The Essence of Bluespec

A Core Language for Rule-Based Hardware Design

Thomas Bourgeat
MIT CSAIL
Cambridge, Massachusetts, USA
bthom@csail.mit.edu

Clément Pit-Claudel
MIT CSAIL
Cambridge, Massachusetts, USA
cpitcla@csail.mit.edu

Adam Chlipala
MIT CSAIL
Cambridge, Massachusetts, USA
adamc@csail.mit.edu

Arvind
MIT CSAIL
Cambridge, Massachusetts, USA
arvind@csail.mit.edu

Abstract

The Bluespec hardware-description language presents a significantly higher-level view than hardware engineers are used to, exposing a simpler concurrency model that promotes formal proof, without compromising on performance of compiled circuits. Unfortunately, the cost model of Bluespec has been unclear, with performance details depending on a mix of user hints and opaque static analysis of potential concurrency conflicts within a design. In this paper we present Kôika, a derivative of Bluespec that preserves its desirable properties and yet gives direct control over the *scheduling* decisions that determine performance. Kôika has a novel and deterministic operational semantics that uses dynamic analysis to avoid concurrency anomalies. Our implementation includes Coq definitions of syntax, semantics, key metatheorems, and a verified compiler to circuits. We argue that most of the extra circuitry required for dynamic analysis can be eliminated by compile-time BSV-style static analysis.

CCS Concepts: • **Software and its engineering** → **Semantics**; **Compilers**; • **Hardware** → **Theorem proving and SAT solving**.

Keywords: HDL, Semantics, Compiler Correctness

ACM Reference Format:
Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385965

1 Introduction

A synchronous digital circuit is a state-transition system that specifies how the present state, held in registers, is transformed into the next state at every clock cycle. Popular hardware-description languages like Verilog expose this view fairly directly. However, for design purposes, it is not easy to think of the functionality of a complex digital circuit in terms of a global state-transition system.

Verilog does allow division of a design into separate concurrent blocks, each of which computes a subset of register updates every cycle. The natural concurrency of computing many state updates at once provides significant optimization opportunities, but, as in concurrent software, it introduces opportunities for bugs because of shared state. One well-developed alternative is guarded atomic actions, as implemented in the hardware-description language Bluespec SystemVerilog (BSV) [30]. In BSV, the design specifies all the state elements, i.e., registers, and describes the behavior using a set of atomic rules. Each rule specifies a deterministic state transformation. It is guaranteed that *rules appear to execute atomically*, one-at-a-time, much like the established software concept of transactions. However, literal one-rule-at-a-time (ORAAT) execution in a hardware circuit would bring unacceptably poor performance. We still do need rules to execute concurrently, though in a controlled way that preserves the illusion of atomic execution. The BSV compiler does static analysis to construct a per-design *scheduler* circuit automatically, which chooses among the set of (enabled) rules in each clock cycle.

To appreciate the considerations that go into choosing a schedule, it is important to start from the quantitative metrics that matter for circuits. The most commonly cited are power, performance, and area. We think of circuits as directed graphs whose nodes are registers and gates, where

PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385965

no cycles are permitted on paths that only traverse gates. To a first approximation, area and power follow from register and gate counts, which we want to keep down. A major determinant of performance is the clock-cycle time, which is proportional to the *critical-path length*, i.e., the length of the longest path between any registers, even from the same register to itself. Path length here refers to the propagation delay of all the gates on a path between two registers.

We might be tempted to aim for a free lunch by removing gates to shorten the cycle time, but then we have postponed work to happen on later cycles, not necessarily shortening the total compute time. For example, consider a decomposition of function f into $f_2 \circ f_1$ (section 5) to allow pipelining. Let us assume that a circuit implementing f in one go requires a single cycle of length 10 seconds to execute, while circuits for f_1 and f_2 require 4 and 6 seconds, respectively. The unpipelined system processes one token per 10 seconds (the time to run f), while a system that repeatedly runs one of f_1 or f_2 has cycle time equal to the *maximum* of those cycle times, 6. Thus, in the two cycles it takes to run through the full pipeline, we take up $2 \cdot 6 = 12$ seconds, and our "optimization" actually made things worse. However, if both stages f_1 and f_2 execute concurrently each cycle, i.e. in a pipelined manner, we can process one token every 6 seconds.

The challenge in describing designs of this kind is that we often want rules to execute concurrently *even when they access some of the same state elements*. In the previous example, pipeline stages f_1 and f_2 would need to share some kind of queue, which f_1 enqueues into and f_2 dequeues out of simultaneously each cycle, f_1 enqueuing in cycle n the data consumed by f_2 in cycle $n + 1$.

The commercial BSV compiler relies on a static analysis to do ORAAT-preserving concurrent scheduling of rules. Its static analysis, combined with user-provided annotations (e.g. descending urgency and execution order), generally creates excellent circuits. This approach, however, is not satisfactory for two reasons. (1) The static analysis should be an abstraction of the dynamic semantics of a program. BSV's dynamic semantics applies only to one-rule-at-a-time executions, and the cycle-level semantics necessarily depends on the static analysis of rules. (2) BSV programmers often think deliberately about static-analysis details and even change their code to nudge the compiler in the right direction to achieve the desired degree of concurrency. We take a different approach in this paper, providing a new core calculus Kôika maintaining the essence of BSV, preserving all its desirable properties and yet allowing direct control over the atomic actions executed each clock cycle, without relying on static analysis. Kôika programmers still need to think about the real rule conflicts but not about the compiler's abstraction thereof. Our calculus includes a *deterministic, cycle-accurate* operational semantics, enabling formal reasoning about performance, without removing the ability to prove invariants by induction on sequential executions: the effect of the set of rules completed each cycle is proven to be always explainable in terms of ORAAT semantics.

Often the rules we want to run concurrently require controlled communication amongst themselves. BSV's ephemeral history registers (EHRs) provide a mechanism to enhance concurrency in rule scheduling. EHRs essentially enrich rule-based designs with what are known as *bypasses* in hardware design. EHR semantics guarantees that the observed behaviors can be reproduced with serial execution of rules. However, pure ORAAT semantics are unable to capture the performance implications of EHRs because in ORAAT semantics, EHRs are indistinguishable from ordinary registers. Kôika's semantics, on the other hand, capture both the functional and performance aspects of EHRs.

The commercial BSV compiler can be viewed as producing one *schedule* (concurrency strategy) automatically, and our calculus supports most of these schedules and others that allow more concurrency for performance[1]. Though space limitations prevent us from delving into the concurrency payoffs from Kôika's more flexible scheduling, we present the semantics of Koika, its ORAAT property, its compilation into circuits, and the proof of the compiler's correctness.

This paper makes the following contributions:

1. Kôika, the first core calculus for a BSV-like rule-based language to support formal reasoning about both functional and performance properties;
2. A cycle-accurate operational semantics that does not depend upon any static analysis;
3. A key metatheorem that Kôika's operational semantics only produces executions that can be mimicked with one-rule-at-a-time execution;
4. A simple algorithm to compile Kôika programs into RTL circuits, preserving the concurrency of the operational semantics;
5. A mechanization of Kôika and its metatheory in the Coq proof assistant;
6. A formally verified compiler from Kôika to primitive circuits;
7. A case study of an embedded-class pipelined processor written in Kôika.

Paper organization: We start with an introduction to Kôika (section 2), next defining its formal semantics (section 3), which allows the execution of multiple rules in one cycle. We prove that the ORAAT property emerges from this semantics (section 4). Then we use Kôika's semantics to characterize the behavior of a pipeline (section 5). After that we present a compilation of Kôika to circuits (section 6) and discuss some efficiency concerns (subsection 6.4), followed by related work (section 7) and a brief conclusion.

The Kôika release accompanying this paper is available at https://github.com/mit-plv/koika/tree/pldi2020.

[1] BSV and Kôika schedules can be difficult to compare because BSV allows multiple rules to write into the same register but picks which write prevails.

2 Introduction to Kôika

We start with a gentle introduction to Kôika, our calculus of atomic actions, by presenting its features incrementally and explaining their meanings informally (formal semantics are presented in section 3). For readers familiar with BSV, this section will serve mostly as a refresher.

2.1 Rules or Atomic Actions

Programs are composed of *rules* (roughly: atomic units of execution) that manipulate values stored in *registers*. Taken together, these rules define what happens in each *clock cycle*.

The following rule increments the value in register r:

```
rule increment =
  let v = r.rd in r.wr(inc(v))
```

This rule first reads the value stored in register r into a variable v, then applies the combinational (i.e., pure) function inc to v, and finally writes the result in register r. All of these actions are performed as one atomic unit.

Reads and writes: Our next example showcases one way in which hardware languages differ from traditional software languages. Rule swap swaps the values of two registers r and s without using a temporary variable:

```
rule swap = s.wr(r.rd); r.wr(s.rd)
```

All read operations of registers, i.e., rds, refer to the values found in the registers at the *beginning* of the cycle (equivalently, these are the values committed to the registers at the end of the preceding cycle). Correspondingly, the effect of a write operation, i.e., wr, is not observable until the beginning of the next cycle; all register updates happen simultaneously at the end of each cycle.

Double writes: Delaying writes to the end of each cycle requires us to clarify the semantics of double writes, i.e., the case where write is called twice on the same register. We could have decided to let the later write overshadow the earlier one, but use of shadowing is typically considered an antipattern; so instead we *abort* the execution of the rule, meaning that the system behaves as if the rule did not execute.

```
rule aborts = r.wr(0b11); s.wr(0b00); s.wr(0b01)
```

These aborts are detected dynamically, during execution, and the whole rule is canceled atomically (in the example above, this implies that the write to r will not be performed either). In the following example, a double write happens conditionally when r and s both hold the value 0, in which case the rule aborts, though otherwise it succeeds.

```
rule conditional_abort =
  if r.rd == 0 then t.wr(0b1);
  if s.rd == 0 then t.wr(0b1)
```

Collatz function: We now know enough to look at a small realistic example, which we will reuse for illustrative purposes throughout the paper. The example below computes terms of the Collatz sequence, defined by the two equations $u_{n+1} = u_n/2$ if u_n is a multiple of two, and $u_{n+1} = 3 \cdot u_n + 1$ otherwise.

```
rule divide =
  let v = r.rd in
  if iseven(v) then
    r.wr(v >> 1)
```

```
rule multiply =
  let v = r.rd in
  if isodd(v) then
    r.wr(3 * v + 1)
```

Up to now we have focused on the semantics of rules in isolation, but this example has two rules. The so-called one-rule-at-a-time (ORAAT) semantics of a collection of rules is to pick a rule nondeterministically, execute it, and commit its results. (In case of an abort, the state does not change.) The process is repeated endlessly, as if exactly one rule executed in each clock cycle: if one rule writes to a register, the next rule observes the newly written value. The ORAAT semantics need not produce a deterministic answer because the rules are not required to be confluent.

