Effective Simulation and Debugging for a High-Level Hardware Language using Software Compilers

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

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

Stella Lau
MIT CSAIL
Cambridge, Massachusetts, USA
stellal@csail.mit.edu

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

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

ABSTRACT

Rule-based hardware-design languages (RHDLs) promise to enhance developer productivity by offering convenient abstractions. Advanced compiler technology keeps the cost of these abstractions low, generating circuits with excellent area and timing properties.

Unfortunately, comparatively little effort has been spent on building simulators and debuggers for these languages, so users often simulate and debug their designs at the RTL level. This is problematic because generated circuits typically suffer from poor readability, as compiler optimizations can break high-level abstractions. Worse, optimizations that operate under the assumption that concurrency is essentially free yield faster circuits but often actively hurt simulation performance on platforms with limited concurrency, like desktop computers or servers.

This paper demonstrates the benefits of completely separating the simulation and synthesis pipelines. We propose a new approach, yielding the first compiler designed for effective simulation and debugging of a language in the Bluespec family. We generate cycle-accurate C++ models that are readable, compatible with a wide range of traditional software-debugging tools, and fast (often two to three times faster than circuit-level simulation). We achieve these results by optimizing for sequential performance and using static analysis to minimize redundant work. The result is a vastly improved hardware-design experience, which we demonstrate on embedded processor designs and DSP building blocks using performance benchmarks and debugging case studies.

CCS CONCEPTS

• **Hardware → Simulation and emulation**; **Hardware description languages and compilation**; • **Software and its engineering → Compilers**.

KEYWORDS

Hardware simulation, hardware debugging, compilation

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446720

ACM Reference Format:
Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective Simulation and Debugging for a High-Level Hardware Language using Software Compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446720

1 INTRODUCTION

Most hardware designs are expressed in a spectrum of languages ranging from low-level RTL (Verilog [27] or VHDL [20]) to sequential software languages with annotations for high-level hardware synthesis (Vivado HLS [31], Handel-C [14], Clash [23], etc.). Different points on this scale entail different trade-offs.

Verilog offers limited programming abstractions and composability, which makes it tedious to write and debug, but it offers developers fine-grained control over the resulting circuits. HLS systems — that is, hardware design systems that start from software languages to generate Verilog — offer rich abstractions and excellent debugging and simulation facilities but poor control over generated circuits. This is not surprising because the sequential computation model of software languages is deeply at odds with hardware computation models, which try to run all parts of a circuit in parallel all the time.

Rule-based languages, such as Bluespec [21], Kôika [3], and Kami [6], offer an interesting middle ground, with predictable performance and yet high-level, usable, and composable semantics. Rule-based designs describe the manipulation of (hardware) state elements using state-transforming atomic *rules*, which (appear to) execute sequentially. An RTL compiler then introduces concurrency by translating rules into individual circuits that run in parallel yet preserve the illusion of sequentiality (*one-rule-at-a-time semantics*).

Significant research effort has been dedicated to synthesizing high-quality hardware from Bluespec designs. Comparatively little effort has been expended on cycle-accurate simulation, debugging, and testing of rule-based designs: these tasks are typically performed at the generated-Verilog level. As a result, despite its convenient abstractions, Bluespec is not particularly pleasant to debug. Debugging is mainly performed using printf and wave analysis.

In addition, simulating Bluespec-generated Verilog is not particularly fast. The key issue is that compilers that target RTL optimize for fast hardware — not fast simulation! Efficient execution

in hardware requires maximum parallelism, so code generators (and hardware engineers) will often introduce additional circuitry (and increase area) to increase concurrency (and reduce critical paths). This directly hurts simulation performance when running on a limited-concurrency platform such as a desktop computer or a server.

This paper tackles both of these issues by fully decoupling simulation and synthesis while keeping simulation cycle-accurate (we say that a model is *cycle-accurate* if the values that it computes for all of the state elements and I/Os of a design match those produced by the real circuit at every cycle, regardless of how these values are computed). We design a new compilation backend for Kôika (a Bluespec-inspired language that provides fine-grained control over scheduling) that targets C++ instead of Verilog. This backend supports all features of Kôika and is separate from Kôika's RTL backend. By implementing optimizations specifically geared towards sequential execution and leveraging existing C++ compiler technology, we can produce much-faster software models: speedups of 2× to 3× are typical in our experience. This performance does not come at the cost of intelligibility: by mapping Kôika abstractions to zero-cost idiomatic C++ patterns, we can obtain readable models that closely reflect the structure of the original design.

Concretely, our Kôika simulator, called Cuttlesim, works by compiling each design into a custom C++ program. The baseline version of such a program reads like a fairly direct transliteration. However, we also apply optimizations like exiting early when an atomic rule aborts, so that later code can be skipped (in RTL, in contrast, the whole rule is computed in all cases). Some of these optimizations depend on static analyses that are much easier to perform on Kôika than on RTL.[1] We then discover opportunities to specialize the data structures and algorithms used for tracking conflicts between rules. These models are compatible with the whole range of traditional software-debugging tools, enabling a whole new hardware-debugging and verification experience. The information these tools return is easy to map back to the original Kôika hardware design (which can be matched nearly line-by-line with the generated C++ code), without making changes to standard tools like `gdb` and `gprof`.

So, overall, we suggest a new workflow for hardware development: write in rule-based hardware-description languages, compile automatically to C++ programs, debug and profile with standard software tools, repeat, and only later synthesize to RTL. This paper presents our Cuttlesim prototype and some representative uses of it in developing embedded-class RISC-V processors and simple DSP components. The paper makes the following contributions:

(1) We show how using completely separate toolchains for software simulation and hardware synthesis leads to faster simulation and improved debugging experience.
(2) We describe techniques to build fast software models of rule-based designs, using lightweight transactions.
(3) We show that rule-based designs are amenable to heavy optimization through static analysis that exploits the simplicity of the input language.
(4) We give concrete evidence of the value of this approach using simulation performance benchmarks and debugging and design-exploration case studies.

We note that we are focused on designing concrete circuits, not evaluating microarchitectural ideas independently of implementation. As such, we focus on cycle-accurate simulation, not on processor simulation in the style of Gem5 [2] or ZSim [24].

Cuttlesim is part of the Kôika HDL distribution; it is freely available under the GNU GPL at https://github.com/mit-plv/koika/.

2 RULE-BASED DESIGNS AND THEIR SIMULATION OVERHEADS

This section explains why simulating rule-based designs after compilation to Verilog is inefficient and why rule-based designs are in fact amenable to fast simulation. We start with a brief reminder about rule-based designs and their compilation to Verilog. Details about Kôika and its hardware synthesis are available in [3].

2.1 Rule-Based Designs

Languages in the Bluespec family encourage designers to decompose hardware designs into small units of work called *rules*. In a pipelined CPU design, each rule would typically encode one stage (fetch, decode, execute, writeback, etc.).

Rules are written in a simple language with traditional constructs like conditionals, variable bindings, and combinational functions, plus three special primitives: *read*, *write*, and *abort*. These primitives define how rules change the system state and communicate within a cycle. Each read and write is annotated with a port (0 or 1): a read at port 0 observes the value of a register at the beginning of the cycle; a read at port 1 observes the latest write at port 0 if any or the beginning-of-cycle value otherwise; a write at port 1 is not observable until the next cycle. An abort cancels a rule's execution.

The semantics of the language specify that rules (should appear to) execute atomically, one-at-a-time: that is, the results computed by a design should be the same as if exactly one rule executed per cycle, with no concurrency and no intracycle communication. Implementing these semantics requires ruling out linearity violations, typically through static analysis or dynamic tracking of read-write sets (for example, a write at port 0 precludes a read at port 0 in the same cycle: if both rules ran in the same cycle, the second one would observe the original value, not the result of the write). In Kôika (the language that Cuttlesim implements), each program has *rules* as well as an explicit *scheduler*, which specifies the order in which rules should (appear to) run.

As an example, assume that we are modeling a two-state machine, whose internal state is represented with a register `x`. An additional register `st` keeps track of whether the machine is in state *A* or *B*. The dynamics of the state machine can be described using two rules, `rlA` and `rlB`, each predicated to run only if the machine is in the right state, as follows:

```
rule rlA =
  if (st.rd0 != 'A) abort;
  st.wr0('B);
  let new_x := fA(x.rd0(), get(input)) in
  x.wr0(new_x); put(output, new_x)
```

[1] These optimizations are possible because maintaining cycle-level accuracy only requires computing the same state-element updates in each cycle in Cuttlesim models and in RTL: there is no need to preserve the *amount* of computation that is performed in each cycle.

Here, `fA` is some combinational function doing potentially complex work, and `input` and `output` are external ports used to communicate with the outside world. Rule `rlB` is similar, and since the two rules are mutually exclusive it does not matter which order we schedule them in. In each cycle, one of the two rules will execute, read `input`, update the machine's internal state, and write it to `output`.

To understand why the hardware that Kôika generates for this design does not lend itself to fast simulation, we first need to understand how Kôika compiles designs to Verilog.

