Concurrency in P4

September 30, 2016

1 Motivation

P4 lacks a formal concurrency model. This is important when considering interactions between multiple packet processors in the data plane. These packet processors could be match-action tables, pipelines, or cores. Further, these packet processors can share state. As an example, consider flowlet switching from the SIGCOMM 2015 P4 tutorial. The state stored in register `flowlet_id` is shared by two tables:

1. It is read in the action `lookup_flowlet_map` in the `flowlet` table
2. It is later written in the action `update_flowlet_id` in the `new_flowlet` table.

Here's the relevant code snippet for flowlet switching, written in P4-16.

```
control ingress {
  ...
  ...
  apply {
    apply(flowlet);
    if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TOUT) {
        apply(new_flowlet);
    }
    apply(ecmp_group);
    apply(ecmp_nhop);
    apply(forward);
  }
}
```

In the snippet above, `flowlet` reads a piece of state (`flowlet_id`) that `new_flowlet` writes. Because there is no support for locking within P4, this could lead to a race, where a second packet reads an old value of `flowlet_id`. It is unclear what behavior to expect for these programs because P4 lacks a formal concurrency model when state is modified in the data plane. This draft is an attempt at fixing this.

2 Model

We first specify a formal model for concurrency below. We provide a high-level overview and then show how it can be applied to various productions within the P4 grammar. The compiler section of this draft discusses compilation strategies.

An instance of a parser or control block type (§9.2.8 of the spec) is called a parser or control block instance. All computations within P4 happen inside these instances as specified by the statements (§12) within them. We say a parser or control block is "invoked" when some external event triggers the execution of a parser instance or a control block instance. This external event could be the arrival of an unparsed packet (`packet_in`) for a parser instance or parsed headers (`in` or `inout` function arguments) for a control block instance.

When an event invokes a parser or control block instance, we can conceptually think of the instance launching a new thread to handle that event in the background. The instance is now free to handle the next event. The concurrency model doesn't specify the implementation. For instance, the number of threads in a thread pool is implementation-defined. In fact, the implementation doesn't even need to use a thread pool. It can instead use a processing pipeline, where a small portion of the work for each event is handled by one thread, which then passes off the event to the next thread (e.g., match-action pipelines).

Each of the launched threads runs to completion, executing all statements within the control block or parser instance one after another, sequentially. Once the thread completes processing an event within an instance, it triggers an event within another parser or control block instance, as specified by the target architecture.

Any interleaving of threads is permitted as long as statements within a thread aren't reordered. The atomic (or instantaneous) units of execution within each thread's statements are defined by providing an `@atomic` annotation around specific statements or expressions in the program.[1] This annotation denotes that the computations within those statements or expressions appear to execute atomically/instantaneously.

In practice, the P4 compiler annotates many expressions and statements atomic by default, to reduce the burden on the P4 programmer. We dicuss these below.

2.1 atomic by default

Statements and expressions that are atomic by default are all statements and expressions that do not access an extern instance, i.e., statements or expressions that read or write the following types of data:

1. intrinsic metadata (§4)
2. metadata (§4)
3. packet headers (§4)
4. local variables declared within a control block's apply block, a parser's state, or an action's body (§11.2)[2].

We also include statements that recursively only include statements that read or write the above types of data. An example is a statement that calls an action, which respects the conditions above.

The rationale for not including extern instances in the atomic-by-default list above is that extern instances encapsulate data-plane state, unlike the scenarios above that don't include any state visible to all threads. State modification through extern method calls cannot be declared atomic by default because depending on the extern method, some intermediate state may be visible to another thread, which contradicts the requirement that modifications be instantaneous. We recommend that method calls on extern instances be implemented as atomic by target architecture implementers, e.g., serializing all calls to a counter or register to provide the illusion of atomicity. An atomic method call can be specified by using the `@atomic` notation within an extern declaration (the next section provides an example).

Certain combination of statements are also annotated atomic by default. As an example, consider the statements:

```
x = x & 4w0;
```

and

```
x = 4w0 & x;
```

, where x is a local variable. Then the block statement:

```
{
  x = x & 4w0;
  x = 4w0 & x;
}
```

is atomic by default because it calls no extern instance.