2.2 Scheduling

ORAAT is a conceptual model. In designing efficient hardware, however, we strive to execute as many compatible rules as possible in parallel in each clock cycle, without violating the illusion of running rules one-at-a-time. In order to introduce concurrency, we define a *schedule*, which specifies the order in which we expect rule effects to become observable.

It is straightforward to see that rules operating on disjoint register sets can be run in parallel without affecting the final outcome. Regardless of scheduling order, their effects commute. In the following example, however, opportunities for parallel execution depend on scheduling choices:

```
rule write_r = r.wr(0b10)
rule read_r = s.wr(inc(r.rd))
```

Both of these rules access register r, and they may be sequenced in two ways: attempt to run write_r then read_r, or attempt to run read_r then write_r.

If we start with write_r then, according to ORAAT, read_r must observe the new value of r; hence, read_r cannot happen within the same cycle. If we start with read_r, on the other hand, it is safe to run both rules in the same cycle: the effect will be just the same as if we had executed read_r, waited until the next cycle, and executed write_r. In that case, we say that the two rules "fired" (ran) *concurrently*, or *simultaneously*.

For this program, a scheduler that runs read_r before write_r allows parallelism. A semantic characterization of the system restricted to ORAAT would not specify how these two rules should be sequenced, and it would therefore be insufficient because such distinctions are crucial in hardware design. Accordingly, unlike plain ORAAT semantics, our definition of a program includes a *scheduler specification*, which describes unambiguously the order in which rules will appear to have run in each cycle. With this specification, each program describes a *unique* sequential machine up to Boolean equivalence, and it becomes possible to reason cycle-accurately about performance.

Here is how we write the two schedules above:

```
schedule blocked = [write_r; read_r]
schedule parallel = [read_r; write_r]
```

The key point of making schedules explicit is to allow fine-grained control over concurrency, by enabling several rules to execute in one clock cycle as long as their effects are compatible with the linear order specified by the scheduler. Our semantics ensure that concurrently executed rules produce results compatible with ORAAT semantics. A rule whose execution would cause a violation is aborted dynamically.

It is important to realize that a schedule specifies which rules execute within one cycle. It says nothing about inter-cycle scheduling; the same schedule is used every cycle. For example in the *blocked* schedule, `read_r` will never actually be performed because it will be preempted by `write_r` each cycle.

2.3 Ephemeral History Registers (EHRs)

The language that we have outlined up to this point respects ORAAT but is overly restrictive. Indeed, without adding extra constructs in the language, there is no way to have data flowing between rules within a single cycle: rules are fully isolated from each other.

To relax this restriction while preserving ORAAT, we introduce two new operations on registers, rd$_1$ and wr$_1$. (From now on we treat rd and wr as synonyms for rd$_0$ and wr$_0$, respectively.) These new primitives allow programmers to control data forwarding between rules: data written by wr$_0$ in a register is readable by rd$_1$ on the same register, and data written by wr$_1$ becomes readable by rd$_0$ in the next cycle[2]. This mechanism coming from BSV is associated with the unwieldy name *ephemeral history register (EHR)* [32, 33].

```
rule inc_r =                    rule check_r =
  let v = r.rd0 in                let v = r.rd1 in
  if v < 0b101 then               if even(v) then
    r.wr0(inc(v))                   s.wr0(v)
schedule fwd = [inc_r; check_r]
```

The scheduler in this program specifies that `inc_r` should execute first. Thus, if `check_r` attempted to read register `r` by rd$_0$, it would abort; but it may read `r` using rd$_1$. Doing so, it observes the value written by `inc_r`, if any, or the initial value of `r` otherwise. We say that the write to `r` was *forwarded* to `check_r`. Similarly, `check_r` would abort if it attempted a wr$_0$ into `r`, but a wr$_1$ would succeed and take precedence over any value previously written by wr$_0$.

rd$_0$ and wr$_0$ can be performed in any order within a rule. However, if a wr$_0$ is performed by a rule then, to preserve ORAAT semantics, no later rule can perform a rd$_0$ to the same register in the same cycle. More generally, Kôika places dynamic restrictions on these new operations. No reads or writes can follow a wr$_1$ in a subsequent rule (as an example, a rd$_0$ following a wr$_1$ would observe a stale value if it ran the same cycle as a preceding wr$_1$), and a wr$_0$ cannot follow a rd$_1$ in the same rule or across rules. We also add the restriction that wr$_0$ cannot follow wr$_1$ in the same rule. This restriction is not directly required but simplifies the semantics, and we cannot think of an interesting program that would benefit from relaxing the restriction. Roughly, we have the following restrictions across rules: rd$_0$ < wr$_0$, wr$_1$; rd$_1$ < wr$_1$; wr$_0$ < rd$_1$ — but note that, in the absence of a wr$_0$, rd$_0$ and rd$_1$ may be interleaved freely. We will formalize these restrictions in section 3. Finally, notice that the rds are *per-register*: it is completely valid, and in fact often useful (particularly in building pipelines), to use wr$_0$ to store in a given register the result of a computation involving a rd$_1$ of another register. In other words, the numbers should *not* be read as timestamps describing a global order within the clock cycle.

Allowing for data forwarding between rules increases flexibility and enables additional concurrency, but it potentially lengthens the critical path of the generated circuit. Careful designers typically use forwarding sparingly, when it unlocks additional parallelism without increasing the critical path in a destructive way.

The Collatz example revisited: After revealing additional primitives beyond just rd and wr, we can revisit our Collatz example. It can be written as follows, using EHRs:

```
rule divide =                   rule multiply =
  let v = r.rd0 in                let v = r.rd1 in
  if iseven(v) then               if isodd(v) then
    r.wr0(v >> 1)                   r.wr1(3 * v + 1)
schedule collatz = [divide; multiply]
```

Note that `multiply` performs a rd$_1$, allowing both rules to run in the same cycle in certain cases. More precisely, the circuit behaves in the following way:

- If the value in `r` is even but not a multiple of 4, both rules fire: the circuit writes $3 \cdot (\mathtt{r}/2) + 1$ in `r`.
- If the value in `r` is a multiple of 4, only the first rule fires: the circuit writes $\mathtt{r}/2$ in `r`.
- If the value in `r` is odd, only the second rule fires: the circuit writes $3 \cdot \mathtt{r} + 1$ in `r`.

[2] It is natural to consider generalizing this mechanism to a register with arbitrarily many "ports" (0, 1, 2, ...), but it turns out that the two-port version is expressive enough to encode any number of ports, so we restrict our attention to this simpler case.

This example shows that the concurrent execution of rules can be enhanced substantially by using EHRs.

3 Formal Description of Kôika

Combinational functions (pure mathematical functions that do not read or write registers) play no role in our semantics. Therefore, we avoid describing them by assuming a set of named combinational functions.

3.1 Syntax

A program is described by a set of rules and a scheduler:

$$\begin{array}{lcll} \text{Program} & P & ::= & [\mathtt{rule}\ \mathit{rule\_name} = a]^* \\ & & & \mathtt{schedule}\ \mathit{schedule\_name} = s \end{array}$$

$$\begin{array}{lcl} \text{Schedule} \quad s & ::= & \mathtt{done} \mid \mathtt{cons}\ \mathit{rule\_name}\ s \end{array}$$

As an abbreviation for `cons r1 (cons r2 ...)` we write `[r1, r2, ...]`, which represents the sequencing of rules.

Each *rule_name* in a schedule refers to a rule, which is an action that returns tt (the unit value).

$$\begin{array}{rcll} \text{Actions} & a & ::= & b \mid x \mid \mathtt{skip} \mid r.\mathtt{rd}_p \mid r.\mathtt{wr}_p(a) \\ & & & \mid \mathtt{let}\ x = a\ \mathtt{in}\ a \mid f(a, \ldots, a) \\ & & & \mid \mathtt{if}\ a\ \mathtt{then}\ a\ \mathtt{else}\ a \mid \mathtt{abort} \\ \text{Ports} & p & ::= & 0 \mid 1 \\ \text{Bitstrings} & b & ::= & \mathrm{tt} \mid \mathtt{0b(0|1)+} \\ \text{Registers} & r & & \text{Variables}\ x \qquad \text{Externals}\ f \end{array}$$

`skip` is the unit value for actions, standing for no action, which returns tt when executed. As a shorthand, we write a_1 `;` a_2 for `let` x `=` a_1 `in` a_2 with unused x. Similarly, we write `if` b `then` a for `if` b `then` a `else skip`, as we have already used in the introduction to the language. We will consider only well-formed programs in this paper. For example, a rule that attempts to write a 12-bit value into a one-bit register, or calls an external combinational function with arguments of inappropriate bit-widths or inappropriate number of arguments, or refers to a nonexistent register, etc., will not be considered. (Our implementation applies a very standard type system to rule out these failures.)

3.2 Semantics

A rule in our language is an action that returns tt. It is characterized by the log ℓ of reads and writes it performs. The semantics of executing a single rule in isolation can be thought of as defining a function from the register values (notated $\mathcal{R}$) to generate a log ℓ. This log ℓ is built inductively along with the local environment of binders Γ. The effect of executing this rule in isolation would be to use the generated log to update the registers: $\mathcal{R}_{\text{next\_cycle}} = \mathsf{update}\,(\mathcal{R}, \ell)$, with:

$$\begin{cases} \mathsf{update}\,(\mathcal{R}, []) = \mathcal{R} \\ \mathsf{update}\,(\mathcal{R}, \ell +\!\!+ [(\mathsf{rd}_*, r)]) = \mathsf{update}\,(\mathcal{R}, \ell) \\ \mathsf{update}\,(\mathcal{R}, \ell +\!\!+ [(\mathsf{wr}_*, r, v)]) = \mathsf{update}\,(\mathcal{R}, \ell)[r \mapsto v] \end{cases}$$

Indeed there are at most two writes per register in the log, `wr`$_0$ and `wr`$_1$, and we will come shortly to how it is guaranteed that `wr`$_0$ never precedes `wr`$_1$ for any register.

