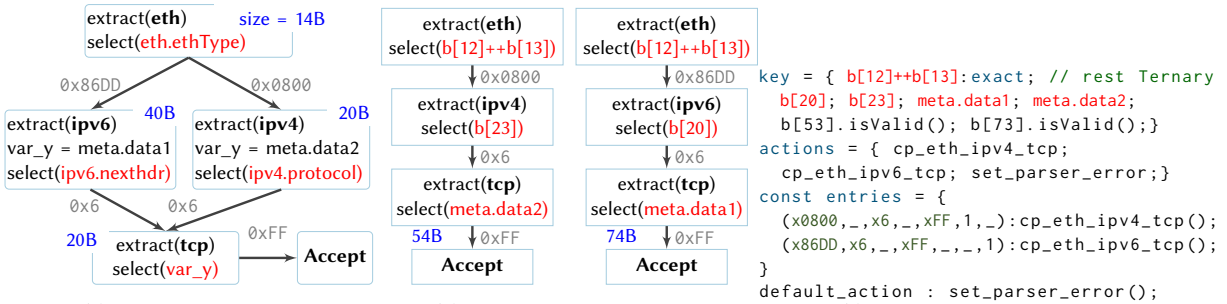
5.3 Homogenizing Programmable Blocks

To compose μP4 programs, it is crucial to homogenize the abstract machines of the processing blocks. This enables transfer of execution control between modules—such as that needed for a modular router—which existing systems do not support [18, 38].

Parser. A close look at the design of programmable parsers reveals that they essentially perform repeated match-action operations [17]. So, with the operational-region in byte-stack, we can perform these operations simultaneously in hardware. We use the parser in Fig. 10a as a running example to explain how we synthesize MATs for a parser.

The parser's FSM has four states to extract Ethernet, IPv4, IPv6 and TCP headers of size 14, 20, 40 and 20 bytes, respectively. μP4C performs static analysis of the parser to identify two possible paths from the start state to the accept state. Then, it replaces the header fields in each path—e.g., `eth.ethType` and `ipv6.nexthdr`—with their evaluated offsets in the byte-stack (b)—e.g., `b[12]++b[13]` and `b[20]`—as shown in Fig. 10b. It also performs *Forward Substitution* [28] on every path to eliminate any anti-dependency between two states—e.g., `var_y` is replaced with `meta.data1` in one and `meta.data2` in the other.

μP4C creates a match-action entry for each path as follows. Using the byte-stack offsets and variables used in `select` expressions of states in each path (Fig. 10b), μP4C synthesizes a match-key by merging them (Fig. 10c). To ensure that a packet is long enough to be parsed, the key also encodes a validity test for the last byte



(a) An example P4 parser **(b)** Static analysis of the parser **(c)** Parser converted to match-action control

Figure 10: Example illustrating transformation of a parser to a MAT control block by μP4C.

extracted on the path. Then, it synthesizes an action for each path—e.g., `cp_eth_ipv6_tcp`—which has assignment statements to instantiate the relevant header fields. μP4C pairs a key with the appropriate action to create an entry for the path—e.g., in the first entry in Fig. 10c, `x0800`, `x6` & `xFF` are matched to the first, third and fifth keys to parse a byte-stack as Ethernet, IPv4 and TCP headers while the second and fourth keys are ignored.

Deparser. Using the emit order of header instances and the number of bytes extracted by the parser, μP4C synthesizes MAT entries to copy back user-defined headers at appropriate offsets in the byte-stack. A μP4 program may set a header instance (in)valid, thereby modifying a packet's size. In case of an increase, the byte-stack is large enough to hold new headers, while in case of a decrease, data is shifted—e.g., for `callee1` in Fig. 9, if `mpls` header is removed, the following 60 bytes are moved up by an offset of 4 in the stack.

After this transformation, the entire program is ready to be allocated on the programmable match-action units of a target. This is helpful in flexibly composing μP4 programs as the composed program can again be mapped to the same unit.

5.4 Preprocessing for μP4C backend

μPA's logical externs allow expressing non-linear processing such as packet replication. We outline our compilation approach that makes these logical externs amenable to be mapped to targets. If a program uses `copy_from` method to duplicate a packet, μP4C prepares a *Packet-Processing Schedule (PPS)* graph for the μP4 program. A node in a PPS graph denotes the μP4 program's sub-program, called `thread`, that processes a single packet instance, while an edge represents dependency among threads. To prepare a PPS by extracting sub-programs, we construct a Program Dependence Graph (PDG) [14] from the μP4 program's IR and perform *slicing* that is defined based on a variant of *program slice* [36]. We define our slicing criteria based on the semantics of logical externs, `pkt`, `in_buf` and `out_buf`, along with `apply` method exposed by μPA's interfaces. See §C for more details.

5.5 Mapping to the Target Pipeline

μP4C's backend generates target-specific P4 source based on semantic understanding of the target's pipeline model. It requires a mapping from μPA metadata and externs to target-specific ones and any constraints on target's metadata. The backend performs a straightforward translation of μPA's extern method calls with the corresponding externs in the target architecture. It allocates the control blocks in μP4-IR to the programmable blocks exposed by a target while respecting any constraints on placement.

