On the Semantics of `par` #921

sampsyo started this conversation in **Semantics**

sampsyo on Feb 17, 2022 Maintainer edited

This is just to note down some highlights from a broader discussion we have often but haven't really nailed down (because it's hard!). What should `par` really mean?

I think we have three possible answers to this question. One is what we currently do but it's unsatisfying. The other two are potential ways to make it more satisfying.

Semantics 1: "DRF0"

The semantics we have currently embraced implicitly, at multiple points in the compiler and also in the interpreter, is this "least-common denominator" semantics. In this world, the arms of a `par` statement may never communicate. Any execution with a data race---defined as two accesses to the same data item from different par arms, at least one of which is a write---has undefined behavior. This is the same idea as the C memory model, or the basic "DRF0" memory model from the literature.

(Using "data item" above is kind of weasely. Here, I mean *both* Calyx ports *and* state values internal to components, e.g., registers' values.)

Because this semantics is so permissive, it permits everything our current implementation does:

- The compiler chooses to resolve this undefinedness by letting `par` arms communicate willy-nilly and never stopping them. But there are no guarantees about exactly when one value from one `par` arm becomes visible to other arms.
- The interpreter chooses to raise an error on some cases that would have undefined behavior: namely, conflicting parallel writes to the same port. But it does not signal an error for racy accesses to register values.
- Moreover, the interpreter in `--allow-par-conflicts` mode bypasses these checks and also allows willy-nilly communication, similarly without guarantees.

Under this semantics, #919 is not a bug: the interpreter and compiler do different things for racy programs like systolic arrays, but they are allowed to do so because all races are undefined.

Also on the positive side, every program has sequential semantics. In other words, our current "naive" `par-to-seq` pass, which just runs every arm of a `par` serially, is *sound* because it preserves the semantics of programs without data races. The fact that it breaks the systolic array programs is consistent because those programs are invalid under this semantics.

Of course, this semantics is also massively unsatisfying! We really, really want to write programs where `par` arms can communicate! That's why we have a problem.

Semantics 2: "Lockstep"

This is the first of two *proposed* semantics for `par`: it does not match what we currently do in our implementation, but we could consider adopting it and changing the implementation to match.

The idea is that `par` arms run in lockstep, cycle by cycle. So if one arm writes to a register on cycle 3, then it is visible in that register to all other arms on cycle 4.

In this semantics, it is *probably* the case that `par` arms can only communicate through state, not port values (wires). But I don't know for sure; that's something we'd need to narrow.

There are also some unknowns about when the cycle clocks "start" and when they "advance" through control statements. For example, if you have a control program like this:

```
par {
  seq { a; b };
  seq { x; y }
}
```

...and all the groups `a`, `b`, `x`, and `y` take the same number of cycles, there are some relevant questions about the relative timing of their execution. Do we know that `a` and `x` start and finish executing at the same time? If so, do we know that `b` and `y` start at the same time as each other? Do we know that `b` starts exactly one cycle after `a` ends, or is some slop allowed? All of these questions are relevant because their answers dictate what the groups can "see" when they read values written by groups in other arms.

This semantics (for some set of answers to the above questions), like DRF0 semantics, describes the current implementation of the compiler. The TDCC pass resolves those questions about relative timing and the gaps between FSM states.

Under this semantics, unlike under DRF0 semantics, #919 *is* a bug because the interpreter and compiler answer the above cycle-level timing questions differently. And while I may not know what those answers are, they need to have a consistent answer—and if two implementations disagree about their answers, then one of them has a bug.

Semantics 3: "Explicit Synchronization"

This is an alternative to "Lockstep" semantics that takes the polar opposite approach. Like "Lockstep," this semantics is a refinement of "DRF0" in the sense that every program that is valid under DRF0 is also valid under Explicit Synchronization and has the same semantics—but *more* programs *also* have defined semantics.

Data races, as defined above, are still disallowed. However, we introduce some manner of new construct—not currently present in Calyx—that *does* permit communication between `par` arms in a defined way. The key to this semantics is deciding what those special, blessed synchronization-and-communication elements are. Options include:

- "Synchronizing registers," which **@rachitnigam** has been exploring in the context of verification. We'd build a register augmented with a full-empty bit that works like an M-structure, for example, and this would be the only way for `par` arms to communicate.
- We could bake in FIFO queues or something that would allow unbounded, latency-insensitive communication between hardware elements.