Certain categories of statements are explicitly not atomic by default. In particular, while it is recommended that individual method calls on extern instances be atomic, a pair of calls to two different extern instances is not atomic by default. An example is a pair of counters, say, `c1` and `c2`. While the increment method call for each counter appears to execute atomically, other statements can be interleaved between the increments to `c1` and `c2`, unless the programmer encloses both method calls within one atomic annotation.

The above discussion should give a sense for how atomic is used in practice. To reduce programmer burden, we recommend that P4 compilers implement a inference procedure for conservatively inferring whether any given production within a grammar is atomic by default. The procedure could be modeled based on some of the rules given above.

3 Examples

With the atomic annotation, the flowlet switching example can now be written in one of two ways. We later discuss pros and cons of each approach.

[1] An expression can be treated as a statement by assigning the result of the expression to a temporary variable.

[2] Right now, P4 doesn't permit extern instances as local variables. The FAQ section of this draft expands on this.

3.1 Using registers

```
control ingress {
  ...
  ...
  apply {
    @atomic{
      apply(flowlet);
      if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TOUT) {
          apply(new_flowlet);
      }
    }
    apply(ecmp_group);
    apply(ecmp_nhop);
    apply(forward);
  }
}
```

This has the correct behavior of *atomically* reading the register `flowlet_id`, and updating it (by incrementing it: `https://github.com/p4lang/p4c/blob/master/testdata/p4_14_samples_outputs/flowlet_switching.p4#L145`) if the condition within the if clause is true. It assumes the standard library extern element `register` (§8.5.2) supports an atomic read and write, but nothing more.

Underneath, a compiler would have to turn this conditional read-modify-write into a hardware instruction that atomically updates a piece of memory if a condition is true. The compiler section of this draft discusses how this compiler would work.

3.2 Using more complex extern types

We could imagine an extern type that provides a thin wrapper around a hardware instruction that allows a program to atomically increment a state variable if a boolean condition is true. A possible declaration for this extern type is given below.

```
extern Conditional++ {
  @atomic void conditional_inc(bool condition);
  @atomic int read();
}
```

Here, `conditional_inc` is declared an atomic method call by the implementer of the target architecture who specifies the behavior of `Conditional++`. The behavior of `Conditional++`'s `conditional_inc` method is to atomically increment an internal state variable if condition is true.

With `Conditional++`, flowlet switching can now be written as:

```
control ingress {
  // Declare a Conditional++ object shared across all control block threads
  Conditional++ conditional++;
  apply {
    conditional++.conditional_inc(ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TOUT);
    apply(ecmp_group);
    apply(ecmp_nhop);
    apply(forward);
  }
}
```

3.3 More examples

We list out another example in both forms (using registers and using more complex extern types). This example is drawn from `https://github.com/packet-transactions/domino-examples/tree/master/domino_programs`, which is a set of examples written in a high-level packet-processing language called Domino. Domino is centered around user-defined atomic blocks. Hence, using our `@atomic` notation, every Domino program can be transformed into an equivalent P4 program using an enclosing `@atomic`—so long as the standard library supports registers with atomic read and write method. This transformation is fairly mechanical:

1. Where Domino uses a state variable, we use a register in P4.
2. If Domino assigns to a state variable, we use register.write() in P4.

3. Similarly if Domino accesses a state variable, we use register.read() in P4 to read it into a temporary local variable.
4. We then use the temporary local variable in place of the state variable.
5. So, the Domino expression `s = s + 1;`, where s is a state variable with a 32-bit int type, would turn into:

```
Register<bit<32>> (1) s;
bit<32> s_tmp = register_read(0, s);
s_tmp = s_tmp + 1;
register_write(0, s_tmp);
```

Rate-Control Protocol (RCP) With registers alone, RCP can be written like this.

```
control ingress {
  Register<bit<32>>(1) input_traffic_Bytes;
  Register<bit<32>>(1) sum_rtt_Tr;
  Register<bit<32>>(1) num_pkts_with_rtt;
  ...
  apply {
    @atomic{
      bit<32> input_traffic_Bytes_tmp;
      input_traffic_Bytes.read(0, input_traffic_Bytes_tmp);
      input_traffic_Bytes_tmp = input_traffic_Bytes_tmp + metadata.pkt_len;
      input_traffic_Bytes.write(input_traffic_Bytes_tmp, 0);
      if (pkt_hdr.rtt < MAX_ALLOWABLE_RTT) {
        bit<32> sum_rtt_Tr_tmp;
        sum_rtt_Tr.read(0, sum_rtt_Tr_tmp);
        sum_rtt_Tr_tmp = sum_rtt_Tr_tmp + pkt_hdr.rtt;
        sum_rtt_Tr.write(sum_rtt_Tr_tmp, 0);

        bit<32> num_pkts_with_rtt_tmp;
        num_pkts_with_rtt.read(0, num_pkts_with_rtt_tmp);
        num_pkts_with_rtt_tmp = num_pkts_with_rtt_tmp + 1;
        num_pkts_with_rtt.write(num_pkts_with_rtt_tmp, 0);
      }
    }
  }
}
```

