Modeling hardware blocks of network ASICs using P4

Arthur Simon, Hardik Soni, Khaled Diab, Puneet Sharma
Hewlett Packard Labs
first.last@hpe.com

Abstract

We present a novel use case of the P4 language. We use the P4 tool chain and the reference P4 software switch, BMv2, to develop a register-level functional simulator for network ASICs which are going under hardware design and development. Our requirement is to facilitate functional testing of ASICs' features, their interfaces with the driver and entire software stack. We describe our experience in using the constructs of the P4 language to model fixed-function parsers, deparsers, TCAMs and RAMs commonly used in hardware design.

1 Motivation and Use Case

Emerging Network Interface Cards (NICs) and switch ASICs designs maximizes the packet-processing by specialized hardware blocks to support low-latency and high throughput networks. The next generation networking ASICs are feature-rich with proprietary and standardized packet-processing stacks implemented with hardware accelerators. Due to complexity of hardware design, the ASICs' functionalities are required to be verified and tested with the drivers and software stack during design and implementation phase.

Without a simulator for the ASIC, development and functional verification of drivers and software stack is challenging. Therefore, a software tool chain is needed to specify and model hardware functionality of the ASICs during the early phase of design and development.

The P4 language is designed to describe packet-processing behaviour of re-configurable network devices. However, it can also be used to describe the functionality of network devices with fixed-function ASICs. We take a step further and use the P4 language to build a simulator for network device based on fixed-function ASICs by capturing the hardware components and operations, precisely.

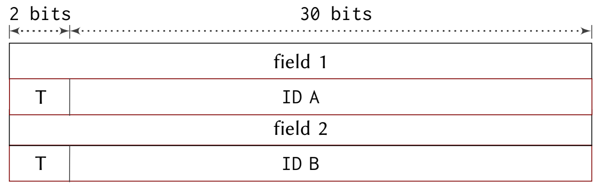
Although the P4 language is designed and widely used for describing packet-processing behaviour for re-configurable ASICs, surprisingly enough, P4 can also be used, *with some extensions*, to describe the design of network ASICs detailing the fixed-function hardware components and their interconnection.

The fixed-function network ASICs for custom protocol stacks are implemented using optimized hardware blocks. The match-action abstractions are implemented using RAMs, Hash-tables in addition of TCAMs. Although the language constructs of P4 abstract out the implementation details, our requiement is to capture the operations as implemented in the hardware. We report various possible enhancements to capture component level hardware implementation of fixed-function network ASICs.

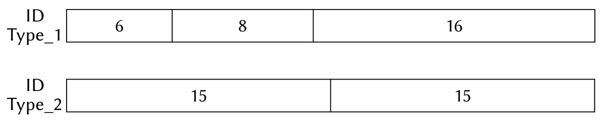
2 Overloaded Fields in Protocol Headers

For hardware optimization, headers of custom protocol stacks are designed to maximize the reuse of fields in headers. Normally, such optimizations are performed by compilers to optimize resource usage on re-configurable targets. It helps to reduce the size of bus carrying parsed header fields. Although the `struct`, `header`, `header_union`, `header_stacks` are not designed to describe or encode such optimizations, they are still powerful enough to capture such optimization with a simple relaxation in the P4 language.

Consider an example of a header shown in Figure 1a. The header consists of two 32-bits wide fields, `field 1` and `field 2`. In addition, the header contains two pairs of a type (`T`) and an identifier (`ID A` and `ID B`) fields. Each identifier field is further parsed using one of the formats shown in Figure 1b based on value of type fields, `T`.