The key insight to this semantics, however, is independent of what we tack onto the language: there just has to be *something* that, unlike everything else in the language, *does* allow cross-arm communication and synchronization. Existing stuff, including ports and `std_reg` and all that, would maintain its current "DRF0" semantics and would not support parallel communication of any sort.

Needless to say, "Explicit Synchronization" is currently unimplemented by any part of our project. The systolic array generator would need to be changed to use whatever fundamental communication construct we introduce; unlike under "Lockstep," it would remain just as invalid as under "DRF0" semantics.

Discussion & My Bias

I hope it is clear by now that our "DRF0" semantics, while necessary to describe the status quo, is not what we want in the long term. We need to choose and solidify one of the two alternatives above—or some other alternative we have not thought of yet (not sure what that is, though! I think our bases are pretty much covered here).

To make a choice, I believe the salient difference is that the two options are very different in terms of their restrictiveness: what kinds of implementations they permit. "Lockstep" semantics is restrictive because it relies on the global clock to say what programs mean: the arms of a `par` block must start at the same time; the steps in a `seq` must have a defined latency between them; etc. The compiler passes and interpreter have to obey this exact timing behavior to be correct. The "Explicit Synchronization" semantics is permissive by comparison: the different arms of a `par` block can run at different "speeds," for example, and FSM-generating compiler passes can put as much latency as they want between state transitions.

The most salient way this has come up in our compiler work so far has been the bugaboo surrounding "early transitions" in FSMs, discussed in #662 and #781. The essence there is that the current (as of this writing) implementation of the TDCC pass inserts a "dead zone" cycle between FSM states. It is possible to avoid this, but it's complicated. So we have a situation where TDCC can either be "fast and hard" or "slow and easy." The problem is that "Lockstep" semantics, which we have implicitly been embracing from time to time, does not allow the choice: because that semantics cares so deeply about individual clock cycles, it seems to dictate exactly how TDCC must behave when it generates FSMs. "Explicit Synchronization" would allow both implementations: we could conceivably have a "dumb and slow" control-compiler pass and a "fast and complex" control-compiler pass coexist right next to each other, and they would both be valid implementations of Calyx semantics.

I confess a bias toward "Explicit Synchronization" over "Lockstep." Here is where reasonable people can disagree, but hear me out. The first reason is that it admits more flexibility in the compiler implementation, as exemplified by the aforementioned "early transitions" debacle. The second reason is more philosophical and harder to explain.

This second reason is that it makes it simpler to reason about what Calyx programs do. Under "Lockstep" semantics, the arms of a `par` might communicate *at any point*. *Everything* they do might be affected by a different arm. This interference makes it hard to reason locally about what any individual arm does. In contrast, "Explicit Synchronization" semantics make `par` arms "local by default." Most things in a given `par` arm only affect (and are affected by) other things within the same `par` arm. Interference from other arms is restricted to synchronizing registers and the like. And if you want to reason about those, you never have to count cycles—you just have to understand what writes to synchronizing registers happen-before those synchronizing registers.

↑ 2 ☺

2 comments · 2 replies **Oldest** Newest | Top

rachitnigam on Feb 17, 2022 Maintainer

Thanks for the write up **@sampsyo**! I have a bunch of thoughts on this but I'll summarize them by saying this: The lack of a clock and an explicit lack of timing behavior in Calyx is a good thing–it is precisely the freedom from clock that enables modular reasoning, optimization, etc. However, it is also true that Calyx will forever be doomed to run into these problems as long as there is no mechanism to observe a clock *because* Calyx needs to interact with RTL.

For example, consider an RTL implementation of a time-sensitive protocol: "P sends a message on cycle 0 and expects a response at cycle 3". There is no way for *any* Calyx program that interacts with this module to be proven correct since there is no notion of time that can be observed from within Calyx.

Oliver Richardson pointed out an even more general problem: "there is no way to build common knowledge without a global clock (or some other notion of preexisting common knowledge)". The example above is a specialized case of the two general's problem; DRF0 Calyx programs are forever doomed to be incorrect.

Adrian lays out two options, "lockstep" and "explicit sync". I'll also state that lockstep doesn't seem like the right approach. It simultaneously has too much information, by specific explicit lockstep execution of assignments, and not enough information, specifically, which groups execute when. Let me offer my own two solutions.

A `static` Fragment of Calyx

In this world, `static` becomes a keyword modifies both control operators and groups. So if you want a `par` block where groups start and execute at the same time, you better write:

```
static par {
  static group1;
  static group2;
}
```

It is the job of the Calyx compiler to make sure that things are scheduled correctly with respect to the timing behavior. However, the compiler is *still free* to optimize the structure in any way it wants.
Additionally, the compiler knows exactly when things run so optimizations can make sure not to change that behavior.