We want to give the semantics of executing multiple rules, as specified by the schedule, every clock cycle. Under such circumstances, a rule can see the side effects of rules scheduled earlier. Therefore, we accumulate the effects of all preceding rules in a global log L. Thus the semantics of a rule whose body is a are as follows:

$$[\![a]\!](\mathcal{R}, L) = \begin{cases} \mathsf{Log}\ \ell & \text{if } a \text{ succeeds and produces the log } \ell \\ \mathsf{Fail} & \text{if } a \text{ fails} \end{cases}$$

which we can use to define the effect of executing multiple rules according to a scheduler.

$$\frac{}{(L, \mathtt{done}) \Downarrow L}\textsc{Done}$$

$$\frac{[\![a]\!](\mathcal{R}, L) = \mathsf{Log}\ \ell \qquad (L +\!\!+ \ell, s_{\text{next}}) \Downarrow L'}{(L, \mathtt{cons}\ a\ s_{\text{next}}) \Downarrow L'}\textsc{SeqLog}$$

$$\frac{[\![a]\!](\mathcal{R}, L) = \mathsf{Fail} \qquad (L, s_{\text{next}}) \Downarrow L'}{(L, \mathtt{cons}\ a\ s_{\text{next}}) \Downarrow L'}\textsc{SeqFail}$$

Figure 1. Scheduler semantics, with `rule` $rl = a$

Note that there is no difference between a rule that fails and an empty rule.

Semantics of actions. Now we can describe in detail the way log generation by rules is defined inductively:

- $\mathcal{R}$ records the state of all available registers at the beginning of a clock cycle. Hence $\mathcal{R}$ remains invariant throughout the execution of a rule, and in fact throughout the execution of all the rules in a schedule.
- Γ tracks pairs of names and values created by `let` constructs. It starts out empty.
- ℓ accumulates the reads and writes of the rule.
- L accumulates a trace of all the reads and writes performed by rules executed earlier in the same clock cycle (i.e., as part of the same schedule). L remains invariant through the execution of the rule but affects the validity of reads and writes by this rule.

The semantics of actions are defined by structural induction in Figure 2. We write $\Gamma \vdash (\ell, a) \downarrow_{(L,\mathcal{R})} (\ell', v)$, to indicate that *in environment* Γ*, with log* L *and registers* $\mathcal{R}$*, executing an action* a *transforms* ℓ *into* ℓ' *and returns value* v. When there is no ambiguity, we omit L and $\mathcal{R}$ and write $\Gamma \vdash (\ell, a) \downarrow (\ell', v)$ instead.

Careful inspection of our semantic judgments reveals that all premises are deterministic and computable. That means we can define a computable evaluation function unambiguously, returning either the result of executing the action or Fail if at any point in the execution the conditions of the

$$\frac{\Gamma[x] = v}{\Gamma \vdash (\ell, x) \downarrow (\ell, v)}\textsc{Var} \qquad \frac{}{\Gamma \vdash (\ell, \texttt{skip}) \downarrow (\ell, \mathrm{tt})}\textsc{Skip}$$

$$\frac{}{\Gamma \vdash (\ell, b) \downarrow (\ell, b)}\textsc{Const}$$

$$\frac{\forall\, 1 \le i \le n.\, \Gamma \vdash (\ell_{i-1}, a_i) \downarrow (\ell_i, v_i)}{\Gamma \vdash (\ell_0, f(a_1, \ldots, a_n)) \downarrow (\ell_n, f\ v_1 \ldots v_n)}\textsc{Call}$$

$$\frac{\Gamma \vdash (\ell, a_c) \downarrow (\ell', \texttt{0b1}) \qquad \Gamma \vdash (\ell', a_t) \downarrow (\ell'', v)}{\Gamma \vdash (\ell, \texttt{if}\ a_c\ \texttt{then}\ a_t\ \texttt{else}\ a_f) \downarrow (\ell'', v)}\textsc{IfT}$$

$$\frac{\Gamma \vdash (\ell, a_c) \downarrow (\ell', \texttt{0b0}) \qquad \Gamma \vdash (\ell', a_f) \downarrow (\ell'', v)}{\Gamma \vdash (\ell, \texttt{if}\ a_c\ \texttt{then}\ a_t\ \texttt{else}\ a_f) \downarrow (\ell'', v)}\textsc{IfF}$$

$$\frac{\Gamma \vdash (\ell, a_1) \downarrow (\ell', v) \qquad \Gamma[x \mapsto v] \vdash (\ell', a_2) \downarrow (\ell'', v')}{\Gamma \vdash (\ell, \texttt{let}\ x \texttt{=} a_1\ \texttt{in}\ a_2) \downarrow (\ell'', v')}\textsc{Bind}$$

$$\frac{(\mathrm{wr}_1, r, *) \notin L \qquad (\mathrm{wr}_0, r, *) \notin L}{\Gamma \vdash (\ell, r.\texttt{rd}_\texttt{0}) \downarrow (\ell +\!\!+ [(\mathrm{rd}_0, r)], \mathcal{R}[r])}\textsc{Read0}$$

$$\frac{(\mathrm{wr}_1, r, *) \notin L \qquad v = \begin{cases} \mathcal{R}[r] & \text{if } (\mathrm{wr}_0, r, *) \notin L +\!\!+ \ell \\ v_0 & \text{if } (\mathrm{wr}_0, r, v_0) \in L +\!\!+ \ell \end{cases}}{\Gamma \vdash (\ell, r.\texttt{rd}_\texttt{1}) \downarrow (\ell +\!\!+ [(\mathrm{rd}_1, r)], v)}\textsc{Read1}$$

$$\frac{\begin{matrix}\Gamma \vdash (\ell, a) \downarrow (\ell', v) \qquad (\mathrm{wr}_0, r, *) \notin L +\!\!+ \ell' \\ (\mathrm{wr}_1, r, *) \notin L +\!\!+ \ell' \qquad (\mathrm{rd}_1, r) \notin L +\!\!+ \ell'\end{matrix}}{\Gamma \vdash (\ell, r.\texttt{wr}_\texttt{0}(a)) \downarrow (\ell' +\!\!+ [(\mathrm{wr}_0, r, v)], \mathrm{tt})}\textsc{Write0}$$

$$\frac{\Gamma \vdash (\ell, a) \downarrow (\ell', v) \qquad (\mathrm{wr}_1, r, *) \notin L +\!\!+ \ell'}{\Gamma \vdash (\ell, r.\texttt{wr}_\texttt{1}(a)) \downarrow (\ell' +\!\!+ [(\mathrm{wr}_1, r, v)], \mathrm{tt})}\textsc{Write1}$$

Figure 2. Rule semantics (assuming well-formed programs)

relevant rules are not met and the rule execution cannot proceed. Thus, where we previously wrote "$[\![a]\!](\mathcal{R}, L) = \mathrm{Log}\ \ell$ if a succeeds and produces the log ℓ," we can now be precise:

$$[\![a]\!](\mathcal{R}, L) = \begin{cases} \mathrm{Log}\ \ell & \text{if } \emptyset \vdash ([], a) \downarrow_{(L,\mathcal{R})} (\ell, \mathrm{tt}) \\ \mathrm{Fail} & \text{otherwise} \end{cases}$$

Finally, at the end of each cycle, we update the values of all registers based on the reads and writes accumulated in log L. The same considerations of determinism and computability apply to the execution of schedulers, so we can define our final state-transition function δ_s, capturing all updates done to registers in a cycle when following scheduler s:

$$\delta_s(\mathcal{R}) = \mathrm{update}\,(\mathcal{R}, L) \ \text{ if } ([], s) \Downarrow L.$$

Coq formalization Our mechanization matches this exposition quite closely, with the difference that the logs are represented in the opposite order. Programs are written as deeply embedded, dependently typed ASTs (see `action` in `TypedSyntax.v`), though for convenience we also have an untyped layer (`UntypedSyntax.v`), a typechecker (`TypeInference.v`), and a grammar that closely match the syntax used in this paper (`Parsing.v`). The semantics are formulated in denotational style, as recursive functions returning either Log(...) or Fail (see `interp_rule`, `interp_scheduler`, and `commit_update` in `Semantics.v`, which correspond to what we here call $[\![a]\!](\mathcal{R}, L)$ and update $(\mathcal{R}, L)$).

4 The One-Rule-at-a-Time Theorem

Our semantics builds a log accumulating the updates performed by all rules that the dynamic scheduler allows to run. It guarantees that performing a single update of the registers at the end of the cycle, after running multiple rules, yields the same state as performing updates after running each rule (as if a single rule had run in each cycle).

Let us illustrate that statement with the following example:

```
rule incr = x.wr0(x.rd0 + 1)
rule copy = y.wr0(x.rd1)
rule decr = x.wr1(x.rd1 - 1)
schedule _ = [incr; copy; decr]
```

The choice of scheduler and the port annotations (using `rd1` in `copy` and `wr1` in `decr`) ensure that all rules can run in each cycle. Overall, this program assigns $(x + 1) - 1$ to register `x` and $x + 1$ to register `y`. This result (obtained by running multiple rules in a single cycle) respects one-rule-at-a-time semantics, because the same result can be obtained by running one rule per cycle, in the order specified by the scheduler: $x \longleftarrow x + 1$, then $y \longleftarrow x$, and finally $x \longleftarrow x - 1$.

Most of the checks that appear in the premises of Figure 2 are there to preserve ORAAT semantics. For example, allowing a `rd0` to follow a `wr0` performed by an earlier rule in the same cycle would not respect ORAAT: the `rd0` would observe the old value of the register if it ran in the same cycle, vs. the new value if it ran in the next cycle.

We define the action of a single rule on the registers by applying the semantics of actions directly:

$$\frac{[\![a]\!](\mathcal{R}, \emptyset) = \mathrm{Log}\ \ell \qquad \mathcal{R}' = \mathrm{update}\,(\mathcal{R}, \ell)}{\mathcal{R} \to_a \mathcal{R}'}\textsc{rule}$$

And we define a relation indicating that a state is reachable in several rules: $\mathcal{R} \to^*_{[]} \mathcal{R}$ and $\mathcal{R} \to^*_{\mathrm{h::t}} \mathcal{R}''$ when $\exists \mathcal{R}'.\ \mathcal{R} \to_h \mathcal{R}' \wedge \mathcal{R}' \to^*_{\mathrm{t}} \mathcal{R}''$.

We can now give the proper statement of a key property:

Theorem 1. *If $\delta_s(\mathcal{R}) = \mathcal{R}_1$ then there exists a sequence of rules of the program that one-at-a-time reach the same state: $\exists\ rls \in traces(s).\ \mathcal{R} \to^*_{rls} \mathcal{R}_1$, where* traces($s$) *refers to all sequences of rules named in the scheduler s.*

This theorem can be found as `OneRuleAtATime` in the file `OneRuleAtATime.v`. In this paper, we detail only the following key lemma.