μP4C performs a *partitioning* transformation to allocate packet-processing blocks of a single thread on programmable control blocks of target pipeline. In case of packet replication, this transformation can be performed on every thread. Next, we explain the partitioning for single thread while using V1Model as a reference target architecture.

μP4C's backend for V1Model maintains a FSM with two states—ingress and egress. The FSM captures constraints on the usage of `egress_spec`, `egress_port` and queuing metadata in the ingress and egress blocks. Each state represents a set of assertions to be verified on visiting each program statement in the Control Flow Graph (CFG) of the thread—e.g., to prevent accessing dequeue timestamp of a packet in ingress pipeline, the graph traversal asserts that every visited statement is NOT a `im_t`'s `get_value` method call whose argument maps to V1Model's intrinsic metadata for `deq_timestamp`. If an assertion associated with a state fails, the program statement is marked and not visited. State transition occurs when graph traversal cannot continue due to absence of unmarked and unvisited nodes. At this point, μP4C creates two sub-graphs of the thread CFG—one having visited and the other having unvisited program statements. The FSM also transits to egress state. Similarly, in the egress state, the graph can enforce egress-specific constraints.

For handling data dependency—e.g., sharing live local variables—μP4C synthesizes *partition-metadata* that can be passed as user-metadata between ingress and egress control blocks. Nodes across sub-graphs may be connected by edges to represent control dependencies among sub-graphs. μP4C converts control dependencies into data dependencies by synthesizing appropriate metadata.

6 Implementation

Our prototype compiler, μP4C, implements a core subset of μP4 by extending the P4 reference compiler [9]. Specifically, the prototype supports all the μP4 constructs with the exception of multicast and orchestration interfaces. It includes backends to compile μP4 programs for two target architectures: (i) Barefoot's Tofino Native Architecture (TNA) [27] and (ii) V1Model [11]. In addition to the frontend extensions for μP4's syntax, μP4C's midend and backend comprise ~13,500 LoC. Using this prototype, we have implemented several features of a datacenter switch [6] in μP4 and built dataplanes by composing different subsets of the features. The

backends allow us to evaluate the portability across targets and resource overheads on a hardware target for μP4. We have released our μP4C implementation along with the example programs under an open-source license [32].

In the rest of this section, we discuss: (i) differences with P4 (§6.1), (ii) supporting new target architectures (§6.2), (iii) experience with building a backend for Tofino (§6.3), and (iv) limitations and avenues for further improvement (§6.4).

6.1 Comparison with P4

One of our goals is to keep the changes to $P4_{16}$ syntax and grammar minimal to ease the adoption of μP4. We briefly describe some of the main differences. First, μP4 allows defining custom package types which hide the implementation and expose an interface to reuse code. Second, μPA defines interfaces to encapsulate a set of programmable blocks so that the logical dataplane pipeline can be extended. Third, μP4 requires explicit data passing between modules in contrast to implicit globally shared metadata in P4. Finally, instead of target-specific constructs, programmers use logical μPA constructs. To represent packets and to extract and emit headers, μPA defines a different set of externs compared to those defined in the P4 core library. Apart from these, μP4 conforms to $P4_{16}$'s constructs such as parsers, controls, externs, etc.

6.2 Supporting New Target Architectures

The only target-specific component of μP4C is the backend (§5.5), which translates μP4-IR into a P4 program for the specified target. As the backend maps μPA's generic metadata to metadata specific to the target architecture, it requires architecture-specific mappings for metadata and methods corresponding to `meta_t`, `im_t` and `mc_engine`. The generated program needs to conform to the target's pipeline model and so, the backend has to take care of any constraints on target metadata and externs. Further, if the midend of the target's compiler is available, μP4C's backend can pass the IR to the target compiler's midend to generate the executable for the target. Thus, to support a new target, we need to provide μP4C's backend with a mapping from μPA's logical constructs to target-specific constructs.

6.3 μP4C's TNA Backend for Tofino

ASIC designers often make design choices that are driven towards optimizing processing rate, power, chip area and cost. Such choices manifest as constraints while allocating the finite resources available on the chip to support reconfigurability [23]. We need to understand these choices and constraints to build efficient backends. We briefly discuss our experience in developing a backend for Tofino.

Tofino is based on RMT [4] which consists of a pipeline of match-action units assembled in multiple stages. Packet header data and other metadata is carried along these stages by packing them in multiple fixed-size *containers*, which form the Packet Header Vector (PHV). Tofino supports containers of sizes 8, 16 and 32-bits [29]. Moreover, only a small subset of these containers is accessible from the action ALUs in each stage. These add to a complex set of constraints which must be satisfied while optimizing resource allocation. Indeed, these constraints might be infeasible in some cases—e.g., for a program with a long dependency chain of MATs.