2.2 Generating Circuits for Rule-Based Designs

Kôika generates hardware by creating one circuit per rule, in isolation, then wiring these rule circuits together as specified by the scheduler. Individual rule circuits track reads and writes to each register, at each port. Scheduling logic ensures that the reads and writes performed by two rules are compatible and merges their results: if the execution of a rule leads to a conflict or an explicit abort, its results are discarded.

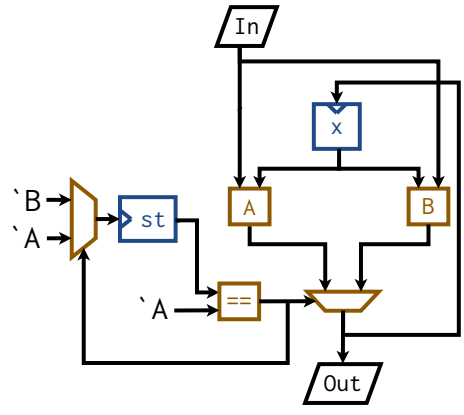
The circuit generated by Kôika for the trivial example above is a combinational circuit that computes the new values of `st` (*A* or *B*) and `x` (the machine's internal state). The resulting Verilog looks roughly like this:

```
module stm(input wire CLK,
           input wire [31:0] in,
           output wire [31:0] out);
   reg st = 1'b0;
   reg [31:0] x = 32'b0;

   wire [31:0] fB_out, fA_out;
   fA mod_fA(.x(x), .in(in), .out(fA_out));
   fB mod_fB(.x(x), .in(in), .out(fB_out));

   assign out = st == `A ? fA_out : fB_out;
   always @(posedge CLK) begin
      st <= st == `A ? `B : `A;
      x <= out;
   end
endmodule
```

The same circuit is represented in the figure below:



In this simple example, the circuits generated for `st` and `x` are both Muxes of the values computed by each rule, predicated by the state that the machine is in at the beginning of the cycle. In a more complex setting, a nontrivial Boolean circuit would decide whether the results of each rule should be committed or discarded.

Note how the circuits corresponding to all rules run in every cycle, though only one of them "commits" (that is, just one of them updates the state).

2.3 Overheads in Simulating Kôika

The compilation strategy employed by Kôika [3] is to ensure maximal concurrency between rules by running all rules concurrently in every cycle and reconciling their results a posteriori. Unfortunately, this makes it inefficient to simulate in software that runs on a limited-concurrency platform.

Optimizing for circuits and optimizing for simulation on a CPU are different goals, and it makes sense that Kôika would optimize for good circuit performance. In hardware, the cost (critical path) of the generated circuit will be $|\texttt{Mux(st == A, fA\_out, fB\_out)}| = |\texttt{Mux}| + \max(|\texttt{st == A}|, |\texttt{fA\_out}|, |\texttt{fB\_out}|)$, where $|X|$ is the cost of circuit X.

Unfortunately, simulating this circuit as written leads to unnecessary computation. A typical Verilog simulator will generate code to compute *both* `fA_out` and `fB_out` in every cycle, *then* mux their results. The sequential running time, thus, would become $|\texttt{Mux}| + |\texttt{st == ...}| + |\texttt{fA}| + |\texttt{fB}|$. We note in passing that at such a small granularity, there is almost nothing to be gained from introducing thread-level concurrency because of the high cost of synchronization and data movement. Decomposing the simulation of a large design to run on multiple cores can make sense, but the amount of concurrency in a circuit is usually much larger than the number of cores in a typical computing platform.

This is a trivial example, and accordingly one might hope to optimize it and run only the relevant branch of the Mux (Verilator does not, but other simulators might, and it would be reasonable for it to grow that capability). But on a more complex example, additional rules can easily complicate the picture and make the optimizations intractable. For example, maybe in state *A* the machine reads from an input FIFO, and the rule fails if this FIFO is empty. Or maybe the whole state machine is embedded in a larger design, and both rules can only execute if no preceding rules generated conflicts: for example, the system might support an external `reset` command whose execution prevents `rlA` and `rlB` from running in that cycle (this would ensure that `reset` does not need to be sequenced with `rlA` and `rlB` and thus does not lengthen the machine's critical path).

A compiler aware of Kôika's semantics, starting from the original design, would save a lot simply by faithfully mapping Kôika behavior to a sequential execution model. It would generate code that stops executing a rule as soon as it fails. In our example, its cost (running time) would be $|\texttt{st == A}| + |\texttt{fA}| + |\texttt{st == B}|$ if the machine is in state *A*, and $|\texttt{st == A}| + |\texttt{st == B}| + |\texttt{fB}|$ if the machine is in state *B*: in no cycle do we need to pay for the cost of executing both rules.

The important concept here is that Kôika's semantics allow rules to exit early, either from conflicts with previous rules or from explicit aborts. In software, the ideal implementation of these early exits is to jump straight to the next rule, skipping whatever remains of the current rule. Our key contribution is showing how to design

a compiler that takes advantage of Kôika's "early-exit" semantics and generates C++ models that are perfectly suited to a CPU's mostly sequential execution paradigm. We do this by designing a lightweight transaction system, using static analysis to further reduce transaction costs. Static analysis is particularly easy in this context, because Kôika's semantics are simple and explicit.

There is a surprising benefit to this perspective shift. By designing the simulator carefully, we can produce readable models, amenable to all sorts of software-focused methodologies (step-through debugging, code-coverage analysis, etc.). For example, here is how the C++ model `rlA` might look, annotated with Gcov execution counts, after 500 cycles:

```
-:500 DEF_RULE(rlA) {
-:500   if (READ0(st) != state::A)
-:250     return false;
-:250   WRITE0(st, state::B);
-:250   bits<32> y = fA(READ0(x), get(input))
-:250   WRITE0(x, y); put(output, y);
-:250   COMMIT();
-:250 }
```

It turns out that simply collecting code-coverage statistics on the generated C++ code offers an incredible wealth of architectural information, without having to add a single hardware counter.

We give evidence of the performance of our simulator, and detailed examples of the new design methodologies that our tooling enables, in section 4. But first, we present the technical insights that make these results possible.

3 COMPILING KÔIKA FOR SIMULATION

The performance of Cuttlesim's models is achieved through two distinct technical contributions: a lightweight implementation of transactions that is shared by all designs, and design-specific optimizations that are derived using static analysis.

Details on the structure of the models that Cuttlesim generates, how they lend themselves to further optimization by C++ compilers, and what techniques we employed to make them both readable and seamlessly debuggable are given in Appendix A.

3.1 Matching Kôika's Transactional Semantics

The original Kôika paper describes Kôika's semantics using *rule logs*, each of which keeps track of the reads and writes performed by a single rule; and a *cycle log*, which keeps track of the reads and writes performed by all the rules that are scheduled in a cycle. Each cycle starts with an empty *cycle log*. Rules are executed one-by-one, and every time a rule attempts to perform a read or write, a check is made against both the cycle log and rule log up to that point in the cycle to ensure that the action is permitted; if not, the whole rule aborts. When a rule executes successfully, its rule log is appended to the cycle log. Otherwise, the rule log is discarded. At the end of the cycle, the values of the design's state elements (registers) are updated based on the reads and writes accumulated in the cycle log.

The C++ models that Cuttlesim produces closely follow this idea. Each design is compiled into a C++ class. Rules become functions that construct rule logs. The design's scheduler becomes a function that calls each rule in turn. The key difficulty in compiling Kôika is to devise efficient representations for these logs, and to implement the transactional semantics of the rules, which require maintaining shadow states. We start with a naive model, which we then refine incrementally into an efficient implementation. This lets us explain each optimization individually.

In a naive model, Cuttlesim keeps three pieces of data: the values of the design's registers at the beginning of the cycle (its publicly observable state), a cycle log, and a rule log. The logs are arrays containing a structure for each register, which indicates whether the register was read or written at port 0 or 1, plus two data fields (`data0` and `data1`) that keep values written at port 0 or 1 in that register. Reads and writes are implemented as C functions that check whether the operation is permitted by Kôika's semantics and either update the rule log or cause the rule function to abort. Specifically, a read at port 0 checks for writes at any port in the cycle log and returns the beginning-of-cycle value of the register; a read at port 1 checks for writes at port 1 in the cycle log and returns the most recent write0 value from either log, falling back to the beginning-of-cycle state; a write at port 0 checks for reads at port 1 and writes at port 0 or 1 in both logs; and a write at port 1 checks for other writes at port 1 in both logs.

When a cycle begins (resp., entering a rule), Kôika clears the read-write sets of the cycle log (resp., the rule log) and invalidates its data fields. When a rule succeeds, its log is *committed* into the cycle log: read-write sets are or-ed together, and the cycle log's data fields are updated to reflect writes found in the rule log. When a rule fails, nothing needs to be done: the rule log will be reset upon entering the next rule, so the rule simply returns early. When a cycle completes, the model's register values are updated by *committing* the cycle log: if a write occurred at port 1 the `data1` value is copied from the log to the state; otherwise if a write occurred at port 0 the `data0` value is used instead; and otherwise the state is left unchanged.

3.2 Optimizing Transactions

This simple implementation of transactions is slow: models spend inordinate amounts of time checking and copying read-write sets, copying data between logs, and committing results. We improve the design through a sequence of refinements, starting with those that are design-independent.