Lemma 1. $\forall a, \Gamma, \ell, L_{new}, L_{old}, \ell', v, \mathcal{R}.$
$\Gamma \vdash (\ell, a) \downarrow_{L_{old} +\!\!+ L_{new}, \mathcal{R}} (\ell', v) \Rightarrow$
$\Gamma \vdash (\ell, a) \downarrow_{L_{new}, update(\mathcal{R}, L_{old})} (\ell', v)$

Proof. This lemma corresponds to `interp_rule_commit` in `OneRuleAtATime.v`. The proof works by induction on a. We outline the most interesting cases here. One key invariant is that there is at most one $\texttt{wr}_0$ and $\texttt{wr}_1$ per register, as the semantics prevent double $\texttt{wr}_0$ or $\texttt{wr}_1$.

$r.\mathbf{rd}_1$.. Assume $\Gamma \vdash (\ell, r.\texttt{rd}_1) \downarrow_{L_{\text{old}} +\!\!+ L_{\text{new}}, \mathcal{R}} (\ell', v)$. There are two cases depending on where the $\texttt{rd}_1$ read the value from.
Case 1: *Read from register.* Necessarily we have $\ell' = \ell +\!\!+ [(\text{rd}_1, r)]$, $\mathcal{R}[r] = v$, $(\text{wr}_1, r, *) \notin L_{\text{old}} +\!\!+ L_{\text{new}}$ and $(\text{wr}_0, r, *) \notin \ell +\!\!+ L_{\text{old}} +\!\!+ L_{\text{new}}$, which implies that $(\text{wr}_1, r, *) \notin L_{\text{new}}$ and $(\text{wr}_0, r, *) \notin L_{\text{new}} +\!\!+ \ell$. We also get $(\text{wr}_0, r, *) \notin L_{\text{old}}$ and $(\text{wr}_1, r, *) \notin L_{\text{old}}$, so:

$$\text{update}\,(\mathcal{R}, L_{\text{old}})[r] = \mathcal{R}[r]$$

Hence we can use the Read1 rule reading from the register, with $L ::= L_{\text{new}}$ and $\mathcal{R} ::= \text{update}\,(\mathcal{R}, L_{\text{old}})$ and so $\Gamma \vdash (\ell, r.\texttt{rd}_1) \downarrow_{L_{\text{new}}, \text{update}(\mathcal{R}, L_{\text{old}})} (\ell', v)$.
Case 2: *Read from log.* Necessarily we have $\ell' = \ell +\!\!+ [(\text{rd}_1, r)]$, $(\text{wr}_0, r, v) \in L_{\text{old}} +\!\!+ L_{\text{new}} +\!\!+ \ell$, and $(\text{wr}_1, r, *) \notin L_{\text{old}} +\!\!+ L_{\text{new}}$.

There are three subcases depending on the source of the unique $\texttt{wr}_0$ we read from. The write is in ℓ, in L_{new}, or in L_{old}. The only interesting case is when the write is in L_{old}. We have $(\text{wr}_0, r, v) \in L_{\text{old}}$ and $(\text{wr}_1, r, *) \notin L_{\text{old}}$, so $\text{update}\,(\mathcal{R}, L_{\text{old}})[r] = v$. Moreover, $(\text{wr}_0, r, *) \notin L_{\text{new}} +\!\!+ \ell$ and $(\text{wr}_1, r, *) \notin L_{\text{new}}$.

So we have all the premises to apply Read1, reading directly from the register. This case is the most interesting because, in serializing a trace, we converted a log read into a register read. We get $\Gamma \vdash (\ell, r.\texttt{rd}_1) \downarrow_{L_{\text{new}}, \text{update}(\mathcal{R}, L_{\text{old}})} (\ell', v)$. □

Formalization. The complete proof is carried out in full detail in `OneRuleAtATime.v`. A high degree of automation ensures that the proof of the key invariants is short (about 40 lines) and robust, which enabled us to iterate quickly when designing Kôika's semantics (in all cases, we were able to make changes to the semantics and confirm with few to no proof edits that the new semantics still respected ORAAT).

5 Case Study: A Cycle-Accurate Characterization of a Pipelined System

We now have all the pieces in place to demonstrate how one might use our semantics to prove interesting characteristics of a circuit beyond functional correctness. As a concrete example, we study the pipeline of two combinational functions f_1 and f_2 that our introduction alluded to. Recall that, if f_1 and f_2 both have critical-path length l, then a naïve implementation of their composition $f_2 \circ f_1$ in a single rule `do_f12` would have length $2 \cdot l$. On the other hand, if we decompose the system into two independent rules `do_f1` and `do_f2` connected through a one-element queue, we can reduce the critical-path length to just l.

But this path-length reduction *only matters if we can guarantee that f_1 and f_2 run concurrently in each cycle*, that is, the system actually runs in a pipelined manner. A traditional ORAAT semantics is enough to prove that the "pipelined" system is a correct refinement of the monolithic one (in the sense that it computes the same values) but is not sufficient to prove the two-rules-per-cycle property. In fact, it would be hard even to state such a property because "cycle" is not a meaningful concept in a typical ORAAT formalization.

In the following we present the implementation of a simple pipelined system and sketch its proof. We start with the implementation, in which two rules `do_f1` and `do_f2` are connected through a one-element queue composed of a data-holding register `r` and a flag `empty` indicating whether the queue is empty. Initially, `empty` is set to `true`.

```
rule feed_pipeline =
 clock.wr0(clock.rd0 + 1)
 input.wr0(input_stream(clock.rd0))
```

```
rule do_f2 =
 if empty.rd0 then
  abort
 else
  // dequeue
  out.wr0(f2(r.rd0));
  empty.wr0(true)
```

```
rule do_f1 =
 if empty.rd1 then
  // enqueue
  empty.wr1(false);
  r.wr0(f1(input.rd1))
 else
  abort
```

```
schedule pipeline =
  [feed_pipeline; do_f2; do_f1]
```

One-rule-at-a-time reasoning is sufficient to prove that our pipeline is functionally correct (it computes the composition of f_1 and f_2). The methodology in [9] applies directly.

More interestingly, we can also prove that the system processes one value per cycle and hence deserves to be called a *pipeline*. From the second cycle on, the circuit simultaneously performs `do_f1` and `do_f2` on each cycle (on the first cycle, only `do_f1` can fire, since there is no value in the pipeline for `do_f2` to dequeue and process).

The proof is in two steps. First, by applying our semantics to the program, we derive a sufficient (and, in fact, necessary) criterion for both `do_f1` and `do_f2` to fire simultaneously: both rules will fire in a cycle if `empty` contains `false` at the beginning of that cycle (i.e. the pipeline is not empty).

This property is not an invariant in the one-rule-at-a-time sense, since `do_f2` breaks the invariant by emptying the pipeline, and `do_f1` reestablishes it, but it is a *cycle invariant*: from our semantics, it is straightforward to show that (1) if the pipeline is empty, `do_f1` will fill it, (2) if the pipeline is nonempty, `do_f1` and `do_f2` will both fire in the same cycle, and (3) running `do_f2` and then `do_f2` in a nonempty pipeline maintains a nonempty pipeline (a one-rule-at-a-time argument is enough for this last part).

Hence the pipeline fills and, once full, stays full: from the second cycle on, the pipeline runs both rules on every cycle and processes one element per cycle.

6 Compilation

In this section, we explain the algorithm for generating circuits from Kôika programs. We first describe the target language and the compilation strategy, next explaining in detail how rules and schedulers are translated into circuits and how everything is wired together. In the end, we obtain one combinational state-update function per register, taking as inputs the initial (beginning-of-cycle) values of all registers, outputting the end-of-cycle value of each register.

6.1 Overview

Our compiler targets a minimal RTL language in which each circuit takes the beginning-of-cycle values of all registers as inputs and produces a single output. On the Coq side, we use a dependently typed representation in which the type of each circuit indicates how many bits it computes. Sharing is implicit in the Coq representation.

$$\begin{array}{rcll} \text{Register} & r & & \\ \text{Constant} & b & ::= & \texttt{0b(0|1)+} \\ \text{Circuit} & \mathbf{c} & ::= & b \mid \neg\mathbf{c} \mid \mathbf{c} \wedge \mathbf{c} \mid \mathbf{c} \vee \mathbf{c} \\ & & & \mathsf{Mux}\,(\mathbf{c}, \mathbf{c}, \mathbf{c}) \quad \text{(multiplexing)} \\ & & & \mathbf{R}[r] \quad\quad\quad\;\; \text{(read old value)} \end{array}$$

The denotational semantics $\boldsymbol{\delta}_{\mathcal{R}}(c)$ of these circuits (giving the value computed by a circuit $\mathbf{c}$ as a function of the beginning-of-cycle register values $\mathcal{R}$) are straightforward, and we omit them for space (Mux evaluates to a ternary if). At a high level, our compiler is a compositional denotational semantics into circuits, i.e., a syntax-driven recursive function specifying how to transform programs into collections of circuits, one per register. To minimize the critical path, our compilation strategy does not follow the sequential style used in the semantics; instead, we create circuits that run all rules in parallel, with minimal data forwarding, and only after completing the execution of a rule do we check whether its read/write set conflicts with those of earlier rules. While the circuit that runs rules in parallel could conceivably be obtained through a global Boolean transformation of a circuit running rules sequentially, it is more straightforward to generate the optimized circuit directly. We proceed in three phases.

First, we compile each rule *in isolation* (that is, as if no other rule had executed in the same cycle), generating a collection of signals from the body of each rule:

- An ok circuit indicating whether the rule could fire if it were alone
- A circuit computing a representation of the rule's read/write set (i.e. which registers it needs to read from and write to)
- A circuit indicating which new data, if any, the rule would write in each register

These circuits correspond to a *hardware log*: a finite, compact representation of the log ℓ of all actions performed by the rule, *as if log L were empty*.

Second, we build circuitry to determine whether each rule can in fact be committed. (We say that a rule is "committed" once the scheduler has determined that it is compatible with the ones that preceded it at that point, in which case its writes will be reflected into registers at the end of the cycle. This terminology is justified by the one-rule-at-a-time theorem.) This is done by combining the ok signal of each rule with a check ensuring that the read/write set of that rule is compatible with the read/write sets of rules previously committed in the same cycle (this compatibility testing corresponds to a delayed version of the checks performed against L in the semantics: our compilation strategy delays these checks, because the circuits that we generate execute all rules in parallel). This step also requires appropriate wiring to forward values between rules.