While resource allocation is the task of the target's compiler, `bf-p4c` in this case [1, 9], μP4C's backend must generate TNA P4 programs for which `bf-p4c` can satisfy the resource constraints. So, we aim to generate code that is amenable to `bf-p4c`'s heuristics for constrained resource optimization. The generated code uses (i) a byte-stack and (ii) assignments to/from elements in the byte-stack, involving bit-slicing and concatenation to convert (de)parsers into MATs. μP4C must allocate the byte-stack on PHVs in such a way that assignments can be scheduled on action units without exceeding the maximum number of PHV containers accessible to each action ALU. To address this, μP4C backend needs to be aware of the size and number of PHV containers and associated constraints so that it can (i) optimize container utilization by aligning field sizes with container sizes, avoiding fragmentation of fields and making resource allocation tractable and (ii) break down complex assignment operations which need more PHV containers than available per action ALU into simpler operations. The latter allows the deparser to write data back from user-defined fields to the byte-stack while efficiently using PHVs per action ALU.

We leveraged these observations and insights by integrating another pass in μP4C's backend for TNA. This pass adjusts the size of elements in byte-stack and restructures operations to simplify resource allocation for `bf-p4c`. As a result, we were able to compile various programs for Tofino, which were earlier rejected by `bf-p4c`. This demonstrates the feasibility of using μP4C to support commodity hardware targets.

6.4 Limitations

While μP4 enables portable, modular and composable dataplane programming, we discuss the limitations of our prototype along with some insights to handle these limitations.

Stateful packet processing. Certain applications require stateful processing of packets, and architectures provide various constructs to persist such state—e.g., registers in PSA. Note that state in P4 is fundamentally similar to the notion of a *static* variable. While our prototype does not implement it, μP4 can be extended to support static variables which μP4C can map to architecture-specific constructs such as registers.

Traffic manager. Similar to P4, providing QoS guarantees by controlling the traffic manager or changing the scheduling algorithm is beyond the scope of μP4. It can still be used to emulate scheduling algorithms with programmable schedulers such as PIFO [31].

CPU-dataplane interface. The prototype lacks an interface for communication between the dataplane and the CPU. To support this, we can use target-specific constructs such as `digest` and `packet_in` for PSA, or extend μPA's logical externs.

Packet replication. While μP4 and our design or μP4C support packet replication using multicast and orchestration interfaces, our current prototype of μP4C does not fully implement it yet.

Device capabilities. Devices support MATs with a fixed set of match kinds such as exact, LPM, ternary, and range. μP4 requires a target to support all match kinds used in a program; μP4C does not transform across different match kinds as it requires translating entries at runtime—e.g., translating a range match entry for a target that supports only exact match requires enumerating the

range at runtime, while our focus is on compile-time abstractions. Similarly, μP4 cannot automatically synthesize implementations for new checksum computations, action selector, and action profiles.

Recursion. Our implementation does not support recursion—e.g., consider a use case with VLAN-IP-GRE-VLAN headers by composing VLAN, IP and GRE μP4 program modules. Trivially inlining the modules in such a call-chain would cause the compiler to fail. While possible, our current implementation does not check for potential cyclic dependencies among μP4 modules to reject such programs.

7 Evaluation

Our evaluation focuses on two key questions: (i) does μP4 enable dataplane programming in a modular, composable and portable manner (§7.1–§7.2), and (ii) can μP4 programs be run on commodity programmable switches (§7.3)? We answer both these questions positively by implementing a library of widely-used packet-processing functions as independent μP4 modules and composing them to build new dataplane programs. Table 1 shows a subset of these programs which we refer to throughout this section. Each row corresponds to a μP4 module, and each column corresponds to a composed program ($\mathcal{P}1$–$\mathcal{P}7$) generated by combining the modules of checked rows. The composed programs can perform a variety of functions including Ethernet switching, IPv4 and IPv6 routing, MPLS-based edge routing (encap/decap), Network Address Translation (NAT), IPv6 Network Prefix Translation (NPTv6), Firewall/ACL, and Segment Routing (SRv4 and SRv6). We also implemented the equivalent monolithic programs in P4 for comparison. We verify portability of μP4 programs by reusing the same modules and compiling the composed programs for two architectures: V1Model and TNA.

7.1 Composing Packet-Processing Layers

Packets are usually structured as layers of protocol headers with standard layouts, where each layer is specialized for a specific role. So, implementing modules for processing one or more layers is a natural way to build applications as these modules can be developed incrementally, composed together, and reused.

Layered modularity. By specializing the generic interfaces provided by μPA, users can write and compose μP4 modules–each processing one network layer. To illustrate, we built a modular router ($\mathcal{P}4$) by composing independent modules for processing L2 (Ethernet) and L3 (IPv4 and IPv6) headers in §4 (Fig. 8).

Incremental development. Suppose we need to add support for IPv6 Segment Routing (SRv6) to our modular router. For this, one can independently develop another module (`srv6`) to process an SRv6 header [20]. Then, we can incrementally extend L3 (Fig. 8a) to support SRv6 without touching any other module:

```
srv6() srv6_i;     // instantiation
apply { switch (type) {
0x0800 : ipv4_i.apply(p, im, nh);
0x86DD : { srv6_i.apply(p, im, nh); // copy next segment's
     address from SRH's list to IPv6's destAddr field
           ipv6_i.apply(p, im, nh);   }
} }
```

Programs	$\mathcal{P}1$	$\mathcal{P}2$	$\mathcal{P}3$	$\mathcal{P}4$	$\mathcal{P}5$	$\mathcal{P}6$	$\mathcal{P}7$
μP4 modules: ACL						✓	
Eth	✓	✓	✓	✓	✓	✓	✓
IPv4		✓	✓	✓	✓	✓	✓
IPv6	✓	✓	✓	✓		✓	✓
MPLS		✓					
NAT						✓	
NPTv6						✓	
SRv4							✓
SRv6			✓				

Table 1: Composing μP4 modules to build dataplane programs.