With more complex extern types, it would be written as:

```
extern ConditionalAccumulator<T> {
  ConditionalAccumulator();
  void read(out T result);
  void write(in T accumuland, bool condition);
}
...
control ingress {
  counter<bit<32>>(1, CounterType::packets_and_bytes) input_traffic_Bytes;
  ConditionalAccumulator<bit<32>>(1) sum_rtt_Tr;
  ConditionalAccumulator<bit<32>>(1) num_pkts_with_rtt;
  ...
  apply {
    ...
    @atomic{
      input_traffic_Bytes.count();
      sum_rtt_Tr(pkt_hdr.rtt, pkt_hdr.rtt < MAX_ALLOWABLE_RTT);
      num_pkts_with_rtt(1,    pkt_hdr.rtt < MAX_ALLOWABLE_RTT);
    }
  }
}
```

3.4 Comparing the two approaches

The first approach using registers leads to portable code, because it only assumes simple registers with atomic read and write capabilities.[3] It also allows the target implementer to hide potentially propreitary information regarding how a conditional update is actually implemented, i.e., the target implementer doesn't need to provide a method that reveals the fact that it does a conditional increment or a conditional accumulate. With the second approach, the target implementer needs to specify an external method for every stateful update that can be carried out using the target's hardware capabilities. This could prove cumbersome.

However, the first approach puts the burden on the compiler to figure out exactly how to translate a block of programmer statements within a `@atomic` into hardware instructions. It also leads to the possibility that the programmer writes code that is rejected because the atomic block is too large to implement using an atomic hardware instruction and guarantee atomic semantics. This could be potentially solved by better compiler error messages telling the programmer how to fix the program, but it adds to compiler complexity once again.

4 Compiling atomics

We outline a few strategies for compiling atomics to different target architectures. We focus on the general case of user-defined atomic blocks. The default atomic annotations for stateless constructs and atomic method calls require very little work in the compiler.

4.1 Pipelining into match-action pipelines

On a pipelined switch architecture, such as the RMT architecture or Intel's FlexPipe, state is local to a stage of the pipeline because it is technically challenging to share state between pipeline stages through multi-ported memories.

This means all read-modify-write patterns on a given piece of state have to be completed within a single stage of the pipeline. Given an arbitrary large atomic block, the compiler first needs to decompose this into as many atomic blocks as possible without violating the atomicity of the original large atomic block. Each of these new atomic blocks needs to be mapped to a stage of the pipeline, while respecting dependencies between them.

As an example, suppose the compiler were handed the code fragment below (here `x_register` and `y_register` are register arrays of size 1).

```
apply {
  @atomic {
    bit<32> x_tmp;
    x_tmp = x_register.read(x_tmp, 0);
    x_tmp = x_tmp + 1;
    x_register.write(0, x_tmp);
    bit<32> y_tmp;
    y_tmp = y_register.read(y_tmp, 0);
    y_tmp = y_tmp + x_tmp;
    y_register.write(0, y_tmp);
  }
}
```

The compiler would then whittle it down into the smaller atomic blocks.

```
apply {
  @atomic {
    bit<32> x_tmp;
    x_tmp = x_register.read(x_tmp, 0);
    x_tmp = x_tmp + 1;
    x_register.write(0, x_tmp);
  }
  @atomic{
    bit<32> y_tmp;
    y_tmp = y_register.read(y_tmp, 0);
    y_tmp = y_tmp + x_tmp;
    y_register.write(0, y_tmp);
  }
}
```

[3] While the RCP example above suggests that registers lead to overly bloated code, this could be easily fixed by using programmer-defined atomic actions that provide a wrapper around the boilerplate of reading from a register, then modifying, then writing it back.

It would then infer that the second atomic block depended on the output of the first and respect these dependencies when mapping them to a pipeline.