(a) Main header



(b) Identifier Formats

Figure 1: Examples of a main header and its fields formats

With the current constructs of P4 language, the identifier fields should be considered as an instance of identifier headers because they extract bit-stream in different ways. Therefore, to parse the main header having fields functionally associated to each other, we need to define multiple P4 headers and parser states as show in Figure 2.

```
header id_1_h {
  bit<2>      type;
  bit<6>      if1;
  bit<8>      if2;
  bit<16>     if3;
}
header id_2_h {
  bit<2>      type;
  bit<15>     if1;
  bit<15>     if2;
}
header_union id_t {
  id_1_h      id_t_1;
  id_2_h      id_t_2;
}
header main_part_1_h {
  bit<32>     f1;
}
header main_part_2_h {
  bit<32>     f2;
}
struct headers_t {
  ... // other headers
  main_part_1_h   mh1;
  id_1_h          ida;
  main_part_2_h   mh2;
  id_2_h          idb;
}
```

```
parser id_parser(packet_in b, out id_t id) {
  state start {
    transition select(b.lookahead<bit<2>>()) {
      (2w0): parse_type1;
      (2w1): parse_type2;
    }
  }
  state parse_type1 {
    b.extract(id.id_t_1);
    transition accept;
  }
  state parse_type2 {
    b.extract(id.id_t_2);
    transition accept;
  }
}
parser p(packet_in b, out headers_t h...) {
  state start {
    b.extract(h.mh1);
    id_parser.apply(b, h.ida);
    b.extract(h.mh2);
    id_parser.apply(b, h.idb);
  }
}
```

Figure 2: headers for parser definitions using P4

The P4 code in Figure 2 requires to split functionally related fields into multiple P4 header types. The fundamental reason is `ID 1` and `ID 2` shown in Figure 1a must be encoded as separate P4 headers, `id_1_h` and `id_1_h` (as shown in Figure 2), because the packet bit-stream must be parsed in different layouts for different types of ID fields. It should be noted that all the identifier (`ID`) fields have the same size. Such packing of fields allows efficient reuse hardware resources.

If the pairs of type and identifier fields are heavily used in the custom protocol stack, the P4 header definition and P4 parser require verbose and lengthy encoding. The `header` and `struct` definitions using P4 require over engineering compared to the languages like C/C++. Importantly, it makes difficult to integrate P4 code with other legacy tools written in C/C++. The requirement to split functionally associated fields into different headers puts P4 at relative disadvantages against general-purpose languages, even though P4 is domain-specific and should be able to provide powerful ways to encode packet parsing.

The example shown in Figure 1a is different than parsing TCP options. The TCP options are parsed on only at the end of the mandatory fields. Also, the bit-width of TCP options can be different.

C/C++ being extremely expressive and powerful general-purpose programming languages, they allow complex nesting of derived types. `unions` and `structs` of C/C++ are of primary interest in our case. An union of identifier fields (Figure 1b) as a member of the main header(Figure 1a) would allow more compact and C/C++ like definitions.

Relaxing Type Nesting Rules for P4 Container Types

P4 allows nesting of derived types with certain exceptions as described in [1]. `header` types have special semantics compared to `struct` because of `extract`, `setValid`, `setInvalid`, and `emit` operations on instances of `header` types.

P4 allows to have `header` types as member of `header_unions`. For example, `id_t union` can have members with `id_1_h` and `id_2_h` types. `header_unions` can be a member of a struct type `type_id_t`. `struct` types can be used as members of header types only if the struct contains only bit⟨W⟩, int⟨W⟩, a serializable enum, or a bool types as mentioned at [2].

```
header id_1_h {
  bit<6>      if1;
  bit<8>      if2;
  bit<16>     if3;
}
header id_2_h {
  bit<15>     if1;
  bit<15>     if2;
}
header_union id_t {
  id_1_t      id_t_1;
  id_2_t      id_t_2;
}
struct type_id_t {
  bit<2>      type;
  id_t        id;
}
header main_h {
  bit<32>     f1;
  type_id_t   id_a;
  bit<32>     f2;
  type_id_t   id_b;
}
struct headers_t {
  ... // other headers
  main_h      mh;
}
```

```
parser p(packet_in b, out headers_t h...) {
  state start {
    b.extract(h.mh);
    id_parser(h.mh.id_a.type, h.mh.id_a.id);
    id_parser(h.mh.id_b.type, h.mh.id_b.id);
  }
}
parser id_parser(bit<2> type, header_union id) {
  state start {
    b.extract(h.mh);
    transition select(type) {
      (2w0): parse_type1;
      (2w1): parse_type2;
    }
  }
  state parse_type1 {
    id.id_t_1.setValid();
    transition accept;
  }
  state parse_type2 {
    id.id_t_2.setValid();
    transition accept;
  }
}
```

Figure 3: Relaxing Container Type Nesting Rules

We use `header_union` as a member of a `struct` nested in a `header` type as shown in Figure 3. It essentially enables indirect nesting of header types which obviously raises many questions on defining behaviour of `setValid`, `setInvalid` and `emit` operations as explained at [3].