7.2 Composing Network Functions (NFs)

NFs such as firewall and NAT access headers across protocol layers. Operators often need to deploy such NFs on devices using various combinations as per some policy. To enable this, prior work on dataplane composition (P4Visor [38] and P4Bricks [33]) and virtualization (HyPer4 [18] and HyperV [37]) define operators based on certain use cases. μP4 naturally supports such kinds of composition using a combination of μP4-specific features such as user-defined package types and native P4 constructs such as conditionals.

Realizing composition operators. The following μP4 code implements *sequential* and *override* operators from CoVisor [22].

```
// Sequential: Firewall -> Routing
firewall.apply(p, im, result);
if (result.drop == false) {
  modular_router.apply(p, im, tc);  // tc, out args
  // Override: routing decision
  mpls_ler.apply(p, im, tc); // label based on tc
}
```

Here, a packet is first processed by `firewall`, which may decide to drop it after analyzing one or more headers; otherwise, the packet is routed. While routing, based on the packet's traffic class (`tc`), `mpls_ler` may override the routing decision made by `modular_router`.

`A-B Testing` is another composition operator, defined in P4Visor, that μP4 can implement as follows:

```
/* A-B Testing in core devices*/
// parser extracts 1 byte header with flag
extractor.extract(p, testHdr);
// Inside control block, the `p` w/o testHdr
if (h.testHdr.Flag == 1) test_prog.apply(p, im);
else prod_prog.apply(p, im);
// deparser puts back the test header
emitter.emit(p, testHdr);
```

7.3 μP4 on Commodity Hardware

Despite the hardware constraints mentioned in §6.3, μP4C can be used to compile μP4 programs for commodity programmable switches such as Barefoot's Tofino. To benchmark μP4's overheads, we measure the hardware resource utilization on Tofino for μP4 programs and compare with their monolithic versions. Overall, we find that μP4 programs require more resources, and we discuss sources of these overheads.

Resource allocation. We found interesting cases where monolithic P4 programs failed to meet the resource constraints for Tofino while μP4 programs met (and vice versa). For example, compiling

Program	% resource overhead w.r.t. monolithic			
	PHV Container Used			Bits allocated
	8b	16b	32b	
$\mathcal{P}1$	80.00	312.50	-85.00	32.34
$\mathcal{P}2$	0.00	315.79	-84.21	0.00
$\mathcal{P}3$	272.73	564.71	-85.71	54.58
$\mathcal{P}4$	9.09	331.25	-85.00	1.64
$\mathcal{P}5$	-20.00	226.67	-63.64	47.10
$\mathcal{P}6$	18.18	290.48	-80.00	48.52
$\mathcal{P}7$	NA: Monolithic failed to compile			

Table 2: Resource overhead of running μP4 programs relative to their monolithic versions in terms of PHV utilization on Tofino. $\frac{usage(\mu P4)-usage(monolithic)}{usage(monolithic)} \times 100\%$

μP4C-generated P4 code for $\mathcal{P}2$ using `bf-p4c` failed initially because an assignment operation in the generated code was trying to access more than the number of containers accessible to an action ALU (§6.3). However, such issues are not fundamental inhibitors as we can modify the generated programs to meet known hardware constraints. In this case, it involved breaking down the complex assignment into multiple simpler ones which are executed in a series of MATs. $\mathcal{P}2$ compiled for Tofino after this transformation. For the same problem, somewhat counter-intuitively, we found that increasing the size of MPLS header fields also solved the issue without the previous transformation. This is because the new field sizes aligned well with the container sizes on Tofino, reducing fragmentation of fields into containers; this decreased the number of containers needed for the assignment operation to within that available per action ALU. We also found cases where μP4 programs could compile but their monolithic P4 version did not—e.g., `bf-p4c` failed to allocate resources for the monolithic version of $\mathcal{P}7$.

Resource allocation is a complex problem that affects both modular and monolithic programs. Of course, in practice, a program that fits is more useful than one for which the compiler can not satisfy hardware constraints.

Resource utilization. To understand the cost of composition, Table 2 shows the hardware resource utilization on Tofino for μP4 programs relative to their monolithic version in terms of (i) the number of containers used for each container size, and (ii) the total number of PHV bits allocated. Our main observation is that in each case, the resources required to run μP4 programs were within Tofino's limits. In terms of container utilization, μP4 programs heavily utilize 16b containers—almost 3× of their monolithic counterparts. This is expected as our TNA backend updated the field sizes to align with 16b containers. While we cannot completely control the allocation of different size containers, we find that the total number of PHV bits allocated for μP4 is within 1.5× of monolithic—in some cases, the same as monolithic. Further, note that the usage of 32b containers with μP4 is negligible (1/6×) as compared to the monolithic ones. We believe that with further optimizations, we can reduce such overheads significantly by allocating the containers uniformly.