It is possible that this is equivalent to lockstep; I haven't had a chance to think about this deeply.

A Primitive to Observe Time

Another (possibly complimentary) approach is having an explicit "clock generator" primitive in Calyx which is distinguished and the canonical way to observe time. Because Calyx registers *do not* provide any timing guarantees, it is *not correct* to use them to count clock cycles.

Furthermore, the only timing trick you're allowed to play are within a group and everything else takes DRF0 semantics. I've yet to flesh out this idea more but basically, I was thinking there is some distinguished syntax to observe time:

```
group foo { // %clock == 0 on group start
  c.in = %clock + 1 ? 1'd1;
  c.in = %clock + 2 ? 1'd0;
}
```

There may need to be some other syntax to observe time across groups. The benefit of this syntax is that timing tricks are represented in the IL and are therefore optimizable (famous compilers adage: "what you cannot represent, you cannot optimize")

(As I write this, I realize this might only be address one under wart of Calyx semantics which is that timing cannot be observed by any standard library primitives).

↑ 2 

1 reply

sampsyo on Feb 18, 2022 Maintainer Author

Interesting! I like these options as kind of a "mirror universe" alternative to explicit happens-before-style synchronization. Like "Explicit Synchronization" semantics above, both of these have the critical property of having an *opt-in* policy for cross-`par`-arm visibility. "Normal" stuff, including current programs that don't use the `static` keyword or the `%clock` value, would essentially retain "DRF0" semantics, and to the extent that arms want to communicate, they would have to adopt one of these things. (In the same way that, under "Explicit Synchronization," they would have to adopt synchronizing registers or whatever.)

In general, the underlying notion would be this: one arm can force some event to happen *before* clock cycle N, and a different arm can force some event to happen *after* clock cycle N. If both do this special "forcing" dance, then they can communicate.

You can imagine having *both* this (for statically timed communication) *and* synchronizing registers (for dynamically timed communication). If we want them both.

Write a reply

rachitnigam on Feb 22, 2022 Maintainer

Another gnarly program that uses timing-sensitivity in the implementation:

```
import "primitives/core.futil";
import "primitives/binary_operators.futil";
component main(go: 1, @go go0: 1, @clk clk: 1, @reset reset: 1) -> (done: 1,
@done done0: 1) {
  cells {
    @external mem = std_mem_d2(32, 2, 2, 32, 32);
    @external mem2 = std_mem_d2(32, 2, 2, 32, 32);
    @external result = std_mem_d2(32, 2, 2, 32, 32);
    counter2 = std_reg(32);
    val1 = std_reg(32);
```

```
      val2 = std_reg(32);
      result_reg = std_reg(32);
      mul_reg = std_reg(32);
      mul = std_mult_pipe(32);
      add = std_add(32);
      add3 = std_add(32);
      add4 = std_add(32);
    }
    wires {
      group init {
        counter2.in = 32'd0;
        counter2.write_en = 1'd1;
        init[done] = counter2.done;
      }
      group read {
        mem.addr0 = 32'd0;
        mem.addr1 = 32'd0;
        val1.in = mem.read_data;
        val1.write_en = 1'd1;
        read[done] = val1.done;
      }
      group store {
        result.addr0 = 32'd0;
        result.addr1 = counter2.out;
        result.write_en = 1'd1;
        result.write_data = val1.out;
        store[done] = result.done;
      }
      group incr {
        add3.left = counter2.out;
        add3.right = 32'd1;
        counter2.in = add3.out;
        counter2.write_en = 1'd1;
        incr[done] = counter2.done;
      }
    }

    control {
      seq {
        init;
        par {
          seq {
            read;
            store;
          }
          incr;
        }
      }
    }
  }
```

The key thing is:

```
par {
  seq {
    read;
    store;
  }
  incr;
}
```

`read` executes for a cycle while `incr` increments the value of `counter2` in parallel. `store` uses the new value of `counter2` in the next cycle. `par-to-seq` is hopelessly lost in the face of programs like this.

↑ 1

1 reply

sampsyo on Feb 23, 2022 Maintainer Author

Would this be correct under DRF0?

```
seq {
  par { incr; read };
  store;
}
```

Write a reply

Category

❓ **Semantics**

Labels

None yet

2 participants