Finally, we compute the end-of-cycle value of each register, based on the accumulated hardware log.

6.2 Compiling Rules

Recall that, according to the semantics in subsection 3.2, an action may either succeed (return Log(...)) or abort (return Fail) depending on whether the premises of the corresponding deduction rule hold. In addition, each action returns a value (possibly tt), appends to a per-action log ℓ capturing the read and write operations that the action performs, and updates an environment Γ of bound variables.

Accordingly, our compiler takes a partial hardware log $\boldsymbol{\ell}$ and a piece of Kôika syntax a, producing a collection of combinational circuits that we write as $(\!(\boldsymbol{\ell}, a)\!)$:

- A circuit carrying the return value of a; we notate it $(\!(\boldsymbol{\ell}, a)\!).\mathsf{ret}$, read as "the ret wire of $\boldsymbol{\ell}$ extended with a."
- An updated hardware log keeping track, for each register, of whether it was read or written at port 0 and 1 by the current rule (four 1-bit signals: $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{rd}_0$, $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{rd}_1$, $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{wr}_0$, and $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{wr}_1$), and, if written, of which value was written at each port (two n-bit signals: $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{data}_0$ and $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{data}_1$)[3]. Additionally, the hardware log tracks whether the rule can safely proceed ($(\!(\boldsymbol{\ell}, a)\!).\mathsf{ok}$).

To construct these circuits, our compiler recursively descends through the syntax tree of a, combining intermediate results into larger circuits. As part of this process, it builds a *compilation context*, a map $\boldsymbol{\Gamma}$ keeping track of the $(\!(\boldsymbol{\ell}, a)\!).\mathsf{ret}$ circuits that were generated for the right-hand sides of the let-bindings of the original program.

[3] The invariant for $(\!(\boldsymbol{\ell}, a)\!).r.\mathsf{data}_0$ is a bit more subtle; this wire starts out holding the value of register r and thereafter carries the latest $\texttt{wr}_0$ performed by the current rule or *any previous one*, if any.

Figure 3 details the transformations performed when encountering each syntactic construct, but it helps to start with a detailed example. Key to understanding the compilation pipeline is realizing that the shapes of the generated circuits closely mirror the semantic rules of subsection 3.2. Consider the Write0 rule, repeated below (with slight premise rearrangement to logically equivalent form):

$$\frac{\begin{array}{c}\Gamma \vdash (\ell, a) \downarrow (\ell', v) \\ (\mathrm{rd}_1, r) \notin \ell' \qquad (\mathrm{wr}_*, r, *) \notin \ell' \\ (\mathrm{rd}_1, r) \notin L \qquad (\mathrm{wr}_*, r, *) \notin L\end{array}}{\Gamma \vdash (\ell, r.\mathtt{wr}_{\mathtt{0}}(a)) \downarrow (\ell' +\!\!+ [(\mathrm{wr}_0, r, v)], \mathtt{tt})}\ \textsc{Write0}$$

To generate a circuit corresponding to $r.\mathtt{wr}_{\mathtt{0}}(a)$, given a partial log $\boldsymbol{\ell}$, we first compile a to obtain an updated log $\llparenthesis \boldsymbol{\ell}, a \rrparenthesis$, reflecting ℓ'. We then synthesize the circuitry corresponding to the ok flag as a conjunction of clauses closely matching each premise of the rule above (the first equivalence below means that if evaluating a returns Fail, then so should evaluating $r.\mathtt{wr}_{\mathtt{0}}(a)$; the next two translate predicates characterizing the log ℓ' into circuits):

$$\begin{aligned}
\Gamma \vdash (\ell, a) \downarrow (\ell', v) &\text{ is mapped to } \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ok} \\
(\mathrm{rd}_1, r) \notin \ell' &\text{ is mapped to } \neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{rd}_1 \\
(\mathrm{wr}_*, r, *) \notin \ell' &\text{ is mapped to } \neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{wr}_0 \ \wedge\ \neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{wr}_1
\end{aligned}$$

Putting it all together, we obtain the following circuit for ok (note that we do not explicitly mention $\boldsymbol{\ell}.\mathrm{ok}$ — this is because its value is already embedded in $\llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ok}$):

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{0}}(a) \rrparenthesis.\mathrm{ok} = \ &\llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ok}\ \wedge \neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{rd}_1 \wedge \\
&\neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{wr}_0\ \wedge \neg\, \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{wr}_1
\end{aligned}$$

The other circuits are more straightforward to construct (we use the shorthand syntax $A.* = B.*$ to mean that A's remaining wires are all the same as B's):

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{0}}(a) \rrparenthesis.\mathrm{ret} &= \varepsilon \quad \text{(empty value, i.e. no wires)} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{0}}(a) \rrparenthesis.r.\mathrm{wr}_0 &= \mathtt{0b1} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{0}}(a) \rrparenthesis.r.\mathrm{data}_0 &= \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ret} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{0}}(a) \rrparenthesis.*.* &= \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.*.*
\end{aligned}$$

The last line means that all wires pertaining to r besides wr_0 and data_0 are unchanged from $\boldsymbol{\ell}$, and in addition that all wires pertaining to *other* registers are unchanged as well. Note that, as alluded to earlier, we did not translate the two premises pertaining to the log L. This is because our compilation strategy defers these checks to the scheduling circuitry. This approach minimizes the number of wires threaded between individual actions, and it is safe because L, unlike ℓ, is unchanged throughout the evaluation of a rule.

6.3 Compiling the Hardware Scheduler

The scheduling circuitry combines compiled rules together in scheduling order, determining which rules can be committed and computing the set of register updates to apply at the end of the cycle. The compilation process is quite similar to

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, x \rrparenthesis.\mathrm{ret} &= \boldsymbol{\Gamma}[x] \\
\llparenthesis \boldsymbol{\ell}, x \rrparenthesis.* &= \boldsymbol{\ell}.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, \mathtt{skip} \rrparenthesis.\mathrm{ret} &= \varepsilon \\
\llparenthesis \boldsymbol{\ell}, \mathtt{skip} \rrparenthesis.* &= \boldsymbol{\ell}.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, \mathtt{const}\ b \rrparenthesis.\mathrm{ret} &= b \\
\llparenthesis \boldsymbol{\ell}, \mathtt{const}\ b \rrparenthesis.* &= \boldsymbol{\ell}.*
\end{aligned}$$

$$\begin{aligned}
&\llparenthesis \boldsymbol{\ell}, \mathtt{if}\ a_1\ \mathtt{then}\ a_2\ \mathtt{else}\ a_3 \rrparenthesis.* \\
&\quad = \mathrm{Mux}\,(\mathbf{c_1}.\mathrm{ret}, \llparenthesis \mathbf{c_1}, a_2 \rrparenthesis.*, \llparenthesis \mathbf{c_1}, a_3 \rrparenthesis.*) \\
&\quad \text{(where } \mathbf{c_1} = \llparenthesis \boldsymbol{\ell}, a_1 \rrparenthesis\text{)}
\end{aligned}$$

$$\begin{aligned}
&\llparenthesis \boldsymbol{\ell}, \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 \rrparenthesis.* = \llparenthesis \mathbf{c_1}, a_2 \rrparenthesis.* \\
&\quad \text{(with } \boldsymbol{\Gamma}[x \mapsto \mathbf{c_1}.\mathrm{ret}] \text{ where } \mathbf{c_1} = \llparenthesis \boldsymbol{\ell}, a_1 \rrparenthesis\text{)}
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, \mathtt{abort} \rrparenthesis.\mathrm{ok} &= \mathtt{0b0} \\
\llparenthesis \boldsymbol{\ell}, \mathtt{abort} \rrparenthesis.* &= \boldsymbol{\ell}.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{0}} \rrparenthesis.\mathrm{ok} &= \boldsymbol{\ell}.\mathrm{ok} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{0}} \rrparenthesis.\mathrm{ret} &= \mathbf{R}[r] \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{0}} \rrparenthesis.r.\mathrm{rd}_0 &= \mathtt{0b1} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{0}} \rrparenthesis.*.* &= \boldsymbol{\ell}.*.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{1}} \rrparenthesis.\mathrm{ok} &= \boldsymbol{\ell}.\mathrm{ok} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{1}} \rrparenthesis.\mathrm{ret} &= \boldsymbol{\ell}.r.\mathrm{data}_0 \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{1}} \rrparenthesis.r.\mathrm{rd}_1 &= \mathtt{0b1} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{rd}_{\mathtt{1}} \rrparenthesis.*.* &= \boldsymbol{\ell}.*.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{1}}(a) \rrparenthesis.\mathrm{ok} &= \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ok}\ \wedge \neg \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.r.\mathrm{wr}_1 \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{1}}(a) \rrparenthesis.\mathrm{ret} &= \varepsilon \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{1}}(a) \rrparenthesis.r.\mathrm{wr}_1 &= \mathtt{0b1} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{1}}(a) \rrparenthesis.r.\mathrm{data}_1 &= \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.\mathrm{ret} \\
\llparenthesis \boldsymbol{\ell}, r.\mathtt{wr}_{\mathtt{1}}(a) \rrparenthesis.*.* &= \llparenthesis \boldsymbol{\ell}, a \rrparenthesis.*.*
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \boldsymbol{\ell}, f\ a_1 \ldots a_n \rrparenthesis.\mathrm{ret} &= \mathbf{f}\ \mathbf{c_1}.\mathrm{ret}\ \ldots\ \mathbf{c_n}.\mathrm{ret} \\
\llparenthesis \boldsymbol{\ell}, f\ a_1 \ldots a_n \rrparenthesis.* &= \mathbf{c_n}.* \\
&\text{(with } \mathbf{f} \text{ the external circuit, } \mathbf{c_1} = \llparenthesis \boldsymbol{\ell}, a_1 \rrparenthesis, \mathbf{c_2} = \llparenthesis \mathbf{c_1}, a_2 \rrparenthesis, \ldots\text{)}
\end{aligned}$$

Figure 3. Translations making up the rule compiler. The $r.\mathtt{wr}_{\mathtt{0}}(a)$ case was discussed in the main text and is not repeated here. Note that unlike in the semantics, the circuit generated for $\mathtt{rd}_{\mathtt{1}}$ has no case split, because the $r.\mathrm{data}_0$ wire always carries the latest $\mathtt{wr}_{\mathtt{0}}$ data if any, or $\mathbf{R}[r]$ otherwise.

that of rules, but this time we are building the hardware log corresponding to L, not ℓ, and there are no return values. Concretely, we need to tackle the following issues:

Forwarding data. The rule compiler that we described previously takes two inputs: the syntax of the rule and a partial hardware log. This log is needed because later rules in scheduling order may observe (through calls to $\texttt{rd}_1$) writes performed by previous rules in the same cycle. In a purely sequential compilation scheme, similar to the way our semantics of Kôika are phrased, we would use the log circuit $\mathbf{L}$ produced by accumulating the logs of all preceding rules. But, for performance reasons, we want to compile the control circuits (rd_0, wr_0, etc.) of all rules in parallel, so we assemble each initial log $\boldsymbol{\ell}_0(\mathbf{L})$ by keeping only the data_0 and data_1 signals instead, as shown in Figure 4.

$$\begin{aligned}
\boldsymbol{\ell}_0(\mathbf{L})\,.\mathsf{ok} &= \texttt{0b1}\\
\boldsymbol{\ell}_0(\mathbf{L})\,.r.\mathsf{data}_0 &= \mathbf{L}.r.\mathsf{data}_0\\
\boldsymbol{\ell}_0(\mathbf{L})\,.r.\mathsf{data}_1 &= \mathbf{L}.r.\mathsf{data}_0\\
\boldsymbol{\ell}_0(\mathbf{L})\,.r.* &= \texttt{0b0}
\end{aligned}$$

Figure 4. Fresh log $\boldsymbol{\ell}$ used when starting rule compilation

Deciding which rules to commit and updating $\mathbf{L}$. The circuits generated by compiling each rule include a flag ok, whose run-time value will indicate whether the rule could fire on its own, or whether it would instead reach an $\texttt{abort}$ or perform two writes to the same location. That signal, however, does not take into account whether the reads and writes performed by the rule are compatible with those of other already-committed rules. This means that our scheduling circuitry must compute whether a rule a is safe to run, by checking its ok flag *and* performing all deferred checks pertaining to the log L. The equations are shown in Figure 5.

$$\begin{aligned}
&\mathsf{wf}\,(\mathbf{L}, \boldsymbol{\ell}) \;=\\
&\quad \boldsymbol{\ell}.\mathsf{ok}\\
&\quad \wedge \bigwedge_r \boldsymbol{\ell}.r.\mathsf{rd}_0 \implies \neg(\mathbf{L}.r.\mathsf{wr}_0 \vee \mathbf{L}.r.\mathsf{wr}_1)\\
&\quad \wedge \bigwedge_r \boldsymbol{\ell}.r.\mathsf{wr}_0 \implies \neg(\mathbf{L}.r.\mathsf{wr}_0 \vee \mathbf{L}.r.\mathsf{wr}_1 \vee \mathbf{L}.r.\mathsf{rd}_1)\\
&\quad \wedge \bigwedge_r (\boldsymbol{\ell}.r.\mathsf{rd}_1 \vee \boldsymbol{\ell}.r.\mathsf{wr}_1) \implies \neg\mathbf{L}.r.\mathsf{wr}_1
\end{aligned}$$

Figure 5. Deciding whether two logs are compatible. $x \implies y$ is used as a shorthand for $\neg x \vee y$, and wf for "will fire."

Putting it all together. We define the final circuit by the recursion below. In words, these compilation rules indicate that a scheduler feeds each control-flow path a representation of the cumulative hardware log updated to take into account the new rule (either an updated $\mathbf{L}$' if the rule can execute and commit or the original $\mathbf{L}$ otherwise) and returns the same results as whichever path was selected based on the computation of $\mathsf{wf}\,(\mathbf{L}, \boldsymbol{\ell})$:

$$\begin{aligned}
\llparenthesis \mathbf{L}, \texttt{cons}\ a\ s \rrparenthesis.* &= \mathsf{Mux}\,(\mathsf{wf}\,(\mathbf{L}, \llparenthesis \boldsymbol{\ell}_0(\mathbf{L})\,, a \rrparenthesis)\,, \llparenthesis \mathbf{L}', s \rrparenthesis.*, \llparenthesis \mathbf{L}, s \rrparenthesis.*)\\
\mathbf{L}'.r.\mathsf{data}_0 &= \llparenthesis \boldsymbol{\ell}_0(\mathbf{L})\,, a \rrparenthesis.r.\mathsf{data}_0\\
\mathbf{L}'.r.\mathsf{data}_1 &= \llparenthesis \boldsymbol{\ell}_0(\mathbf{L})\,, a \rrparenthesis.r.\mathsf{data}_1\\
\mathbf{L}'.r.* &= \llparenthesis \boldsymbol{\ell}_0(\mathbf{L})\,, a \rrparenthesis.r.* \vee \mathbf{L}.r.*
\end{aligned}$$

With that, only the very beginning and the very end of the compilation process are left:

- Feeding the circuit with the initial data for the data_0 wire at the root, as well as the initial read-write set (data_1 does not need to be initialized to any specific value as its value is never used before having been set). For data_0, the value is simply the actual register (i.e. $\mathbf{L}_0.r.\mathsf{data}_0 = \mathbf{R}[r]$), and the read-write sets are blank (i.e. $\mathbf{L}_0.r.\mathsf{rd}_0 = .\mathsf{rd}_1 = .\mathsf{wr}_0 = .\mathsf{wr}_1 = \texttt{0b0}$).
- Performing the actual update of the registers: given a scheduler s, the state-update circuit computing the end-of-cycle value of the register r, $\langle s \rangle_r$, is

$$\begin{aligned}
\langle s \rangle_r = \mathsf{Mux}(&\llparenthesis \mathbf{L}_0, s \rrparenthesis.r.\mathsf{wr}_1,\\
&\llparenthesis \mathbf{L}_0, s \rrparenthesis.r.\mathsf{data}_1, \llparenthesis \mathbf{L}_0, s \rrparenthesis.r.\mathsf{data}_0)
\end{aligned}$$

6.4 Performance Concerns

Performance-minded readers might be worrying at this point that the circuitry that we introduce to track read sets and write sets and to compute conflicts with previously scheduled rules would prove prohibitively expensive. In fact, this cost should be minimal for the following reasons. First, no registers are needed to maintain the hardware logs associated with each rule; these are computed dynamically during the cycle and consumed by the end of the cycle. Second, the write sets and associated data values are needed in any compilation scheme to compute the next state function. Thus, only the read-set part of the hardware log represents the overhead of our compilation scheme. Third, the number of gates needed to compute the intersection of these sets with the read-write log is at most proportional to the number of rules and the sizes of the read and write sets of the rules. Furthermore, for many rules, simple static analysis will tell us the exact read/write sets, and standard Boolean optimization will get rid of most circuitry. In those cases, the Boolean logic associated with intersections will be eliminated by constant propagation. Our compiler includes an optimization pass that performs such constant propagation as well as standard Boolean simplifications and partial evaluation:

$$\begin{array}{lll}
 & c \wedge 0 \longrightarrow 0 & c \wedge 1 \longrightarrow c\\
\neg 1 \longrightarrow 0 & 0 \wedge c \longrightarrow 0 & 1 \wedge c \longrightarrow c\\
\neg 0 \longrightarrow 1 & c \vee 1 \longrightarrow 1 & c \vee 0 \longrightarrow c\\
 & 1 \vee c \longrightarrow 1 & 0 \vee c \longrightarrow c
\end{array}$$

$$\begin{array}{ll} \mathsf{Mux}\,(0, x, y) \rightarrow x & \mathsf{Mux}\,(c, 1, x) \rightarrow c \lor x \\ \mathsf{Mux}\,(1, x, y) \rightarrow y & \mathsf{Mux}\,(c, x, 0) \rightarrow c \land x \\ & \mathsf{Mux}\,(c, x, x) \rightarrow x \end{array}$$

$$\begin{aligned} \mathsf{Mux}\,(c, \mathsf{Mux}\,(c', y, x), y) &\rightarrow \mathsf{Mux}\,(c \land \neg c', x, y) \\ \mathsf{Mux}\,(c, \mathsf{Mux}\,(c', x, y), y) &\rightarrow \mathsf{Mux}\,(c \land c', x, y) \\ \mathsf{Mux}\,(c, x, \mathsf{Mux}\,(c', x, y)) &\rightarrow \mathsf{Mux}\,(c \lor c', x, y) \\ \mathsf{Mux}\,(c, x, \mathsf{Mux}\,(c', y, x)) &\rightarrow \mathsf{Mux}\,(c \lor \neg c', x, y) \\ \mathsf{Mux}\,(c, \mathsf{Mux}\,(c, x_1, x_2), y) &\rightarrow \mathsf{Mux}\,(c, x_1, y) \\ \mathsf{Mux}\,(c, x, \mathsf{Mux}\,(c, y_1, y_2)) &\rightarrow \mathsf{Mux}\,(c, x, y_2) \end{aligned}$$

These optimizations are enough to produce optimal circuits for simple examples like Collatz and to eliminate most overheads due to read-write-set tracking in our processor case study (subsection 6.6), shrinking the generated circuit graph from 80k nodes down to 3k nodes.

To summarize, the extra logic for tracking falls into two categories: (1) Programs where dynamic data-dependency detection reveals no more opportunities to avoid rule conflicts than the BSV compiler's static analyses of rules. We assert in such cases that the same Boolean reasoning should carry over to simplify our scheduling circuitry, eliminating the overhead introduced by our compiler. This assertion holds in the examples explored so far. (2) Programs where dynamic detection of concurrency matters. In this case, programmers control the trade-off through scheduling: a potentially longer critical path with more concurrency, or a regular critical path with Bluespec-style concurrency. This trade-off is similar to the one that BSV users face while using EHRs.

To offer complete quantitative evidence of our claims using large programs will require more work and optimizations in the compiler. Currently, Kôika does not have a module system, without which it may be difficult to capture the sharing of circuitry that comes from the use of port modules, as in a register file or memory system. Finally, the rearrangement of gates to reduce critical-path lengths in RTL remains a mystery in the best of times, and it will require more experimentation to discover whether the patterns generated by the Kôika compiler are able to make adequate use of these optimizations; initial evidence (subsection 6.6) is promising.

6.5 Implementation and Verification

We have implemented the compilation strategy outlined above within Coq, verifying its correctness by connecting Kôika's semantics to those of the minimal RTL that we target. To convert from mini-RTL into Verilog, we use a thin unverified pretty-printing layer that targets a correspondingly small subset of Verilog, and from there we can use standard synthesis tools like Yosys to obtain FPGA bitstreams or ASIC designs.