The number of hardware stages required for μP4 programs is also higher than those for monolithic programs, as shown in Table 3. This is because μP4 transforms (de)parsers into MATs to enable flexible composition (§5.1). But, in each case, we were able to successfully fit μP4 programs on Tofino. Note that as μP4 programs do not require specialized hardware for (de)parsers, it has potential gains which we could not benchmark.

		$\mathcal{P}1$	$\mathcal{P}2$	$\mathcal{P}3$	$\mathcal{P}4$	$\mathcal{P}5$	$\mathcal{P}6$	$\mathcal{P}7$
#stages	P4 monolithic	3	4	3	3	3	3	NA
	μP4 composed	5	9	8	5	5	8	7

Table 3: Number of stages utilized on Tofino.

8 Discussion and Future Work

We believe μP4 is an important first step toward making modular data plane programming a reality. This section discusses some ideas for making it practical (§8.1) and for future work (§8.2).

8.1 Overheads and Optimizations

The main source of resource overheads with μP4 is μP4C's transformation of (de)parsers into MATs as it affects the resource allocation heuristics of the target compiler (`bf-p4c`) in several ways. First, these MATs may introduce new dependencies between parsed headers, subsequent MATs and byte-stack in the composed CFG. Depending on the heuristics, the compiler may map the transformed (de)parser and associated MATs on to dedicated MAU stages in the hardware pipeline. Second, programs composed as a linear sequence may perform unnecessary deparsing and parsing of the same headers in MAU stages—e.g., imagine the deparser of one program being simply the "inverse" of the parser of the subsequent program. Finally, the heuristics may fail to satisfy certain constraints, such as those in §6.3, if the composed program does not meet certain implicit assumptions of the target.

We outline several possible optimizations to alleviate these overheads. First, instead of generating a single MAT for a (de)parser, μP4C can generate multiple MATs to split (de)parsing. While this may seem counter-intuitive, it enables the target compiler to perform fine-grained optimization and placement—e.g., by resolving dependencies and co-locating non-conflicting processing blocks in the same stage. Second, for sequential composition, μP4C can analyze the deparser and parser of consecutive programs for partial equivalence. This allows compressing or even eliminating unnecessary deparsing and parsing within a composed μP4 program. Finally, we can leverage the fact that dataplane programs usually implement standard networking protocols; using such domain-specific knowledge, μP4C can reconstruct a single global parser by merging and concatenating all the parsers [33, 38]. This global parser can be executed in the programmable parser unit on the hardware while any metadata in callee μP4 programs can still be initialized by synthesizing MATs. It also reduces the instances of complex assignment operations which are likely to run into hardware constraints explained earlier (§6.3). With this, we expect the number of hardware stages needed for μP4 programs to match those for monolithic programs. However, reconstructing a global parser may be difficult in certain cases—e.g., when a μP4 program invokes different μP4 programs based on information provided by the control plane at runtime. Note that performing similar processing with P4 today would require resubmitting the packet, resulting in reduced

performance. In contrast, a μP4 program may trade-off resources for better performance by avoiding the need for resubmitting.

In this work, our experience with hardware targets is limited to RMT-based Tofino. We have not investigated resource overheads with other RMT-based hardware targets or with different pipeline architectures such as dRMT [5]. Different targets may have a different set of hardware constraints, and the overheads with μP4 as compared to P4 may vary.

8.2 Future Directions

We discuss some directions to address the limitations of μP4 and also outline interesting research directions to explore further.

Abstractions for stateful processing. To expose stateful abstractions for the dataplane, μP4 can be extended to include static variables and, more broadly, the notion of storage classes and lifetime. This provides a general abstraction for architecture-specific stateful constructs such as registers, counters and meters.

Target-agnostic composition. μP4 does not enable easy reuse of P4 programs already tied to a target architecture. However, it would be feasible to translate target-specific P4 programs to μP4 modules, reuse and compose them with other μP4 modules and, finally, retarget the composed μP4 program for another architecture.

Equivalence and verification. μP4 opens up avenues for cross-architecture transformation of dataplane programs. This introduces new challenges in terms of verifying the correctness of transformations and proving the equivalence of the dataplanes generated for different targets.

Control and management interface. μP4 enables building dataplanes in a modular way. To fully leverage the benefits, control plane APIs should also be designed to allow multiple controllers to configure subsets of dataplane modules on a device in a coordinated way. Further, μP4's architecture can be extended to provide abstractions for managing the switch CPU-dataplane interface and programming the traffic manager.

Debugging. While we do not focus on debugging in this work, we believe it is an important direction that μP4 can facilitate—e.g., programs can be linked against μP4 debug modules by using the common interface. It would be interesting to explore different design choices for the debugger's interface, logging information in the dataplane, and switch CPU-dataplane interfaces.

9 Related Work

In recent years, there has been a significant effort towards making network programming composable as networks are complex to manage and configure without having a modular approach. These have ranged from composing policies and controllers to dataplanes [2, 12, 15, 18, 22, 25, 26, 35, 37, 38].