Separate read-write sets and data. Our naive logs store read-write sets and data fields together. This makes logical sense, but it makes clearing read-write sets costly: we need to zero out parts of a structure interleaved with data. A much better approach is to store read-write bitsets separately from write data: this way, resetting all read-write sets is just a matter of zeroing out a structure, which is cache-friendly and efficient. Concretely, this means changing the type of logs to hold two structures: one for read-write sets and one for values written.

Accumulate logs instead of merging them. Keeping the cycle log L and the rule log ℓ separate makes many operations more costly than they need to be: writes need to perform checks on both logs; reads at port 1 need to look for data in both logs and in the beginning-of-cycle state; and committing a rule log requires or-ing read-write sets together (this is very fast) but also checking for writes at either

port in the rule log and optionally copying that data to the cycle log (this is very slow). To speed up these two operations, we change to keeping a cycle log L and an *accumulated* rule log $L +\!\!+ \ell$, ensuring that the full accumulated log (not just its read-write sets) is reset upon entering each rule. This makes checks for write operations much simpler (they only need to check the accumulated rule log) and speeds up rule commits significantly (committing a rule is now a plain copy from the accumulated log).

Reset on failure, not on entry. When a rule fails, we just exit from the corresponding function. This works because the next rule will reset the accumulated rule log by copying the cycle log into it. This reset is redundant when a rule completes successfully: committing the accumulated rule log already ensures that the cycle log and the accumulated rule log match. We thus enforce a new invariant: the accumulated log should match the cycle log at the end of each rule. To maintain this invariant we need to reset the rule log upon failures (by copying the cycle log into it) and to reset the read-write set of the cycle log at the beginning of each cycle. This makes failures more costly, but it allows us to eliminate the resets performed upon entering each rule.

Merge `data0` *and* `data1`. Keeping `data0` and `data1` separately is *almost never* necessary, except in uncommon code like the following:

```
rule rl = r.wr0(1); r.wr1(2); r.rd0(); r.rd1()
```

Assuming that register `r` held value 0 at the beginning of the cycle and was neither written nor read previously, this rule would execute successfully, with the $\texttt{rd}_0$ reading 0 and the $\texttt{rd}_1$ reading 1. Kôika's semantics dictate that the call to `read0` should return `r`'s initial value, 0, and that the call to `read1` should return the latest value written at port 0 (ignoring the `write1` that happened in the same rule), i.e. 1. The value 2 would only be observable after the end of the cycle. If the rule log kept a single data value, the `write1` would overwrite the 1 with 2, and the `read1` would incorrectly observe it. This type of code is rare in real designs, as it is widely considered an anti-pattern in Bluespec. Instead, it can be rewritten, either by moving the `write1` down, or by moving the `read1` up, or by storing the value written at port 0 into a local variable and referencing that instead of calling `read1`. Since this pattern is easy to detect and eliminate, Cuttlesim rejects designs in which it appears, producing an error message and asking users to apply a simple refactoring (the refactoring could be automated, but this pattern is so rare that it does not warrant the effort). Merging the data fields nearly halves the space occupied by the internal state of Cuttlesim models, and it saves time at the end of the cycle: instead of selectively committing either `data0` or `data1` at the end of each cycle, we can check for either write and commit the same data.

Eliminate beginning-of-cycle state. Merging `data0` and `data1` allows us to further debloat the model by entirely eliminating the beginning-of-cycle state kept in addition to the logs. By changing the model to initialize the data parts of both logs to the registers' initial values (instead of leaving them indeterminate), we establish a new invariant: the data stored in both logs at the end of each cycle matches the values stored in the beginning-of-cycle state. The implementation of reads needs to change to read the logs as appropriate. Eliminating the separately kept state saves nearly a third of the remaining model memory, eliminates end-of-cycle commits entirely, saves time in reads, and even allows mid-cycle snapshots.

Taken together, these optimizations yield lightweight and relatively efficient transactions that faithfully implement Kôika's semantics. Profiling, however, reveals that models still spend a lot of time copying data: each commit or reset requires copying entire logs, including the data and read-write states of *all* registers. We now show how the cost of commits, resets, and read-write checks can be reduced dramatically using design-specific optimizations.

3.3 Leveraging Design-Specific Knowledge

All the optimizations described below leverage information gathered using a straightforward abstract-interpretation pass. This pass annotates each read, write, and abort within a rule with a conservative approximation of the (unaccumulated) rule log at that point in the program, plus one Boolean per register indicating whether any of the operations on this register might cause failures (due to conflicts) within that rule. Additionally, it generates an approximation of the whole-cycle log by combining the individual rule logs, plus Booleans indicating whether each register might cause failures[2].

Minimize read-write sets. Kôika's semantics give each register two read ports and two write ports. To determine whether reads and writes to these elements are valid, Cuttlesim tracks four Booleans per state element, indicating which operations have occurred. In most cases, some of this tracking information is redundant.

First, though Kôika's semantics track reads at port 0, this tracking is only useful as part of a concurrent compilation scheme where rules are all executed in parallel and conflict resolution is delayed. When compiling to a sequential model, as in Cuttlesim, the conflict is flagged as soon as the read is attempted: the `read0` part of read-write sets is unused and can be removed.

More generally, we can use the information gathered in our static analysis to classify registers: "plain registers" are read and written only at port 0; wires are written at port 0 and read at port 1; and "EHRs" ("ephemeral history registers") make more complex use of read and write ports. Then, Cuttlesim can exploit this classification to save memory and simplify conflict checking.

Eliminate read-write sets for "safe" registers. A register is *safe* if the reads and writes performed on it can never fail (the register cannot be a source of inter-rule conflicts). This is the case if all reads and writes on it are ordered in such a way that the corresponding checks always succeed. For a simple register, this would mean never reading after writing; for a wire, this would mean never performing a write (at port 0) after a read (at port 1); for other EHRs, this would mean satisfying the whole set of conditions specified by Kôika's semantics. The static-analysis pass performed by Cuttlesim computes a conservative approximation of whether each register is safe; for safe registers, it completely discards read-write sets and performs reads and writes directly, without checking the usual read and write preconditions. This change speeds up reads and writes

[2]The update formula for the flag indicating whether operations on a register can cause failures is essentially a tribool version of Figure 5 from the original Kôika paper [3].

as well as commits, failures, and the read-write-set reinitialization performed at the beginning of each cycle.

Restrict commits and rollbacks to each rule's footprint. The static approximation of each rule log provides an upper bound on its footprint (the set of registers that it reads or writes). Cuttlesim generates custom commit and reset functions that copy and roll back the read-write sets of only those registers that may be read (at port 1) or written, as well as the data parts of only those registers that may be written. However, if a rule touches most of the registers in a design, Cuttlesim reverts to copying whole logs (it is typically faster to perform a single `memcpy` between two logs than to perform a large number of individual field copies).

Speed up early failures. Kôika rules commonly have guards, explicit checks that cause a rule to abort if a condition does not hold. Typically, these checks happen very early, before attempting to write registers; hence, if the rule aborts, there are no modifications to roll back. In that case, Cuttlesim produces code that exits the rule without rollback.

4 EVALUATION

Cuttlesim models are fast and enable new styles of hardware debugging and design exploration. In this section, we support these claims using performance benchmarks (subsection 4.1) and case studies (subsection 4.2). Instructions to run the artifact included with this paper are given in Appendix B.

4.1 Simulation Performance

To evaluate Cuttlesim's performance, we compare primarily against Verilator, an open-source state-of-the-art Verilog simulator[3].

Experimental design. Our evaluation revolves around the simulation of different variants of an embedded processor core supporting the RV32I&E flavors of the RISC-V ISA (minus system instructions, interrupts, and exceptions) running a simple integer-arithmetic benchmark. To demonstrate that Cuttlesim also behaves well on designs without much control logic, we also evaluate the performance of Cuttlesim on two purely combinational circuits: a small finite impulse response filter and the butterfly parts of a large FFT design.

Of course, the point of this evaluation is to measure simulation performance, not to showcase the architectural qualities of specific embedded designs. Our benchmarks are described in Table 1.

As Kôika has not yet been used to design very large systems, our benchmarks are all small-to-medium-sized (tens to thousands of lines of Kôika or Bluespec code, or hundreds to tens of thousands of lines of Verilog code). Consequently, we do not make claims about Cuttlesim's performance on very large designs.

Experimental setup. All benchmarks were run on an Intel Core i7-4810MQ CPU @ 2.80GHz, using the compiler settings recommended by Verilator for maximum performance.

Results. We use our benchmarks to answer three questions:

[3]The authors of Verilator write that "*Verilator has typically similar or better performance versus the closed-source Verilog simulators (Carbon Design Systems Carbonator, Modelsim, Cadence Incisive/NC-Verilog, Synopsys VCS, VTOC, and Pragmatic CVer/CVC).*"

Table 1: Our benchmarks. Metaprogramming examples use code generation. Combinational examples each include a single rule and no scheduling or conflicts. Specialization and heavy optimizations can lead to very small Verilog line counts.