Proving compiler correctness. Our top-level theorem guarantees that computing register updates by interpreting a scheduler according to our original high-level semantics produces the same results as evaluating the resulting, compiled circuits according to our RTL semantics. Succinctly, recalling that $\delta_s(\mathcal{R})$ is the state-update function mapping old register values to new register values according to a schedule s, that $\boldsymbol{\delta}_{\mathbf{c}}$ is the denotation of circuit $\mathbf{c}$, and that $\langle s \rangle_r$ is the compiled circuit that computes the new value of register `r`, the theorem is

$$\forall \mathcal{R}, s, r.\ \delta_s(\mathcal{R})[r] = \boldsymbol{\delta}_{\langle s \rangle_r}(\mathcal{R})\,.$$

The main difficulty of the proofs stems from the gaps between the original denotational semantics and the generated circuits. First, in the semantics, rules have access to the accumulated log L when deciding whether to allow or reject reads or writes; in the circuits, on the other hand, all rules run concurrently, each executing independently as if it were started with an empty $\mathbf{L}$, and all cross-rule consistency checks are delayed until the computation of the wf signal. Second, while the semantics use sequential logs to keep track of the reads and writes, the circuits that we generate build minimal *hardware logs*, with single bits indicating whether each register has been read from or written to at each port. Third, while the interpreter can fail immediately upon encountering a forbidden action (such as a double write), the ok circuits are built once and must ensure that failures are propagated correctly throughout. Details on the corresponding proofs are given in Appendix A, with full proofs in `CircuitProperties.v` and `CircuitCorrectness.v`.

6.6 A Simple RISC-V Processor in Kôika

To evaluate the cost of dynamic tracking, we wrote a simple 4-stage RISC-V processor (RV32I without interrupts; see Appendix B for architectural description and synthesis methodology) both in BSV and in Kôika, and we compared the results. These results are presented in Figure 6.

Program	mandelbrot	median	tm	qsort
Inst. count	41168976	25981	21867	35405
Cycles (Kôika)	70508760	60692	47550	82837
Cycles (BSV)	70508760	60692	47550	82837

Performance	Critical Path (ps)	Logic Area (μm^2)
Kôika (Retiming)	296.87	5504
BSC (Retiming)	258.45	5437
Kôika	686.29	14115
BSC	651.60	11981

Figure 6. Synthesis and architectural results

Each processor implementation takes roughly 1000 lines of code in its respective language. Both processors took exactly the same number of cycles to compute, which shows that the scheduling in both designs is identical. Our design achieves a respectable critical path (on par with the BSV design). Our design achieves a respectable critical path and area, with a 15% slowdown and minimal area overhead with register retiming, and an 18% area overhead and negligible slowdown without register retiming. Our example shows that it is possible to implement a design of this complexity, control its concurrency, and generate working hardware through a proven compiler.

7 Related Work

We begin with other prior work on hardware design based on atomic actions, followed by contrasting BSV/Kôika-style atomic actions with software and hardware transactions. We will also discuss briefly other competing approaches to hardware design, namely hardware description languages (HDLs) like Verilog and Chisel, high-level synthesis from sequential languages, and synchronous languages.

Prior work on guarded atomic actions: The idea of One-Rule-At-A-Time semantics forms the foundation for all the work on rule-based systems starting with Hoe and Arvind [25] and Hoe [24]. Hoe gave an algorithm for scheduling the maximal number of rules in each cycle, based on static analysis of the read and write sets of each rule. Esposito et al. [14] gave another algorithm that does not necessarily schedule the maximal number of rules but is simpler to implement in hardware, and this algorithm is used by the BSV compiler. The first formal operational semantics of guarded atomic actions were given by Dave et al. [11, 12]. They described the state-transformation function for each rule and posed the rule-scheduling problem as one of rule composition, for which they provided a rich set of combinators. Sequential composition of rules in some sense is more expressive than EHRs, but Dave's scheduling primitives proved difficult to use in practice and were not implemented in the BSV compiler. Also, Kôika's semantics and implementation rely on dynamically computed read and write sets, making it possible to exploit more concurrency than the traditional compilations relying on static analysis.

ORAAT semantics does not dictate that rules be executed in one clock cycle; this is an implementation choice of both the BSV and Kôika compilers. A one-cycle-execution restriction on rules benefits from a very clear cost model, leaving the user responsible for decomposing a rule with a long critical path into multiple rules. Spreading an infrequently executed complex rule across multiple cycles, however, can be quite convenient and can dramatically improve the clock period, hence performance [26, 27]. A recently published open-source compiler for BSV by Greaves [17] includes multicycle rules and a fair intercycle scheduler.

Choi et al. [9] have developed Kami, a system for mechanical verification of proofs for designs expressed in a language with guarded atomic actions. The proofs rely on the ORAAT property and consider only the behaviors arising by executing one rule in one cycle. Thanks to our ORAAT theorem (1), proofs in the Kami style will still be valid in our framework.

Transactional viewpoint: Atomic transactions (or simply transactions) are a common abstraction in distributed software [21–23] and database systems. Considerable effort has also gone into providing hardware support for transactional memory [19, 20]. The ORAAT semantics of our atomic rules is the same as the *serializability* property of atomic transactions. However, software transactions are written in a sequential imperative language and invariably require shadow state to handle aborted transactions (hardware transactional memory uses L1 caches to reduce the cost of the shadow state). Kôika completely avoids the shadow state by exploiting the basic properties of hardware registers, which can be read at the beginning of a clock cycle and updated at the end of the clock cycle. As we said earlier, Kôika's atomic rules hold all the temporary values "in wires" during the clock cycle. For performance, both software transactions and atomic rules rely on interleaved execution of atomic entities, but the cost models and, consequently, the implementation techniques are completely different. In our hardware synthesis, there is no cost associated with an aborted transaction, i.e., a rule that does not commit. For software transactions, one only computes approximations of the read/write sets, because the universe of objects is too big. Unlike for software transactions, it is inexpensive to keep the read sets and write sets associated with an atomic rule, because the number of registers is known statically, and the write sets have to be maintained anyway to update registers at the end of the clock cycle.

Structural hardware-description languages: Traditional HDLs like Verilog and VHDL are structural in the sense that they describe interconnections of boxes, i.e., Boolean gates and registers. The main problem with such languages is that they provide inadequate type checking and lack precise semantics, which makes verification and design refinement a Herculean task. Attempts to clean up the semantics of Verilog have had little success; see for example [29]. A popular way to make Verilog more convenient for programming is by embedding it in a language with a good macro facility, which can provide type safety and good combinators for composition [2, 13]. Another example of a structural language is Chisel [3], which is an embedded DSL in Scala and has a powerful metalanguage for generating complex patterns. (BSV also has a powerful static-elaboration facility based on functional languages.) This line of work does not tackle the difficulty of describing complex interactions between sequential machines, which we believe is the true difficulty of hardware design.

High-level synthesis: Another approach to hardware synthesis is to transform programs written in software languages like C, Python, MATLAB, etc. into hardware [8, 10, 16, 18]. HLS compilers rely on compiler techniques developed for parallel and vector architectures starting in the 1980s. In spite of fundamental limitations of this approach (see for example [1]), the commercial appeal is strong enough that many companies have invested significant resources into building better HLS tools [28, 37]. This approach has shown promise for signal-processing applications but has not been shown to be useful to describe processors and other complex designs.

Synchronous languages: Synchronous languages describe reactive systems with equations between streams of inputs and outputs. Such languages use an abstract notion of synchronicity and clocks and usually have clean mathematical semantics [6]. Recently, a compiler to translate a synchronous language into a minimal subset of C was developed and proven correct [7]. It has also been shown that one can compile synchronous languages to hardware [5, 31], though the challenges there are fairly disjoint from our own problems. The reactive viewpoint has worked well for describing control-theory problems [4], but there is little evidence of its suitability to describe complex hardware.

8 Conclusion

A cycle-accurate description is essential to understand the performance of a hardware circuit. Such descriptions, however, often complicate reasoning about functional properties of the hardware. It has been shown that rule-based descriptions, for example as in BSV [30], and the associated ORAAT semantics allow us to build proof systems, for example Kami [9], that are extremely useful for proving functional properties. In this paper we have presented Kôika, a hardware-description language that allows cycle-accurate specifications in a rule-based system with a user-specified intracycle scheduler. Using the Coq proof system, we have shown that Kôika preserves the ORAAT semantics. We have shown that the Kôika semantics can be used to prove performance properties, for example, that a pipelined system indeed behaves like a pipelined system.

We have also presented an algorithm to compile Kôika into circuits, implemented it, and formally verified that our compiler correctly implements Kôika's semantics. We have used this implementation to compile several examples, including a simple pipelined processor. Kôika's semantics and the compiler use dynamic dataflow analysis, which eliminates concurrency anomalies and entails reasonable hardware overhead.

In the near future, we expect to incorporate a module system in Kôika and experiment with the flexibility provided by user-specified scheduling.

A Verification Details

This section gives more details on the invariants and ideas underpinning our compiler-correctness proof.

Verifying circuit transformations. Large parts of the circuitry that our compilation scheme introduces for faithful implementation of Kôika's semantics can be statically eliminated using Boolean optimizations. Accordingly, our compiler implementation is parametric on a verified circuit-optimization function η (whose correctness criterion is stated as $\forall \mathbf{c}, \mathcal{R}.\ \boldsymbol{\delta}_{\eta(\mathbf{c})}(\mathcal{R}) = \boldsymbol{\delta}_{c}(\mathcal{R})$), which it applies to newly created circuits on-the-fly, as it compiles each source action.

Aligning logs and hardware logs. We need to make sure that the values produced by the semantics and by the circuits match up. Since these values are computed from the logs in the semantics, and from the hardware logs (read-write sets and data_* wires) in the circuits, we need to establish an invariant connecting both. We write $L \sim_{rw} \mathbf{L}$ when $\forall r.\ \boldsymbol{\delta}_{\mathbf{L}.r.*}(\mathcal{R}) = 1 \Leftrightarrow (*, r) \in L$, where $*$ stands for one of rd_0, rd_1, wr_0, or wr_1; i.e. when each circuit tracking reads and writes in $\mathbf{L}$ agrees with L. Separately, we write $L \sim_{data} \mathbf{L}$ when (1) $\forall r.\ \boldsymbol{\delta}_{\mathbf{L}.r.\mathsf{data}_0}(\mathcal{R}) = v \Leftrightarrow \mathsf{last\_wr}_0\,(L) = v$ and (2) $\forall r.\ (\mathsf{wr}_1, r) \in L \Rightarrow \left(\boldsymbol{\delta}_{\mathbf{L}.r.\mathsf{data}_1}(\mathcal{R}) = v \Leftrightarrow \mathsf{last\_wr}_1\,(L) = v\right)$, where $\mathsf{last\_wr}_n$ is the latest `wr`n in L if any, or $\mathcal{R}[r]$ otherwise; i.e. when both circuits tracking write values for each register agree with L. Finally, we write $\Gamma \sim_{\gamma} \boldsymbol{\Gamma}$ when $\forall x.\ \boldsymbol{\delta}_{\boldsymbol{\Gamma}[x]}(\mathcal{R}) = \Gamma[x]$, i.e. when the context of compiled bindings kept by the compiler agrees with the binding values in the semantics. We prove lemmas characterizing how these relations interact with muxing of circuits, and we show that under these equivalences the hardware implementation of dynamic checks is faithful to the checks performed against ℓ in the rule semantics.