Recent work on composing control planes and network policies [2, 26] focuses on defining a set of composition operators that can be used to describe complex policies from smaller policies in a consistent manner. This approach suits control programs as each program specifies the complete behavior of a network or device. Inspired by these efforts, recent work [18, 37, 38] has tried to define composition operators for dataplane programs based on specific use cases. μP4 fundamentally differs from these as it enables flexible passing of control flow and data between dataplane programs. This allows programmers to reuse fine-grained packet-processing code and define custom forms of composition. Recent work also motivates the need for similar composition by positioning multitenancy of programmable network devices as a primary requirement [35]. Concurrent work on Lyra [16] shares similar goals as μP4, but takes a different approach by defining a one-big-pipeline abstraction that allows users to express their intent.

PSA [10] attempts to achieve a form of portability by defining a standard architecture that many targets can realize. μP4 is orthogonal to PSA as it focuses on defining an abstraction that programmers can use to write dataplane programs that is portable across several target architectures, including PSA. μP4 is closer in spirit to Domino [30], which provides abstractions for expressing stateful algorithms and a compiler that generates low-level microcode.

At a conceptual level, the design philosophy of μP4's architecture resembles the Click modular router[24]. Click allows composing packet processing modules, called *elements*, that can be connected together using *push* or *pull* ports to create a directed graph. The edges of the graph determine the flow of packets. μP4's approach slightly differs—μPA's interfaces allow control and data transfer along with a packet. Further, the Click framework is targeted for general purpose CPUs while μP4 targets P4-programmable devices.

10 Conclusion

Dataplane programs are evolving to be complex and diverse with novel use cases, such as those arising from in-network compute. At the same time, we are also starting to see a variety of target devices. We believe that the programming model must facilitate writing portable, modular and composable programs. To support this, we introduce μP4 which raises the level of abstraction from target-specific packet-processing pipelines and constructs. With μP4, users can write target-agnostic, self-contained modules independently, and compose them to build larger programs while supporting a range of devices. We believe that μP4 will enable rapid innovation in dataplane programming as users can contribute to and build on portable libraries for packet-processing.

Acknowledgments. We thank the anonymous reviewers, our shepherd, Changhoon Kim, Dan Ports, and Vladimir Gurevich for their valuable feedback. This work was supported in part by NSF grant CNS-1413972, DARPA grant HR0011-17-C-0047, and gifts from Fujitsu, Google, and InfoSys.

References

[1] Barefoot Academy. 2019. Introduction to Data Plane Development with P416, Tofino and P4 Studio SDE. https://barefootnetworks.com/barefoot-academy/.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 113–126.

[3] 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. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*. ACM, Hong Kong, China, 99–110.

[5] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy,

et al. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, Los Angeles, CA, USA, 1–14.
[6] The P4 Language Consortium. 2013. `switch.p4` program. https://github.com/p4lang/switch/.
[7] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.
[8] The P4 Language Consortium. 2019. The BMv2 Simple Switch target. https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.
[9] The P4 Language Consortium. 2019. $P4_{16}$ Reference Compiler. https://github.com/p4lang/p4c.
[10] The P4 Language Consortium. 2019. $P4_{16}$ Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA-v1.1.0.html.
[11] The P4 Language Consortium. 2019. `v1model.p4` - Architecture for `simple_switch`. https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4.
[12] Advait Dixit, Kirill Kogan, and Patrick Eugster. 2014. Composing Heterogeneous SDN Controllers with Flowbricks. In *IEEE 22nd International Conference on Network Protocols (ICNP '14)*. IEEE, Raleigh, NC, USA, 287–292.
[13] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146–1170.
[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (July 1987), 319–349.
[15] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, Tokyo, Japan, 279–291.
[16] 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 2020 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*. ACM, Virtual Event, NY, USA, 1–14.
[17] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design principles for packet parsers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE, San Jose, CA, USA, 13–24.
[18] David Hancock and Jacobus van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, Irvine, CA, USA, 35–49.
[19] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. ACM, Seaside, CA, USA, 1–9.
[20] Internet Engineering Task Force (IETF). 2019. IPv6 Segment Routing Header (SRH). https://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-22.
[21] Internet Engineering Task Force (IETF). 2020. SRv6 Network Programming. https://tools.ietf.org/html/draft-ietf-spring-srv6-network-programming-15.
[22] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, Oakland, CA, 87–101.
[23] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, Oakland, CA, 103–115.
[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (Aug. 2000), 263–297.
[25] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII)*. ACM, College Park, MD, USA, 1–7.
[26] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, Lombard, IL, USA, 1–13.
[27] Barefoot Networks. 2019. Tofino. https://barefootnetworks.com/products/brief-tofino/.
[28] David A. Padua and Michael J. Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM (CACM)* 29, 12 (Dec. 1986), 1184–1201.
[29] Milad Sharif. 2018. Programmable Data Plane at Terabit Speeds. https://conferences.sigcomm.org/sigcomm/2018/files/slides/p4/P4Barefoot.pdf.
[30] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*. ACM, Florianopolis, Brazil, 15–28.
[31] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*. ACM, Florianopolis, Brazil, 44–57.
[32] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. μP4. https://github.com/cornell-netlab/MicroP4.
[33] Hardik Soni, Thierry Turletti, and Walid Dabbous. 2018. P4Bricks: Enabling multiprocessing using linker-based network data plane architecture. (Feb. 2018). https://hal.inria.fr/hal-01632431 (working paper or preprint).
[34] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, Budapest, Hungary, 518–532.
[35] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan RK Ports, and Aurojit Panda. 2020. Multitenancy for Fast and Programmable Networks in the Cloud. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud '20)*. USENIX Association, Virtual Conference, 8.
[36] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.
[37] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *26th International Conference on Computer Communication and Networks (ICCCN '17)*. IEEE, Vancouver, BC, Canada, 1–9.
[38] 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, Heraklion, Greece, 98–111.