	M	C	SLOC Kôika	SLOC Cuttlesim	SLOC Verilog	Cycles
`collatz`	✗	✓	38	42	13	1G
Trivial state machine						
`fir`	✓	✓	126	102	18	1G
Finite impulse response filter						
`fft`	✓	✓	266	802	341	30M
Part of a Fast Fourier Transform						
`rv32i`	✗	✗	1436	3462	787	25.1M
Small RISCV core (branch predictor: pc + 4)						
`rv32e`	✗	✗	1436	2648	413	25.1M
Embedded variant of `rv32i` (predictor: pc + 4)						
`rv32i-bp`	✗	✗	1706	9855	3087	23.7M
`rv32i` with a better branch predictor (btb + bht)						
`rv32i-mc`	✗	✗	2047	20575	2855	46.8M
Dual-core variant of `rv32i` (predictor: pc + 4)						

Q1: Can Cuttlesim models run faster than a state-of-the-art Verilog simulator? Yes. We answer this question by comparing the execution time of various Kôika designs when simulated directly with Cuttlesim or indirectly by compiling them to Verilog using the preexisting Kôika compiler and running them with Verilator. Figure 1 shows our results: on control-heavy designs like CPU cores, Cuttlesim is multiple times faster than Verilator. On combinational circuits, Cuttlesim's advantage is narrower, as expected. Other simulators that we benchmarked against (CVC and Icarus) were orders of magnitude slower than Verilator.

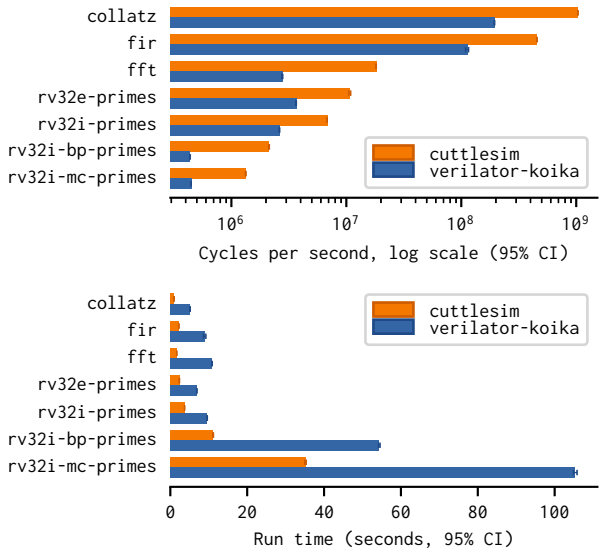


Figure 1: Performance of Verilator and Cuttlesim models

Q2: Is Cuttlesim's advantage only due to Kôika's compiler generating inefficient Verilog circuits? No. We answer this question by benchmarking Cuttlesim models against Verilator simulating Verilog code generated by the commercial Bluespec compiler from equivalent designs. We know that the Verilog generated by Kôika works well with synthesis tools, as the circuits it produces tend to have critical paths and areas comparable to Bluespec-generated ones. However, Kôika targets a much smaller subset of Verilog than Bluespec does. This is for soundness reasons. Kôika's compiler is formally verified against a minimal specification of circuits: the smaller the subset of Verilog used, the more assured we can be that tools will interpret it consistently. Figure 2 shows our results: the Verilog code that Kôika generates simulates roughly within a factor two of that generated by Bluespec for an equivalent design (notably, this number varies depending on the exact version of Verilator used; we notice that the code produced by Bluespec runs faster with Verilator 4, while the code produced by Kôika runs faster with Verilator 3.9). In all cases, it is very likely that a *different* compiler to Verilog could produce significantly better Verilator performance, but optimizing for simulation performance is unnecessary if we have a separate simulator, and indeed in that case we would prefer to focus on the quality of the synthesized hardware (as a concrete example, we could generate much simpler circuits by not introducing concurrency when compiling Kôika's schedulers, but the result would be much slower after synthesis).

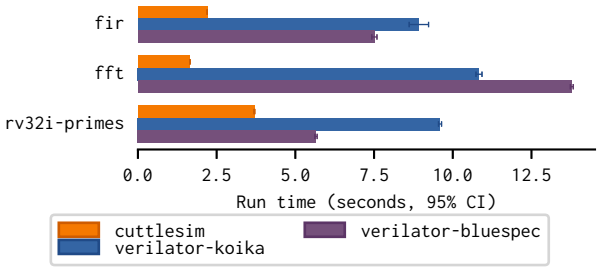


Figure 2: Performance of models on equivalent Bluespec and Kôika designs

Q3: How sensitive is Cuttlesim's performance to external factors, especially compiler choices and tool versions? Somewhat sensitive. We answer this question by compiling Cuttlesim and Verilator models using GCC and Clang and benchmarking the results. Figure 3 shows our results: we find that execution times vary but that Cuttlesim's speed advantages over Verilator are relatively stable.

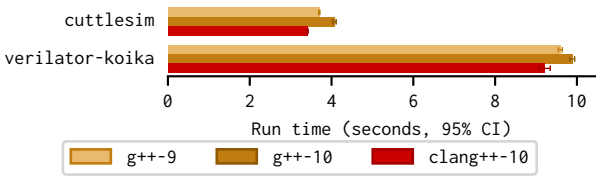


Figure 3: Performance of Verilator and Cuttlesim models

4.2 Debugging, Verification, and Digital Design Exploration Case Studies

Hardware designers spend a significant amount of time simulating designs in order to validate correctness, understand performance properties, and explore design trade-offs. As such, effective design-exploration and debugging tools, which enable designers to understand *how a design works* or *why it does not work* as expected, are invaluable in the development process.

In this section, we show how Cuttlesim improves the state of the art in debugging and architectural exploration of rule-based designs. We walk through a series of experiential case studies illustrating various aspects of the hardware design and debugging process: functional-correctness debugging of a cache-coherence protocol, functional validation of a design using randomized testing, performance debugging of an embedded processor core, and design exploration adding a branch predictor to an existing processor.

All of these case studies focus on cases where cycle-accuracy is paramount: we are working on concrete designs and using Kôika to generate actual circuits. High-level simulators like Gem5 and Zsim would not be applicable.

Case study 1: Debugging a cache-coherence protocol. To illustrate the debugging process with Cuttlesim, we look at a simple case where a programmer is debugging a deadlock in a 2-core machine with L1 "child" caches and a "parent" protocol engine implementing the MSI cache-coherence protocol. To determine what state the system is stuck in, the programmer runs the Cuttlesim model of the system in `gdb` until reaching the deadlock state. Next, they use `gdb`'s interactive interface to print information corresponding to relevant state. In particular, they recall that there are status registers such as miss status handling registers (MSHRs), tracking protocol state.

MSHRs are structures containing various pieces of information, including a tag that is either `Ready`, `SendFillReq` (indicating a cache miss and needed to send a request to the parent), or `WaitFillResp` (indicating waiting for a response from the parent). The tag type was implemented in Kôika as an enum, and the MSHRs as a struct. The enum names have semantic meaning that are preserved in the generated C++ model, and fields of the structure can be accessed naturally by name without doing bit slicing. Furthermore, the programmer does not have to write custom pretty-printers.

Now, the programmer observes that Core 0's cache is deadlocked in the `WaitFillResp` state and the parent protocol engine is in the `ConfirmDowngrades` state (due to an upgrade request from `Invalid` to `Modified` for an address held by Core1). To determine why the rule corresponding to `ConfirmDowngrades` was not executing (and hence why there was no state transition away from `ConfirmDowngrades`), they set a breakpoint on the call to `FAIL()`, which indicates an early exit from a rule. We consider two possibilities.

Suppose `gdb` indicated the failure was caused by a *conflict* between rules. The programmer puts a watchpoint (hardware breakpoint) on the relevant read-write set and executes in reverse (this is made possible by reverse debugging tools like `rr` [11] or GDB's native recording facilities). The compiler stops where the previous write happened, indicating an accidental `write1` instead of `write0`, conflicting with the rule's `read1`.

Contrast this experience with debugging at the Verilog level. Determining why a rule fails means going through each part of the fail signal to tease out what made it true. Then finding the source of the previous write means wading through a forest of other signals, which is tedious and error-prone.

Alternatively, suppose the failure was caused by an explicit `abort`. There are multiple possibilities here: perhaps the downgrading has not finished, or main memory has not responded with the cache line. To determine the cause, the programmer steps through the rule in `gdb` (they are able to do this due to Cuttlesim's fine-grained simulation of transactions, allowing sequential exploration of what code paths were taken). They realize the rule failed due to intermediate state unexpectedly indicating that downgrading has not finished, despite observing that the other core has indeed downgraded its state. Printing out the intermediate state, they realize that it was erroneously computed, successfully pinpointing the problem.

Traditional methods of debugging similar bugs might involve adding `$display` statements (so-called `printf()` debugging) or wave-form debugging (e.g. using GTKWave) with Verilog. However, traditional `printf()` debugging often suffers from having to go through multiple (time-consuming) recompilation cycles, as it is not immediately clear what state is relevant, and the debugger does not want to be overwhelmed with irrelevant state. With wave-form debugging, we sacrifice the high-level abstractions and close correspondence with original source code, requiring an additional mental step to map the logging output to the original source.