The second step is actually mapping these newly minted atomic blocks to hardware instructions that implement them. For example, a simple increment instruction suffices for the first block, while we need an atomic read-add-write for the second one.

A more detailed explanation of these two steps is available in Sections 4.2 and 4.3 of the paper here: `http://dl.acm.org/citation.cfm?doid=2934872.2934900`

4.2 Pipelining into NPU pipelines

A similar approach can be used for NPU pipelines as well. The first step is identical to the one above. The second step is different. After whittling down a programmer's atomic block into smaller atomic blocks, each resulting atomic block can be mapped to a processor core or microengine, instead of an atomic hardware instruction.[4]

4.3 Privatization

In some cases, it may be possible to compute a stateful variable independently in two different threads using a private copy in each thread, merging them later. An example is a counter, which can be implemented using a private counter in each of the threads in a thread pool, merging them later.

5 Frequently asked questions

1. What happens when the controller changes a table entry while a packet is being processed in the data plane?
   This draft doesn't deal with interactions between the control and data plane, because the control plane isn't in P4 and P4 cannot easily mandate its behavior. A reasonable semantics for this case is to say that a packet either sees the new table entry or the old one, but not a muddled combination of the two.

2. What about nested atomic blocks?
   We can take one of two equivalent approaches. Either, we forbid nesting atomic blocks within each other, or the compiler removes all nested atomic blocks and preserves only the outermost atomic annotation. This is the SQL approach to nested transactions (the database analogue of nested atomic blocks). Here one transaction calls a stored procedure, which might itself run as a transaction. When the stored procedure (the inner transaction) commits, it has no effect until the outer transaction commits, at which point it is commited as well (`https://technet.microsoft.com/en-us/library/ms189336(v=sql.105).aspx`).

3. Atomic guarantees sound hard to support?
   Yes, they probably are. They require work in the compiler to support the general case of user-defined atomic blocks, which go beyond the default atomic annotations. They also require work in the compiler to provide reasonable error messages when an atomic block is rejected.

   However, I contend it is better than the alternative, which is to expect the programmer to invoke target-specific atomic methods. This is both unportable and requires a target implementer to unnecessarily reveal the target's details to the programmer. Besides, implementing the required logic in the compiler isn't all that hard: it is around a few hundred lines of code to do the required program analysis to whittle down a programmer's atomic block to minimal atomic blocks. To deal with the problem of rejecting large atomic blocks, we could conservatively restict the complexity of the code within the atomic block, just like we do with the complexity of the expression within P4's `verify` statement today.

4. Are we going to have an entire spectrum of weak/relaxed atomicity models, like every language that has gone down the garden path of atomic blocks?
   No. In cases where this has happened (such as C++: `https://3f993110-a-62cb3a1a-s-sites.googlegroups.com/site/tmforcplusplus/C%2B%2BTransactionalConstructs-1.1.pdf` and `http://en.cppreference.com/w/cpp/language/transactional_memory`), this was the result of interoperability with existing forms of synchronization such as locks, volatiles, condition variables, and mutexes, which didn't provide the same guarantees as atomic blocks, and which could be accessed both within atomic and non-atomic blocks.

   This spectrum of models is not fundamental. For instance, Haskell's STM implementation (`http://community.haskell.org/~simonmar/papers/stm.pdf`) doesn't have relaxed or weaker forms of atomicity. We can take the same approach with P4 and make atomic the only form of concurrency.

[4] This arrangement is called a context pipeline because each stage in the pipeline carries out a different context of packet processing, e.g., classification, policing, metering, etc.: §3.2.1 of `http://www.cs.ucr.edu/~bhuyan/cs162/IXP2400.pdf`.

5. What about exceptions within @atomic block?
   Typically, database transactions abort on exceptions. It's as though the transaction never ran. P4 doesn't have an exception handling mechanism (the verify statement within the parser state machine can be viewed as syntactic sugar for transitioning to the reject state, just like any other statement that transitions to the reject state in the parser state machine). Without exception handling, every @atomic block runs to completion.

6. Can operations on local externs be made atomic by default?
   Yes. Except there is no such language construct for local externs today. Local variables within a parser body or a control block's apply method cannot be externs. All externs declared within a parser or control declaration are global and shared by all instances by default. This would be useful for externs such as parser checksums that run an independent instance for each thread.