Appendices

Appendices are supporting material that has not been peer reviewed.

A Interface Declarations in μPA

In Fig. 11, we present the declarations of `Unicast`, `Multicast` and `Orchestration` interfaces discussed in §4.1. Their runtime parameters allows programmers to express powerful forms of composition using the `apply` method.

B Example Multicast Program

Programmers can use `set_mc_group` method in actions or apply blocks to set a replication group for the packet. To replicate packets by spawning multiple linear μP4 pipelines, μPA's multicast engine extern provides two `apply` methods. Fig. 12 illustrates the use of an apply method that provides values for an instances of im_t and an out parameter of type `PktInstId_t` for each replica. The instance of `im_t` contains output port already set for the replica of the packet. The second argument provides packet instance identifier to identify the replica. Finally, the call to `enqueue` method of `mc_buf` joins all the forked pipelines. This is analogous to `pthread_join`.

The `set_buf` and `apply(pkt, im_t, out 0)` methods shown in Fig. 6 facilitate nested invocation of μP4 packages with `Multicast` interface in linear μP4 pipelines.

C μP4C Midend Transformations

Header stack transformation. μP4 allows the use of header stacks with known size at compile-time. μP4C replaces each header stack instance with multiple instances of the header type. It transforms

```
// Unicast
Unicast<H,M,I,O,IO>(pkt p, im_t im, in I in_param, out O out_param, inout IO inout_param) /*Runtime params*/ {
  parser u_parser(extractor ex, pkt p, out H hdr, inout M meta, in I in_param, inout IO inout_param);
  control u_control(pkt p, inout H hdr, inout M m, im_t im, in I in_param, out O out_param, inout IO inout_param);
  control u_deparser(emitter em, pkt p, in H hdr);
}

// Multicast
Multicast<H,M,I,O>(pkt p, im_t im, in I in_param, out_buf<O> ob) /*Runtime params*/ {
  parser m_parser(extractor ex, pkt p, out H hdr, inout M meta, in I in_param);
  control m_control(pkt p, inout H hdr, inout M meta, im_t im, inout I in_param, mc_buf<H,O> mob);
  control m_deparser(emitter em, pkt p, in H hdr);
}

Orchestration<I,O>(in_buf<I> ib, out_buf<O> ob) /*Runtime params*/ {
  control o_control(pkt p, im_t im, in I in_param, out_buf<O> ob);
}
```

Figure 11: Interface declarations in μP4's architecture.

```
control mc(pkt p, hdr_t h, im_t im,
     mc_buf<hdr_t, out_t> hb ) {
   mc_engine mce;   PktInstId_t id;
   out_t oa;  // some out args
   action replicate(GroupId_t gid) {
      mce.set_mc_group(gid);
   }
   table MulticastRouting{ key = { ... }
     actions = { replicate; }
   }
   table mac{ ...}
   apply {
     MulticastRouting.apply();
     mce.apply(im, id); //similar to C's fork
     mac.apply();
     hb.enqueue(h, im, oa);
   }
}
```

Figure 12: An example usage of Multicast extern.

operations on the header stack instances into appropriate built-in method calls. In P4 parser blocks, programmers can use `next` and `last` operations to iterate through the stack. These operations along with `lastIndex` can be used to write loops in parsers to extract instances in header stack. μP4C unrolls such loops by replicating the parse state and replaces the above operations with appropriate header instances in the state replicas. For `push_front` and `pop_front` operations on stack instances provided by P4, μP4C transforms them into a series of assignments and built-in method calls of header instances. For example, assume that `hs` is a header stack instance of size 3. μP4C will synthesize `hs0`, `hs1`, `hs2` header instances where `hs.hs_inst.push_front(1)` is replaced with `hs2 = hs1`, `hs1 = hs0` and `hs0.setInvalid()`.

Variable-length header transformation. While μP4 allows programmers to define variable-length header types, it imposes a constraint that their variable-length fields must contain integer number of bytes at runtime. μP4C splits any header types containing fixed and variable-length fields into multiple types, where each type contains either fixed-length fields or the variable-length field. μP4C transforms every parser state with two-argument `extract` method call, used to extract variable-length header, into a sub-parser. It splits the header type of the instance in the first argument into multiple header types, where each type contains either fixed-length fields or the variable-length field. Every state in the sub-parser extracts a fixed-number of bytes from the packet byte-stream. The sub-parser contains a state having the second argument (size of the variable-length field) as the expression in its `select` statement. The select statement has a case-list enumerating all possible values up to specified maximum size of the variable-length field. For each select case, the sub-parser transits to a state extracting a fixed number of bytes in the variable-length field. For example, if variable-length field has maximum size of 40 bytes, μP4C creates 40 states extracting different number of bytes. The explosion in terms of the number of states would only increase the number of entries in the transformed MAT for the parser.