With Cuttlesim, there was no need to recompile the design in order to add logging statements: interactive debugging enables programmers to print just the relevant bits of state, as they become needed — including stopping halfway through the execution of a cycle to print the intermediate stated produced by the execution of a few rules.

Furthermore, the ability to step through (sometimes in reverse) the generated model and observe state at fine granularity was useful to test local assumptions about the code interactively. Unlike Verilog-level debugging, stepping through interactively makes it very clear which parts of the design execute in a given cycle.

Finally, we note that this process mostly resembled high-level software debugging of state machines.

Case study 2: Functional verification with scheduler randomization. A good rule-based design should use its scheduler for performance but not for functional correctness: designs should work regardless of the order that rules are executed in. We would like to verify this property experimentally, using randomized testing to validate the design under many different schedulers. Without Cuttlesim and without significant modifications to Kôika's compiler, a limited approximation of this could be achieved by creating many copies of the design with different schedules and compiling each of them independently. Instead, Cuttlesim's C++ models make it trivial to run this experiment in full generality: in C++, it suffices to write a `cycle()` function that calls rules in random order, unlike the default `cycle()` implementation which follows the order specified by the user-supplied scheduler. We used this methodology to gain confidence in the RISC-V cores that we evaluated for performance.

Case study 3: Performance debugging. A standard methodology for designing processors in rule-based languages is to focus initially only on functional correctness in the one-rule-at-a-time semantics (ignoring interrule concurrency and cycle boundaries) and then fine-tune the concurrency by deciding which subset of rules happens in each cycle.

This section describes another flavor of hardware "debugging": architectural performance tuning. Concretely, this involves choosing bypassing paths (using interrule communication through read-write ports) and a scheduler that, together, maximize performance.

Suppose a programmer observes (on a 4-stage pipelined processor with an idealized single-cycle memory) that retiring 100 NOP instructions took 203 cycles. This suggests suboptimal performance, for one would assume that the pipeline would take one cycle per instruction on a program containing no branches and thus no misses.

To investigate, the programmer starts a `gdb` session to step through the execution, following a NOP instruction through the pipeline rule-by-rule. They observe that a NOP instruction is never decoded in the same cycle that an older NOP is executed. Taking advantage of Cuttlesim's ability to step through individual rules, they observe that the decoding stage checks the scoreboard for outstanding writes to the source registers of the instruction being decoded (to prevent read-after-write hazards where an instruction would observe a stalled value not yet updated by an older instruction). In this case, the scoreboard marked the previous NOP as a dependency on the NOP being decoded, and so the new NOP could not be decoded.

One could think that NOP instructions do not write and so do not generate dependencies. However in RISC-V, a NOP is encoded as `ADDI x0 x0 0`, where `x0` is a special non-writeable register always containing `0`. Likely, the processor designer neglected to implement a special case for tracking the dependencies on register `x0`, creating an unintended dependency between the NOP instructions.

Identifying such a bug using traditional tools is particularly difficult because one needs to go *backward*. First, the debugger must ensure that the compiler did not erase any intermediate signals. Next, the programmer must locate and display the stuttering signals (using some Verilog display debugging), and finally they print all the signals used in the computation of the stuttering control signal to identify the cause. With Cuttlesim, the programmer simply steps *forward* through the code and observes the point where the rule fails. Since the generated model corresponds closely to the original source code, this is straightforward and there is no additional mental step to relate the model to the original source.

Case study 4: Branch-prediction exploration. In this section, we describe the process of improving the branch-prediction mechanism of a baseline processor that only had a simple "PC + 4" predictor. The goal of this section is not to equip the processor with a state-of-the-art branch prediction but rather to illustrate the process of architectural refinement in rule-based designs when using Cuttlesim.

Concretely, we need to add a Branch Target Buffer in charge of recording the target addresses of branch and jumps and a Branch History Table in charge of tracking and predicting whether branches are taken or not, and we must update the mechanism that

handles mispredictions. We name the two designs `baseline` and `bp`.

Traditionally, to measure the improvement achieved by such an architectural change, we would add hardware performance counters, iteratively gathering increasing amounts of data (instructions executed per cycle, number of mispredicted instructions, number of cycles spent waiting for a subsequent instructions, etc.).

In Cuttlesim, we can gather all those statistics at once, without adding a single piece of counting hardware. We use a code-coverage tool called Gcov, which measures the number of times that each line of a C++ model was executed. Since the model matches the source design closely, these counts naturally provide detailed architectural information.

The following listings show representative snippets of Gcov output, in the baseline design and in the improved design:

```
// Execute stage (baseline)
14890635: bits<32> nextPc = ctrlResult.nextPC;
14890635: if (nextPc != decoded.ppc) {
 2071903:   WRITE0_FAST(pc, nextPc);
```

```
// Execute stage (improved branch predictor)
14890635: bits<32> nextPc = ctrlResult.nextPC;
14890635: if (nextPc != decoded.ppc) {
  165753:   WRITE0(pc, nextPc);
```

```
// Scoreboard logic (baseline)
21424532: if (score1 != 0 || score2 != 0) {
 9211172:   FAIL();
```

```
// Scoreboard logic (improved branch predictor)
21579776: if (score1 != 0 || score2 != 0) {
 9211302:   FAIL();
```

We learn that the number of mispredictions went down from 2,071,903 to 165,753. From the same Gcov run, we also learn that for this specific program the decoding of instructions is very often stalled by the scoreboard, because of read-after-write hazards. This is explained by missing bypassing paths, forcing the processor to insert bubbles between back-to-back data-dependent arithmetic instructions. From this evaluation and more in-depth inquiry, the working architect may want to think of ways to reduce that potential bottleneck.

Designing profiling harnesses using hardware counters can take significant work. Cuttlesim, in combination with Gcov, enables us to collect these performance numbers with low effort and high speed, making quantitative evaluation of rule-based designs significantly easier.

5 CORRECTNESS AND INTEROPERABILITY

Introducing a separate compilation toolchain for simulation has many advantages, but it also introduces risks: some features available in RTL simulators may not be available in Cuttlesim, and there may be bugs in the models generated by Cuttlesim.

5.1 Cycle Accuracy

Kôika's semantics guarantee that rules appear to execute atomically, and user-provided schedules impose an apparent execution order: hence, all Kôika programs are fully deterministic (this is unlike Bluespec, where the schedule is chosen by the compiler). As a result, Kôika programs have clear and unambiguous performance characteristics in terms of which rules will fire in a given cycle or how many cycles a design will take to reach a given result on a given input. (This is quite different from high-level synthesis, where part of the compilation process involves deciding what runs in which cycle. In Kôika, the programmer can know exactly how many cycles something will take before invoking the compiler to Verilog, and the compiler to circuits is formally verified to preserve these timing properties.)

Cuttlesim implements rules and schedules faithfully: one cycle of a C++ model exactly corresponds to one cycle of the original design, as specified by Kôika's semantics. As a result, any semantic or timing divergence between the C++ models and the generated Verilog code would be due to bugs in Cuttlesim or in user-supplied implementations of external functions.

5.2 Interoperability

Traces. Cuttlesim models include facilities to generate VCD files, a simple and popular format for recording traces of hardware designs. These traces do not include the values of intermediate internal signals generated by Kôika's compiler (indeed, Cuttlesim does not compute these values, since it uses a separate toolchain), but they include all values written to publicly visible state elements; these values match those generated by Verilator and other HDL simulators.

Snapshots. Cuttlesim can snapshot its state to disk and reload these snapshots to resume execution from a previously reached state. As a result, it is possible to mix optimization levels or even simulators: after running a few million cycles with Cuttlesim at `-O3` optimization and saving the design's state to disk, one can reload it into a Cuttlesim model compiled with all debugging options turned on for interactive step-through debugging; or one can load that into a Verilator model compiled from the RTL version of the same design to debug integration with other Verilog components.

External functions. Hardware designs commonly make use of external IP such as memories, peripherals, etc. In Kôika, these are accessed through external functions. When compiling to RTL, these are mapped to wires on the design's interface or internal Verilog modules, and for RTL simulation one uses Verilog models provided by the IP's designer. When compiling to C++ using Cuttlesim, two options exist: users can write their own C++ model of the component, or they can compile the vendor-supplied Verilog model to C++ using Verilator and link the resulting libraries into their Cuttlesim models.

5.3 Correctness

We use a mix of formal verification and testing to reduce the chance of bugs introduced by Cuttlesim:

- We formally verify some of the static analyses that Cuttlesim performs. For example, one of our optimizations needs to check if a subexpression is pure (free of side effects such as variable assignments or register reads and writes). For this, Cuttlesim calls a function written and verified using the Coq proof assistant [26].

- We use differential testing on a range of examples, ensuring that the traces produced by Kôika's reference interpreter (written in Coq), by Verilator, and by Cuttlesim models agree. To make comparisons between Verilator and Cuttlesim easier, we made sure that Cuttlesim's VCD traces match those generated by Verilator exactly, down to signal names.
- We use standard unit tests and integration tests, including running self-checking C programs on our RISC-V cores to catch regressions.