Once a packet is parsed, a set of headers extracts byes to their fields and their validity bits are set. From the start state to accept state, every extracted header is set to valid state.

We define the validity of headers based on validity of all their members. A header is valid only if all of its members are valid. `setInvalid` operations on a header should invalidate all its members, recursively. every `setValid` operation on a header requires all its members to be valid. Therefore, all the member `headers` and `header_union` should be explicitly set to valid before their containers are set to valid. Compilers can make these checks at compile-time.

Figure 3 shows parsing of the P4 headers using with nested P4 `header` and `header_union` types for the example in Figure 1. These approach allows simplified and C/C++ like definitions for custom packet headers.

3 Match-Action units and Tables

In order to model all the hardware components and their operations, we need to model TCAMs with capability to add, update and delete entries at user specified locations. TCAMs are usually configured using control registers [4]. As a part of lookup process, the outcome should be the address of the matching entry in the TCAM. We model every RAM in the ASIC using a P4 `table` with address as a key. Concretely, TCAMs are used as a lookup table. If a packet matches to an entry in a give TCAM, the index or address of the entry in the TCAM is further used to locate the action data in the associated RAM.

The `table` constructs of P4 describe match-action units. The abstraction of the match-action unit is too coarse-grained to encode hardware components like TCAM, RAM and Hash-tables. The match-action unit model comprises of a lookup-table and action code and data RAM whereas we need to model TCAMs, which are only lookup tables, and RAM separately. Also, the control plane APIs for P4 tables do not allow to insert or modify entry at the user specified locations in the tables.

TCAM using P4 `table`: The current behaviour model(BMv2) inserts a new entry at the first empty location in

```
bit<10> match_address;
action index_store(bit<10> index) {
  match_address = index;
}
table tcam_lookup {
  key = {...}
  actions = {
    index_store;
  }
  size=1024;
}
table RAM {
  key = { match_address: exact; }
  actions = {
    ...;
  }
  size=1024;
}
apply {
  if (tcam_lookup.apply().hit) {
    RAM.apply();
    // use match_address
    // for further processing
  }
}
```

(a) Explicitly storing index using action parameter

```
bit<10> match_address;
table TCAM_RAM {
  key = {...}
  actions = {
    ...;
  }
  size=1024;
}
TCAM_RAM.apply();
if (TCAM_RAM.result.hit) {
    match_address = TCAM_RAM.result.index;
    // use match_address
    // for further processing
  }
}
```

(b) Index as a part of `table.apply()` evaluation result

Figure 4: P4 tables and actions modeling detailed match-action operations

the table. We expose another set of control plane APIs for P4 tables to add, update and delete entry at the specified locations. Only either of the two sets of APIs should be used to manage a given P4 table to avoid race conditions.

Figure 4 shows two alternative approaches for modeling detailed operations of TCAMs and RAMs used in fixed-function network ASICs. Figure 4a shows use of `action` parameters to store index explicitly as a part of value for the P4 table. This techniques requires index as action parameter for every action of the table. Figure 4b requires only single table because evaluating `TCAM_RAM.apply` method store results(hit or miss and index of matched entry) in the `result` property of the table.

4 Conclusion

We are currently implementing the above mentioned P4 extensions to enable hardware component (including control-state registers) level simulation of ASICs for networking devices. Although the P4 language is designed to provide packet-processing primitives and hide architecture-specific implementations, we believe that the language is expressive enough to encode component level design and architecture of fixed-function ASICs.

References

[1] The P4 Language Consortium. P416 language specification version 1.2.4, 2024. Last accessed 09 August 2024. URL: `https://p4.org/p4-spec/docs/P4-16-v1.2.4.html#sec-type-nesting`.

[2] The P4 Language Consortium. P416 language specification version 1.2.4, 2024. Last accessed 09 August 2024. URL: `https://p4.org/p4-spec/docs/P4-16-v1.2.4.html#fn-struct_header`.

[3] The P4 Language Consortium. P416 language specification version 1.2.4, 2024. Last accessed 09 August 2024. URL: `https://p4.org/p4-spec/docs/P4-16-v1.2.4.html#sec-header-types`.

[4] Intel. Agilex™ 7 f-series and i-series fpga memory subsystem ip user guide, 2024. Last accessed 09 August 2024. URL: `https://www.intel.com/content/www/us/en/docs/programmable/789389/24-1-2-0-0/result-n-64289.html`.