Packet-processing schedule. If the main μP4 program or any of its callees performs compile-time packet replication using `copy_from` method of `pkt`, μP4C extracts threads and prepares PPS for the μP4 program with orchestration interface. μP4C constructs a Program Dependence Graph (PDG) [14] having statements as nodes and dependencies among them as edges. It performs a series of transformations on PDG to compute PPS defined in §5.4.

For every initialization statement of a `pkt` instance, we define an *access-range*, which is analogous to the concept of live range. It is defined as a span of program statements, on every possible path in the PDG, until the next initialization of the instance is reached. We merge overlapping access-ranges of multiple initializations of the same pkt instance. If an instance has non-overlapping access-ranges, we synthesize a new instance for every access-range to create a one-to-one mapping between them.

Inspired by *program slices* [36], we define a *packet slice* based on access-ranges. A packet slice of a `pkt` instance is an executable subset of PDG which consists of all the program statements affecting the instance's value in its access-ranges. A set of initialization statements of a `pkt` instance with overlapping access-ranges is considered as a slicing criterion for a given packet instance. Packet slices may have multiple entry and exit points due to the presence of conditional statements. We compute packet slices from the PDG of control block using a method similar to one described in [14]. Essentially, we perform a graph traversal in the reverse direction

```
struct e_t {};  struct h_t { ... };
prog(pkt, im_t, out h_t);
test(pkt, im_t, out h_t);
log(pkt, im_t, in h_t, in h_t);
control validate(pkt p, im_t i, out_buf<e_t> ob) {
          pkt pt, pm;  im_t it, im;
/*slice*/ h_t hp, ht;
/*  #  */ apply {
/*  1  */   pm.copy_from(p); // c1
/*  1  */   im.copy_from(i);
/*  3  */   pt.copy_from(p); // c3
/*  3  */   it.copy_from(i);
/* 2,1 */   prog.apply(p, i, hp);
/* 3,1 */   test.apply(pt, it, ht);
/*  1  */   if (hp != ht) {
/*  1  */     log.apply(pm, im, hp, ht);
/*  1  */     ob.enqueue(pm, im);
/*  1  */   }
/*  3  */   im.set_out_port(DROP);
/*  2  */   ob.enqueue(p, i);
/*  3  */   ob.enqueue(pt, it);
          }
       }
```

Figure 13: Slicing for multi-packet processing.

of edges from every exit point statement in the access ranges until the initialization statements in the criteria are reached. The graph traversal continues until all possible definitions of every variable used in the visited statements are reached. If any variable or extern instance is involved in anti-dependency, we resolve it by introducing a copy of the variable. Fig. 13 shows an example program that is sliced with respect to three instances: `pm`, `p` and `pt`.

Packet slices of different instances may have common program statements due to control and data dependencies. Therefore, a packet slice may have method call statements processing different `pkt` instances than the initialization statements used in slicing criteria. For the program shown in Fig. 13, slice 1 pertaining to instance `pm` shares program statements with slices 2 and 3 related to `p` and `pt`, respectively. We create a packet-processing thread per instance by excluding such method call statements from every packet slice but we maintain dependencies by creating inter-thread dependency. All other common program statements are excluded from threads of `pkt` instances. We term such statements *CPS* nodes and maintain their control and data dependencies with the thread nodes. We associate every thread with an identifier, *thread-id*, and synthesize metadata for it. μP4C synthesizes if-conditional statements in every thread to execute its program statements only if its id value is set in thread-id metadata. We realize PPS by using target-specific replication constructs (e.g., clone functions for V1Model) and setting next ids of next threads in `thread-id` metadata. We follow a similar approach to extract threads associated with `in_buf` instances. The `merge` and `to_in_buf` methods of `out_buf` allow to add thread nodes and control dependency among them in PPS.

We transform PDG to PPS graph by coalescing all the nodes in a thread to a single node while maintaining their control and data dependencies with CPS and other thread nodes. Then, we synthesize a variable to transform every control dependency among thread nodes to a data dependency. The thread, which is depended on, sets the variable with a constant value and the other thread uses the variable in the predicate of the new if-conditional statement to continue processing on the control path. If a PPS has a directed cycle involving thread nodes, PPS is not serializable. If the target architecture does not have the capability to process multiple copies of a packet at the same time, μP4C raises an error and lists program statements involved. However, a PPS can have cycles involving at most one thread node and one CPS node. To determine the execution thread for CPS nodes, we compute strongly connected components (SCCs) of the PPS graph. In each component, there can be exactly one thread node and one or more CPS nodes. We further transform the PPS into a DAG by coalescing CPS nodes in a component to its thread node. PPS can still have CPS nodes not part of any SCC, such nodes can be executed as a part of any thread. We schedule such CPS nodes while mapping the threads to the programmable blocks of the target architecture.