6 RELATED WORK

Different hardware-design communities use very different languages and tools and have different requirements in terms of the quality of generated circuits (when circuits are generated) and importance of timing fidelity of simulation. We discuss several of these communities and their approaches in this section.

Digital design and Verilog. Digital design usually begins with writing RTL, and Verilog is a de-facto standard for doing so. Verilog was originally designed with explicit concern for software simulations [27] of circuit designs, but by the 1990s it became the standard interface for hardware synthesis as well because it provided a layer of abstraction over gate libraries. Verilog is actually structured around two sublanguages – a purely structural language to describe circuits and a behavioral language with higher-level features that are not intended to be synthesized to hardware but instead used to write testbenches and behavioral models of the hardware.

Because of its pervasive use, a rich variety of tools that work in conjunction with Verilog are available. For example, as designs became larger, people started generating structural Verilog using facilities coming from metalevel languages [1, 7, 25]. Still, most of these languages provide only limited support for functional and performance debugging of complex designs.

When it comes to Verilog simulation, the programmer has several options depending on which subset of Verilog one is interested in simulating. When one is interested in simulating synthesizable designs expressed in the structural subset of Verilog with a single (or few) clocks, the best choice is often to do cycle-based simulation. In this case, the standard approach is to translate Verilog to C to get C functions that compute the cycle state updates, yielding excellent performance [15, 28]. In contrast, to simulate more of the Verilog language (including behavioral constructs), simulators traditionally use an event-based (or activity-based) approach, e.g. in Icarus [30].

In both cases, the debugging experience is not ideal, leaning heavily on either printf debugging (`display` statements in Verilog) or direct observation of wave forms. When Verilog is compiled to C, the C generated is not intended to be read, and so C debugging tools are not of much use.

FPGA simulation. Another way to achieve very fast Verilog simulation is to map the design on an FPGA. There are two main inconveniences in doing so: (1) the synthesis, placement, and routing flow is very slow (it is not uncommon when targeting a modern FPGA to wait several hours for the design to be ready to be run); and (2) some hardware structures do not map well over FPGAs, and changing them reduces the fidelity of the simulation. Sophisticated techniques have been devised to preserve the cycle-accuracy of the original RTL even when FPGA implementations use different hardware structures [22, 29]. Significant work has also gone into improving the architectural debugging experience when using FPGAs. For example, [19] runs two copies of the design offset in time by, for example, 1 million cycles. When an assertion violation is observed in the front-running design, the simulation stops, allowing the designer to observe the state of the design trailing behind, from before the assertion was violated and hopefully before things went wrong. The user then saves the state, makes sure that the state of the design is sound, and then slowly steps (either on FPGA or even better in a software model) through the design to investigate at which moment it took an unexpected path that ended up violating an assertion later. In this case, the FPGA gives a way to get quickly to a state close to the bug, when standard software simulation could take days to reach that state.

When the architecture simulated is a design distributed on a network, which would not fit on a single FPGA, the FireSim framework [18] may be used for accurate simulation of both the network and the endpoint, using multiple instances in the AWS cloud.

Higher-level abstractions for digital designs. Rule-based languages [3, 6, 16, 21] provide an intermediate middleground. They embody a clean concurrency model that is intrinsically useful for debugging, formal verification, and digital-design exploration. Such designs can also be translated predictably and efficiently to RTL. However, these languages' tools piggyback on RTL simulators for simulation. So far they have not exploited the extra structure coming from the rule abstraction for software simulation - that is what this paper is about.

HLS. High-level synthesis [4, 8–10, 12, 13, 17, 23] is the field of research focused on translating a subset of a software language like C, C++, or Haskell to circuits. The seduction of this approach comes from the fact that for a number of applications, especially in signal processing, the code is already available in some software language, and thus simulating such systems in software is straightforward.

The main difficulty is usually in generating circuits of predictably high quality. For fixed-dataflow accelerators, the approach has been quite successful, even seeing commercial success in recent years [31]. However, there has been little evidence of the success of the approach in designing complex control-heavy machines, like out-of-order processors. Theoretically, functional simulation of the C source should give similar results to the simulation of the RTL, but there is no a-priori notion of cycle-accuracy in such simulation. [5] points out that to match the cycle-by-cycle behavior of the synthesized circuits, the choices made by the synthesis engine need to be reconstructed exactly as-is in the software simulator, taking into consideration optimizations performed by the synthesis engine. In one dimension, this observation is the same for us: our simulator needs to implement the schedule specified by the program. But because this schedule is explicitly specified by the program, instead of constructed and optimized by the compiler, our simulator does not need to reverse-engineer the choices made by the Kôika to Verilog compiler to provide cycle-accurate simulation.

High-level architectural exploration. Computer architects explore many complex microarchitectural alternatives before building a microprocessor. It is quite difficult to produce an RTL-level design for each alternative. A proper evaluation also requires running huge benchmark programs on each design, so simulation speed is paramount. Hence, most architects evaluate ideas with models [2, 24, 32] that do not represent actual machines' cycle counts accurately.

Evaluating architectural ideas at a higher level of abstraction is at least three orders of magnitude faster than simulating the corresponding design at the cycle-accurate level. It is also relatively easy to change software simulators, but the constant danger of this approach is that the simulated design can easily omit critical details of the hardware design. One source of simulation speed in this approach is *direct execution*: if we simulate an x86 machine on an x86 platform, most of the simulator's instructions can be executed directly on the underlying hardware.

7 CONCLUSION

Rule-based languages offer high-level semantics, powerful abstraction facilities, and composability, together with fine-grained control over generated circuits. They enable rapid prototyping and development of hardware designs without sacrificing circuit quality, offering an enticing middle ground between raw RTL and high-level synthesis from software languages.

Until now, most research on rule-based languages had focused on semantics and compiler technology for circuit generation. Little attention had been paid to simulation, debugging, or testing. These tasks were simply performed at the Verilog level, using generic RTL tooling.

We have shown that a much better experience is possible by fully decoupling hardware synthesis from simulation and debugging. We built a specialized compilation toolchain from the Kôika language to C++ that leverages high-level semantic properties to obtain significant speedups over state-of-the-art RTL-level simulators. Because our compiler preserves the structure of the designs, the cycle-accurate models that it generates can be used for debugging, exploration, validation, and testing, enabling hardware designers to leverage the whole ecosystem of software debugging. Through case studies, we have illustrated a new style of hardware development, dramatically improving over the state of the art in hardware-design debugging.

Although we have focused on rule-based designs, our insights have broad applicability, and we hope to see more applications of software tools to improve the hardware-development workflow in the future. The recent explosion of open-source hardware designs, toolchains, and processes, along with widespread availability of cheap FPGAs, has allowed a whole new set of hardware hobbyists to join the hardware-development community. We hope that improved tooling, especially of the kind that software developers are used to working with, will further lower the barriers to entry and make hardware design even more approachable.

ACKNOWLEDGMENTS

We thank our shepherd Adrian Sampson and the anonymous reviewers of our submission for their insightful feedback and their help in improving this paper.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-NA0003525. This work was funded in part by NNSA's Advanced Simulation and Computing (ASC) Program, in addition to the Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0018 and the National Science Foundation under Grant No. CCF-1521584.

This paper describes objective technical results and analysis. 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 the U.S. Department of Energy, DARPA, or the United States Government.

A COMPILER DETAILS

To assemble a model, Cuttlesim proceeds in 4 steps:

- A Kôika source file is parsed into an untyped AST.
- Its well-formedness is checked using a simple type-checker.[4]
- A partially verified static-analysis pass annotates the type-checked AST with information about reads, writes, and `aborts`.
- A code-generation pass informed by the results of this analysis generates C++ code.[5]

A.1 Structure of Cuttlesim Models

The C++ models that Cuttlesim generates closely follow the structure of the corresponding Kôika hardware designs. Each design is translated into a single header file containing C++ type definitions (matching the type definitions used in Kôika) and a model (a class encapsulating the state of the model as well as methods to interact with it). The generated code can then be driven by a user-written testbench that supplies implementations for external functions, instantiates the model, and runs it. An additional header-only library is included by all models; it includes shared datatype definitions, implementations of Kôika's primitives, and convenience functions to instantiate and drive models.

Each model class exposes a simple interface: `cycle()` steps the simulation forwards by one cycle, updating the model's internal state accordingly; `snapshot()` returns a copy of the design's observable state (the values stored in registers); `reset()` reinitializes the simulation to a given state. Individual snapshots can be pretty-printed to standard out or written to disk in a standard hardware trace format (VCD) for analysis or comparison against Verilog models. Additional methods allow users to run or produce traces over multiple cycles.

Datatypes. Each Kôika datatype is translated to a matching C++ type: bits to a custom templated bitvector type (so that `bits 5` in Kôika becomes `bits<5>` in C++), enumerations and structures to their C++ equivalents, and arrays to C++11's `std::array`. For

[4] The original Kôika system only existed as a domain-specific language embedded inside of the Coq proof assistant, using dependent types to capture well-formedness, but we have extended it by adding an untyped layer and a typechecker as well as a standalone input language.

[5] A Makefile and a testbench stub are generated alongside the model to give easier access to debuggers, profilers, trace visualizers, and other tools.

structures and enums, the compiler also generates an implementation for equality comparison, pretty-printing, packing values into bitvectors, and unpacking values out of bitvectors. For arrays and bitvectors, these functions are derived using template metaprogramming.