Tracking dynamic failures. The key lemma is to prove that $\boldsymbol{\delta}_{(\!(\boldsymbol{\ell}_0(\mathbf{L}), a)\!).\mathsf{ok}}(\mathcal{R})$ is 0 if $[\![a]\!](\mathcal{R}, L) = \mathsf{Fail}$. We proceed by induction; the main difficulty is to prove that if we reach a failure state at any point within a rule, then the wf computation properly returns zero when the results of the rule are eventually combined with those of previous rules. For this, we start by defining a partial order on single-bit circuits (`circuit_le` in `CircuitProperties.v`): we say that $\mathbf{c}_1 \leq_{\mathcal{R}} \mathbf{c}_2$ if $\boldsymbol{\delta}_{\mathbf{c}_2}(\mathcal{R}) = 0 \Rightarrow \boldsymbol{\delta}_{\mathbf{c}_1}(\mathcal{R}) = 0$. It is easy to show that $\wedge$, $\vee$, and Mux are increasing, and $\neg$ decreasing, in $\leq_{\mathcal{R}}$. This relation extends to read-write sets by comparing them elementwise. We prove two lemmas using this relation: first (`rwset_circuit_le_compile_action_correct`), that ok is decreasing, and that read-write sets are increasing (i.e. for all a and $\boldsymbol{\ell}$, the read-write set of $(\!(\boldsymbol{\ell}, a)\!)$ is greater than the read-write set of $\boldsymbol{\ell}$). Second, that wf itself is decreasing as well, i.e. that $\mathsf{wf}\,(\mathbf{L}, (\!(\boldsymbol{\ell}, a)\!)) \leq_{\mathcal{R}} \mathsf{wf}\,(\mathbf{L}, \boldsymbol{\ell})$.

Final invariant. With these pieces in place, we can now state our main lemma — its proof follows by induction from the lemmas above (this invariant establishes the correctness of the part of the compiler that handles individual rules; there is a corresponding but simpler one for schedules):

$\forall a, L, \ell, \Gamma, \mathbf{L}, \boldsymbol{\ell}, \boldsymbol{\Gamma}.$

$$\left.\begin{array}{r}\Gamma \sim_{\gamma} \boldsymbol{\Gamma} \\ L \sim_{rw} \mathbf{L} \\ L \mathbin{+\!\!+} \ell \sim_{data} \boldsymbol{\ell} \\ \boldsymbol{\delta}_{\mathsf{wf}(\mathbf{L},\boldsymbol{\ell})}(\mathcal{R}) = 1\end{array}\right\} \Rightarrow \begin{cases}\begin{cases}\boldsymbol{\delta}_{\llparenthesis \boldsymbol{\ell}, a \rrparenthesis}(\mathcal{R}) = v \\ L \mathbin{+\!\!+} \ell' \sim_{data} \llparenthesis \boldsymbol{\ell}, a \rrparenthesis \\ \boldsymbol{\delta}_{\mathsf{wf}(\mathbf{L},\llparenthesis \boldsymbol{\ell}, a \rrparenthesis)}(\mathcal{R}) = 1\end{cases} \\ \qquad \text{if } \Gamma \vdash (\ell, a) \downarrow_{(L,\mathcal{R})} (\ell', v) \\ \boldsymbol{\delta}_{\mathsf{wf}(\mathbf{L},\llparenthesis \boldsymbol{\ell}, a \rrparenthesis)}(\mathcal{R}) = 0 \\ \qquad \text{otherwise}\end{cases}$$

These lemmas, and the final theorem, are proven in the file `CircuitCorrectness.v`.

B Architectural Description of the Processor

The core is a simple 4-stage pipelined processor (Fetch, Decode, Execute, Writeback) with a bypassing path from Writeback to Decode and a bypassing redirection from Execute to Fetch.

The decoding and execution logic have been written side-by-side in Kôika and BSV to make them match as closely as possible. The branch predictor is the simplest predictor: $pc+4$ (i.e., assuming we never jump). The scheduling order picked is Writeback, Execute, Decode, Fetch (chosen explicitly in Kôika and inferred by bsc). We wrote pipeline and bypass FIFOs to connect all the stages and test different orderings. The reported results use pipeline FIFOs to connect all the stages.

To obtain area and critical-path numbers, we compiled both designs (BSV and Kôika) to Verilog and fed the resulting code through an open-source synthesis toolchain composed of Yosys [36] and ABC [15], configured to use a 45nm PDK [34] with and without register retiming.

To collect architectural performance numbers, we connected the cores to 32KB of BRAM preloaded with a binary image of the RISC-V program we were running.

The design were simulated using Verilator [35] but also successfully synthesized for an FPGA using Vivado 2017.4 for AC701. The designs both have a maximal clock frequency between 100MHz and 110MHz.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Grant No. CCF-1521584 and by the National Science Foundation under Grant No. HR001118C0018 . Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DARPA.

References

[1] Abhinav Agarwal, Man Cheuk Ng, and Arvind. 2010. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed-Solomon Decoder. *Embedded Systems Letters* 2, 3 (2010), 72–76. https://doi.org/10.1109/LES.2010.2055231

[2] Markus Aronsson and Mary Sheeran. 2017. Hardware software co-design in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017.* 162–173. https://doi.org/10.1145/3122955.3122970

[3] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012.* 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[5] Gérard Berry. 1992. Mechanized Reasoning and Hardware Design. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, Chapter Esterel on Hardware, 87–104. http://dl.acm.org/citation.cfm?id=149943.149953

[6] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[7] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 586–601. https://doi.org/10.1145/3062341.3062358

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011.* 33–36. https://doi.org/10.1145/1950413.1950423

[9] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *PACMPL* 1, ICFP (2017), 24:1–24:30. https://doi.org/10.1145/3110268

[10] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[11] Nirav Dave, Arvind, and Michael Pellauer. 2007. Scheduling as Rule Composition. In *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2007), May 30 - June 1st, Nice, France.* 51–60. https://doi.org/10.1109/MEMCOD.2007.371249

[12] Nirav H. Dave. 2011. *A unified model for hardware/software codesign.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/68171

[13] Conal Elliott. 2017. Compiling to categories. *PACMPL* 1, ICFP (2017), 27:1–27:27. https://doi.org/10.1145/3110271

[14] T. Esposito, M. Lis, R. Nanavati, J. Stoy, and J. Schwartz. 2005. System and method for scheduling TRS rules. United States Patent US 133051-0001.

[15] Alan Mishchenko et al. [n.d.]. ABC: System for Sequential Logic Synthesis and Formal Verification. https://github.com/berkeley-abc/abc.

[16] Daniel D. Gajski. 2001. SpecC Design Environment. *System Design* (2001), 217–235. https://doi.org/10.1007/978-1-4615-1481-7_5

[17] David J. Greaves. 2019. Further sub-cycle and multi-cycle schedulling support for Bluespec Verilog. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2019, La Jolla, CA, USA, October 9-11, 2019*, Partha S. Roop,

Naijun Zhan, Sicun Gao, and Pierluigi Nuzzo (Eds.). ACM, 2:1–2:11. https://doi.org/10.1145/3359986.3361199

[18] Sumit Gupta, Nikil D. Dutt, Rajesh Gupta, and Alexandru Nicolau. 2004. Loop Shifting and Compaction for the High-Level Synthesis of Designs with Complex Control Flow. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*. 114–121. https://doi.org/10.1109/DATE.2004.1268836

[19] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael K. Chen, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software. *IEEE Micro* 24, 6 (2004), 92–103. https://doi.org/10.1109/MM.2004.91

[20] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional Memory Coherence and Consistency. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*. IEEE Computer Society, 102–113. https://doi.org/10.1109/ISCA.2004.1310767

[21] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. 2008. Composable memory transactions. *Commun. ACM* 51, 8 (2008), 91–100. https://doi.org/10.1145/1378704.1378725

[22] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, Alan Jay Smith (Ed.). ACM, 289–300. https://doi.org/10.1145/165123.165164

[23] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[24] James C. Hoe. 2000. *Operation-centric hardware description and synthesis*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/86439

[25] James C. Hoe and Arvind. 2000. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*. 511–518. https://doi.org/10.1109/ICCAD.2000.896524

[26] Michal Karczmarek and Arvind. 2008. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*. 24–31. https://doi.org/10.1109/ICCAD.2008.4681547

[27] Michal Karczmarek, Arvind, and Muralidaran Vijayaraghavan. 2014. A new synthesis procedure for atomic rules containing multi-cycle function blocks. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*. 22–31. https://doi.org/10.1109/MEMCOD.2014.6961840

[28] Mentor. [n.d.]. ModelSim. https://www.mentor.com/products/fpga/verification-simulation/modelsim/.

[29] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Rosu. 2010. A formal executable semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*. IEEE Computer Society, 179–188. https://doi.org/10.1109/MEMCOD.2010.5558634

[30] Rishiyur S. Nikhil. 2004. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04)*. IEEE Computer Society, Washington, DC, USA, 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[31] Frédéric Rocheteau and Nicolas Halbwachs. 1991. Implementing Reactive Programs on Circuits: A Hardware Implementation of LUSTRE. In *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*. 195–208. https://doi.org/10.1007/BFb0031993

[32] Daniel L. Rosenband. 2005. Hardware synthesis from guarded atomic actions with performance specifications. In *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*. 784–791. https://doi.org/10.1109/ICCAD.2005.1560170

[33] Daniel L. Rosenband and Arvind. 2004. Modular scheduling of guarded atomic actions. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*. 55–60. https://doi.org/10.1145/996566.996583

[34] North Carolina State University. [n.d.]. FreePDK45. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[35] Veripool. [n.d.]. Verilator. https://www.veripool.org/wiki/verilator.

[36] Clifford Wolf. [n.d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[37] Xilinx. [n.d.]. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.