Formalizing Semantics of `par` #932

rachitnigam started this conversation in **Semantics**

rachitnigam on Feb 23, 2022 Maintainer

TLDR: `par` blocks in Calyx currently do have well-defined guarantees on what kind of cross-thread communication is allowed. This proposal formalizes Calyx's memory models as: Data race free or nothing (DRF0) by default, and `@sync` annotation for communication.

See Semantics of `par` (#921) for background discussion

DRF0 by Default

By default, `par` blocks in Calyx are not allowed to "communicate" with each. Communication is defined as any possible data race between the ports of a cell or its internal state. For example, the following innocuous program is incorrect:

```
x = std_reg(32);
y = std_reg(32);

group g1 {
  x.in = y.out;
  x.write_en = 1'd1;
  g1[done] = x.done;
}
group g2 {
  y.in = x.out;
  y.write_en = 1'd1;
  g2[done] = y.done;
}

control {
  par {
    g1;
    g2;
  }
}
```

In normal hardware design languages (HDLs), we would expect the program to swap values between `x` and `y` after one cycle. However, Calyx programs make no guarantees about the scheduling of `par` blocks. This means that the compiler is free to assume that `g1` and `g2` may start executing in the same cycle, or that `g1` executes 100 cycles after `g2`. The latter is just an egregious example but demonstrates that complete lack of guarantees that `par` blocks provide.

The lack of guarantees in the language model is a *good thing*---too many guarantees means that the compiler is limited in the kinds of optimizations it can perform.

`@sync` Annotation for Synchronization

Obviously, some programs do require some notion of synchronization when executing parallel threads. We propose a new annotation, called `@sync` that enables threads to declare that events other threads are visible to them.
For example:

```
par {
  seq {
    a;
    @sync(1) b;
    c;
  }
  seq {
    d;
    e;
    @sync(1) f;
  }
}
```

In the program above, the compiler needs to guarantee that the execution of `b` can observe the events from `a` (in its own thread) and `d` and `e` in the second thread.
However, *it does not* guarantee that both `b` and `f` start executing in the same cycle.

The number `n` in the `@sync(n)` annotation acts as the name of the barrier and can be referred to the by many different threads. Programs are free to use as many `@sync(n)` as needed.

Well-Formedness Constraint

A required well-formedness constraint for `@sync` annotations is that the same thread may not mention the same barrier in sequentially ordered events. For example, the following is allowed:

```
par {
  if lt.out {
```

```
      a;
      @sync(1) b;
    } else {
      @sync(1) c;
      d;
    }
    seq {
      f;
      @sync(1) g;
    }
  }
```

Since `b` and `c` occur in two branches of `if`, they are not sequentially ordered with respect to each other. However, the following `@sync` annotations are ill-formed:

```
par {
  seq {
    @sync(1) a;
    b;
    @sync(1) c;
  }
  seq {
    d;
    @sync(1) e;
  }
}
```

This is because in the first thread, execution of `a` and `c` are sequentially ordered--- `a` must execute before `c` and therefore it is impossible for `a` to view events before `c` (which includes `a` itself).

Optimizing with Barriers

Passes should be able to analyze the various barriers in the program and use that information to reason about ordering of execution of various groups. A simple litmus test of whether we have correctly implemented these semantics is if resource sharing can share components across various threads when it can establish specific ordering relationships.

Compiling Barriers

The compiler is responsible for reifying the barriers in the final program and maintaining their invariants. A natural choice for this is using synchronizing registers which implement M-structures useful for blocking reads and writes.
An implementation challenge is generating "multi-armed" synchronizing registers that can be referred to by many different threads.

Synchronizing with Static Time

The default `@sync` primitive only establishes an ordering between parallel threads and does not guarantee that, for example, two groups will start executing at the same time. This kind of clock-based synchronization can be useful for certain programs that *do need* to make use of clock-based tricks for optimizations.

While we currently don't have a concrete proposal for this, here are our aspirations:

1. Make use of the `@sync` annotation in conjunction to `@static` to enable this kind of reasoning.
2. Keep static reasoning *local* and compositional with the rest of the dynamic control program.
3. Say something specific about timing guarantees across component boundaries.

↑ 2 ☺

3 comments

Oldest | Newest | Top

rachitnigam on Feb 23, 2022 Maintainer Author

Curious about thoughts from CIRCT folks: **@mikeurbach** **@stephenneuendorffer** **@cgyurgyik**

↑ 1 ☺ 0 replies

Write a reply

sampsyo on Feb 23, 2022 Maintainer

Just to summarize, some broad points here are:

- We want constructs for synchronization (i.e., ways to enforce ordering between events in different `par` arms, aka “threads” by analogy to software languages).
- We want them to be explicit so that programs can’t implicitly assume the freedom to communicate willy-nilly across `par` arms---for the same reasons we argued against “Lockstep” semantics in On the Semantics of `par` #921.
- We want synchronization to work in both latency-insensitive and static-timing contexts. In a latency-insensitive context, synchronization needs to be implemented with actual hardware (e.g., a handshake). In a static-timing context, synchronization needs to be “free” in the sense that it just depends on proving that events on the two `par` arms take the right number of cycles to enforce the ordering of events.

So, `@sync(n)` acts as the latency-insensitive version of the explicit synchronization construct we want. We hope to build the static-timing version on the same basis.

You could bikeshed about the use of numbers as names for barriers (maybe they should be stringy symbols or declared in `cells` or whatever) or the existence in annotations (maybe they should be leaf control statements instead). But something like this is the way.

↑ 1 ☺ 0 replies

Write a reply

 **rachitnigam** on Feb 23, 2022 Maintainer Author

@EclecticGriffin just curious–how hard would it be to implement the `sync` semantics in the interpreter? I don't know all the internal details so not sure if already having lockstep semantics would make it easier? The evaluator would need some mechanism to "stall" the execution of groups that have reached a particular `sync` boundary already.

↑ 1 ☺ 0 replies

Write a reply

Category

❓ **Semantics**

Labels

Summer 22

2 participants