Models. Kôika designs are composed of rules calling internal and external functions, tied together by a scheduler. Accordingly, Kôika models are C++ templates parameterized by a type of external functions, with one method per rule and per internal function, plus one method (`cycle` to implement the scheduler). Rule methods take no arguments and implement a transactional API: either they succeed, return true, and update the model's internal state; or they fail, return false, and leave the model untouched[6]. An internal function takes the same arguments as its Kôika counterpart plus a pointer to write its results to, and it returns a Boolean indicating whether it succeeded (failures propagate all the way up to the calling rule and cause it to abort). External function calls are translated to method calls on an instance of a user-supplied `extfuns` class passed to the model as a template parameter.

Rules and function bodies. To match the style and structure of the original programs as closely as possible, Cuttlesim translates rules and functions in a single pass, going directly from Kôika ASTs to C++ without generating intermediate representations. When possible, Kôika constructs are translated to corresponding C++ forms (conditionals to if statements, let bindings to C++ variable declarations, variable assignments to C++ assignments, sequences to consecutive statements, etc.). The main sources of divergence are transactional constructs (register reads and writes as well as `aborts`, whose semantics require special support), evaluation order (C++ does not specify function-argument evaluation order, while Kôika does), and expression/statement discrepancies. Kôika does not distinguish between statements and expressions: all expressions can contains reads, writes, variable assignments, and early returns (`aborts`). C++ does distinguish, so Cuttlesim hoists out stateful subexpressions and introduces temporary variables as needed to match Kôika's semantics.

A.2 Generating Compiler-Friendly Code

Introduce in-place mutation. Kôika values are immutable: the primitive that changes a field of a structure returns a fresh structure with that field changed, which the caller may assign back to the same variable. The same is true for arrays, and it neatly matches to the execution model of circuits, where replacing a set of wires by another is essentially free. This style is suboptimal in C++, however: there is no point in creating a copy of a structure or an array and immediately discarding the original just to modify one of its fields. Cuttlesim recognizes this pattern and translates it to a direct assignment.

Special-case common patterns. A common way to create a fresh zero-initialized structure or array in Kôika is to create a zeroed-out vector of the right width and unpack it to obtain a value of the appropriate type. Cuttlesim symbolically evaluates the argument of each call to unpack and, if it is 0, produces code that initializes the structure directly and skips the unpacking. Other similar patterns provide easy gains without diminishing the readability of the generated code.

Generate optimizable code. Cuttlesim mostly restrict itself to high-level optimizations on the generated code: we expect users to interact with the generated C++ models, so we leave most of the low-level optimizations to later compilation passes, in the C++ compiler. As a result, the performance of Cuttlesim's models is heavily dependent on how well the compiler optimizes them.

To facilitate study of the assembly that compilers produce, Cuttlesim has a minimal mode in which it produces just the core of the model, stripped of all pretty-printing and reporting, reduced to a single function performing n iterations of the model. We spent considerable time optimizing the code generated by Cuttlesim to improve this output. Often, this means allowing the compiler to learn more information about the code. For example, it should not matter whether read-write sets are reset at the beginning or at the end of each cycle, at least as long as logs are not modified between calls to `cycle()`. When resetting at the beginning of the cycle, however, the compiler does not need complex analysis to propagate the information that read-write sets start out all-zero, nor does it need to restrict the corresponding optimizations to the cases in which nothing is called between consecutive calls to `cycle()`. This allows for much better optimization, especially if rules are small enough to be inlined. Another example: bitvectors of sizes 1 through 8 are all represented as 8-bit `uint8_t` values. In the library routine that converts a `bit<1>` to a Boolean, the compiler does not know that the routine's input must have all top 7 bits set to 0. Explicitly stating it (`if (v > 1) __builtin_unreachable()`) allows the compiler to generate better code. Sometimes even adding more code can help the compiler generate less: in the preceding example, changing `bool(v)` into `bool(v & 1)` enables the same optimizations. Similarly, when a rule touches many registers, it is often faster to copy the whole log than to reset only touched registers using individual copies, because the former compiles to heavily optimized copying routines. Other times removing seemingly trivial code can make a significant difference: for example, zero-initializing each field of a structure individually as part of its constructor does not yield the same results as using the automatically generated constructor, which zero-initializes the whole structure — including (in C++11) any structure padding. The latter can often lead to much more compact and faster code, and in general improper structure padding and alignment issues can cause significant performance trouble (we measured one instance in which explicitly preventing Clang from packing two fields together yielded a 10-fold speedup on a very simple model).

A.3 Generating Usable Code

A defining goal of Cuttlesim is to produce readable and debuggable models. For this purpose, Cuttlesim employs a few tricks:

Ad-hoc polymorphism and template metaprogramming. Most Kôika primitives are polymorphic: `+` works on bitvectors of any size, `pack` works on any datatype, `get` and `put` work on arrays of any size, etc. On the C++ side, the compiler uses operator and function

[6] The Booleans returned by rules are a debugging aid: the model does not use them.

overloading to get readable output and template metaprogramming to minimize the amount of generated code.

Macros and notations. We make minimal use of macros, as they can hurt step-by-step debugging. When we do use them, we make sure that they do not hide significant amounts of code. We make heavy use of C++11's numeric literal operator templates to define readable constants, so that in C++ models generated by Cuttlesim, `4'0110_b` and `4'6_d` are 4-bit bitvectors with equal value, while `4'11010_b` and `4'26_d` are compile-time out-of-range errors.

This enables users to debug at a much finer-grained level than complete cycles, while preserving their intuitions about Kôika's semantics.

Scoping and expressions. C++ scoping rules are slightly different from Kôika's. We use them when possible, and we attempt to minimize the number of nested scopes that are created. Concretely, if a Kôika program says `if ... then let a := ... in let b := ... in let a := ...`, we generate code similar to `if (...) { auto a = ...; auto b = ...; { auto tmp = a; auto a = ...; } }`. When expressions contain stateful subparts, we hoist these parts out into temporary variables. Because C++ debuggers are typically line-oriented, this also makes it easier to step through individual stateful subexpressions. (For a similar reason, we do not use statement expressions, a common C++ extension allowing statements such as variable declarations inside expressions.)

Tooling. Cuttlesim auto-generates a Makefile with targets for many tools that can be useful to explore our models. Concretely, we have targets for building optimized and debugging builds; running the model and tracing its execution to generate VCD files; visualizing VCD traces in GTKWave; running a debug build under GDB, LLVM; collecting and stepping through rr traces; profiling a model's execution; collecting and visualizing coverage statistics; and a number of targets that invoke the regular Kôika compiler to produce circuits, simulate them with Verilator, and compare the resulting VCD traces to those produced by Cuttlesim (discrepancies between VCD traces indicate bugs in user-supplied implementations of external functions: for self-contained models that do not use external functions, the traces will always match).

B ARTIFACT

B.1 Abstract

Kôika is a high-level hardware design language. This paper is about Cuttlesim, a simulator for Kôika designs. Typical hardware simulators like Verilator start from circuits, but Cuttlesim is different: it works by compiling Kôika hardware designs to C++ directly, through a new pipeline that does not use Kôika's circuit compiler.

Our artifact is an easy-to-run distribution of Kôika and Cuttlesim and of the hardware designs used in the evaluation section of our paper. It is packaged as a virtual machine, built using a simple setup script that pulls from the main `koika` repo.

Our artifact provides evidence to answer the three questions in the Evaluation section, including reproducing all graphs:

- Q1: Can Cuttlesim models run faster than a state-of-the-art Verilog simulator?
- Q2: Is Cuttlesim's advantage only due to Kôika's compiler generating inefficient Verilog circuits?
- Q3: How sensitive is Cuttlesim's performance to external factors, especially compiler choices and tool versions?

B.2 Artifact Checklist (Meta Information)

- **Algorithm:** Static analysis & compilation of Kôika programs
- **Program:** Custom-written Kôika & Bluespec designs
- **Compilation:** Coq >= 8.10, OCaml >= 4.07, GCC 9 & 10, Clang 10, Verilator 4
- **Run-time environment:** GNU/Linux
- **Hardware:** Typical x86-64 box
- **Metrics:** Run time
- **Output:** Performance plots
- **Experiments:** Run the benchmarking scripts
- **How much disk space required (approximately)?:** 6GB (local build + dependencies) or 12GB (VM)
- **How much time is needed to prepare workflow (approximately)?:** 1h (local build) or 2min (VM)
- **How much time is needed to complete experiments (approximately)?:** 2h
- **Publicly available?:** https://github.com/mit-plv/koika/tree/asplos2021
- **Code licenses (if publicly available)?:** GNU GPL v3
- **Archived (provide DOI)?:** 10.5281/zenodo.4342100

B.3 Description

Our artifact can be reviewed in two ways: by running everything locally on your own (GNU/Linux, x86-64) machine or using a prebuilt VM. These instructions focus on the VM approach, but you can use the same 60-line setup script to set up a local environment.

These instructions are also available in a plain-text file to make it easier to copy and paste commands: https://github.com/mit-plv/koika/blob/asplos2021/etc/ae/readme.rst.

Before proceeding, we recommend skimming through Kôika's README, which should help you get a better sense of how everything fits together. If you are curious about rule-based hardware languages, you may also want to skim through the original Kôika paper [3].

To get started, download the artifact VM on Zenodo at https://doi.org/10.5281/zenodo.4342100 and import the OVA virtual machine into VirtualBox. Start the VM and log in with username `ubuntu` (no password). On some versions of VirtualBox, booting can lead to a blank screen; in that case, resize the VirtualBox window to force a redraw.

All data and scripts are in `~/cuttlesim` in the VM. All code is public and hosted on GitHub at https://github.com/mit-plv/koika in the `asplos2021` branch.

B.4 Warming Up

While this section is optional, it will help you get a sense for what Cuttlesim does. We will look at a trivial design computing terms of the Collatz sequence, compile it, and run it with Cuttlesim and Verilator.

Navigate to `~/cuttlesim/koika/examples` and open `collatz.v` in Emacs. This is a Coq file, as Kôika is a Coq EDSL

(annoyingly, Coq files have the same extension as Verilog files), and the VM includes a preconfigured Emacs installation. It includes a definition of the design's state (`reg_t` with one register `r0` of type `bits_t 16`, initialized to the value 18) and two rules `divide` and `multiply`; comments inside the file give more details about the example.

Now run `make _objects/collatz.v/`. This will compile the example to Verilog (using the original Kôika compiler) and C++ (using our new compiler). Browse to `_objects/collatz.v/` and open `collatz.v`. This is a (very short!) Verilog file, implementing bypassing from `divide` to `multiply` as expected. Compiler optimizations make the file tiny, which is great for circuit performance but not so good for mapping each signal back to the original Kôika design for debugging. Instead, open `collatz.hpp` and skip to `DEF_RULE(divide) {`. This is the cycle-accurate C++ model generated by Cuttlesim. There is one class method per rule, plus one class method `void cycle()` which implements the scheduler. Notice how the syntax and structure of the C++ code mirrors the original Kôika syntax.

Run the following commands in `_objects/collatz.v/`:

```
make NCYCLES=150 collatz.run
make NCYCLES=150 collatz.verilator.run
```

The first one will run the *C++ model* built by Cuttlesim; the second will run the *circuit* built by Kôika using Verilator, an open-source hardware simulator. Cuttlesim prints `r0 = 16'b0000000000000100 (0x4, 4)`: the value 4 is the result of the Collatz system reaching one, then multiplying by 3 and adding 1. With 151 cycles, the output would be 2 (4 divided by 2, then multiply would not run since 2 is even). Now try both with `NCYCLES=1000000000`. Despite the heavy optimizations that made the Verilog circuit tiny, Cuttlesim is still about 4 times faster than Verilator.

You can run `make gdb` to explore the design interactively (command `b module_collatz<extfuns>::cycle()` will place a breakpoint at the beginning of each cycle), or you can run `make NCYCLES=150 collatz.hpp.gcov` and open the resulting coverage file to see that `divide` executed in every cycle whereas `multiply` failed in every other cycle (the annotation `75: 108: FAIL_FAST();` shows that the `FAIL` path was taken 75 times) -- no hardware counters needed!

Finally, let us simulate a larger design. Navigate to `examples/rv/` and run `make core` to compile a RISCV core. Then run `make verilator-tests` to run example programs and unit tests with Verilator and `make cuttlesim-tests` to run the same tests with Cuttlesim, which should be about twice as fast.

If you want to see more of Kôika, we recommend reading through the literate example in `pipeline_tutorial.v`. To see a complete list of simulation targets supported by Cuttlesim, along with documentation, run `make help` in any Cuttlesim-generated output directory (such as `_objects/collatz.v/` or `rv/_objects/rv32i.v/`).

B.5 Experiment Workflow

Each experiment consists of compiling a Kôika design to C++, running it using Cuttlesim or Verilator, and comparing the results. All experiments are conveniently packaged as a single script `etc/bench.sh`.

All dependencies in the VM are precompiled. If you want to rerun the builds (it takes ~30 minutes), run the following commands. There are four designs to compile (the RV32i and RV32e variants of our processor, plus the enhanced branch predictor variant and the multicore variant):

```
cd ~/cuttlesim/koika/examples/rv;
  make DUT=rv32i; make DUT=rv32e
cd ~/cuttlesim/koika_bthom-bp/examples/rv;
  make DUT=rv32i;
  ln -fs $(realpath _objects/rv32i.v/) \
    ~/cuttlesim/koika/examples/rv/_objects/rv32i-bp.v
cd ~/cuttlesim/koika_sim-multicore/;
cd examples/dynamic_isolation/;
  make DUT=rv32i_no_sm;
  ln -fs $(realpath _objects/rv32i_no_sm.v/) \
    ~/cuttlesim/koika/examples/rv/_objects/rv32i-mc.v
```

Then, navigate to `~/cuttlesim/koika/etc/` and run `./bench.sh` in a terminal, redirecting its output to a file (this script just runs the `make cuttlesim` and `make verilator` targets for each of the examples that this paper discusses):

```
./bench.sh 2>&1 | tee bench-results
```

Once this completes, run `./summarize.py bench-results` to generate plots.

The default script runs only one iteration of each measurement, to make sure that it completes reasonably quickly (it should take 10 to 20 minutes). Change `REPEAT=1` at the beginning of the file to `REPEAT=5` or `REPEAT=10` to improve precision (we ran it with `REPEAT=10` for the plots in this paper). In the output, a line starting with << indicates that a new test has started running, and a line starting with >> records the output of a single repeat of a given test.

B.6 Evaluation and Expected Results

Running `etc/summarize.py` will generate four plots from the results gathered in the previous step and write them to `~/cuttlesim/koika/etc/bench/`:

- `cuttlesim-verilator-cps.pdf` (Fig. 1)
- `cuttlesim-verilator-wall.pdf` (Fig. 1)
- `koika-bluespec-verilator-wall.pdf` (Fig. 2)
- `cuttlesim-verilator-wall-gcc-clang.pdf` (Fig. 3)

B.7 Exploring and Extending

The `examples/` directory of Kôika's repository contains many more examples, which can be compiled using `make _objects/example_name/`. Running `make help` in the resulting directory will offer a collection of conveniently set-up targets exposing all the tools that we commonly use, like VCD trace generation, GCOV instrumentation, GDB and LLDB debugging, performance profiling, etc.

REFERENCES

[1] 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

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh

Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] 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 Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[4] 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

[5] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. 2019. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 178–183. https://doi.org/10.1145/3289602.3293918

[6] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[7] The MyHDL community. [n.d.]. MyHDL. http://www.myhdl.org/.

[8] 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

[9] George Economakos, Petros Oikonomakos, Ioannis Panagopoulos, Ioannis Poulakis, and George K. Papakonstantinou. 2001. Behavioral synthesis with systemC. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*. 21–25. https://doi.org/10.1109/DATE.2001.914995

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

[11] Jakob Engblom. 2012. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 1–6.

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

[13] Abhijit Ghosh, Joachim Kunkel, and Stan Y. Liao. 1999. Hardware Synthesis from C/C++. In *1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany*. 387–389. https://doi.org/10.1109/DATE.1999.761152

[14] Mentor Graphics. [n.d.]. Handle-C. https://www.mentor.com/products/fpga/handel-c/.

[15] D. J. Greaves. 2000. A Verilog to C compiler. In *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668)*. 122–127. https://doi.org/10.1109/IWRSP.2000.855208

[16] 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*. 2:1–2:11. https://doi.org/10.1145/3359986.3361199

[17] 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

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[19] D. Kim, C. Celio, S. Karandikar, D. Biancolin, J. Bachrach, and K. Asanović. 2018. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 76–764. https://doi.org/10.1109/FPL.2018.00021

[20] Zainalabedin Navabi. 1997. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc.

[21] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[22] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel S. Emer. 2009. A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 2, 3 (2009), 16:1–16:26. https://doi.org/10.1145/1575774.1575775

[23] QBayLogic. [n.d.]. Clash: A modern, functional, hardware description language. https://clash-lang.org/.

[24] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[25] Jane Street. [n.d.]. Hardcaml. https://github.com/janestreet/hardcaml.

[26] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0*. https://doi.org/10.5281/zenodo.3744225

[27] Donald E. Thomas and Philip Moorby. 1996. *The Verilog hardware description language (3. ed.)*. Kluwer. https://doi.org/10.1007/978-1-4615-3992-6

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

[29] Muralidaran Vijayaraghavan and Arvind. 2009. Bounded Dataflow Networks and Latency-Insensitive circuits. In *7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), July 13-15, 2009, Cambridge, Massachusetts, USA*. IEEE, 171–180. https://doi.org/10.1109/MEMCOD.2009.5185393

[30] Stephen Williams. [n.d.]. Icarus Verilog. http://iverilog.icarus.com/.

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

[32] Matt Yourst. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *ISPASS 2007: IEEE International Symposium on Performance Analysis of Systems and Software*, 23–34. https://doi.org/10.1109/ISPASS.2007.363733