JIT Circuit Simulation with LLVM

Brennan Shacklett
Stanford University
bps@stanford.edu

Jennifer Tao
Stanford University
jenntao@stanford.edu

ABSTRACT

The project aimed to create a circuit simulator that produces JIT compiled native code through LLVM's APIs. A JIT based simulation approach provides native simulation speed, while still allowing native code to be recompiled to provide circuit debugging features unavailable in statically compiled simulators. The result of the project is a simulator that performs competitively against existing simulators in the field while providing a much larger array of interactive debugging features.

1 BACKGROUND

Circuit and logic design has long been seen as a difficult and esoteric subject to enter into, largely because of poor tooling and the slow iteration time of working on circuits. Recently, however, groups such as the Stanford Agile Hardware Center [5], have begun research on how to make developing hardware faster, easier, and in general more like developing software.

When it comes to designing hardware with fast iteration times, one of the key sticking points is the problem of Place and Route. Hardware designs can be compiled to FPGAs or CGRAs—high performance programmable hardware that quickly execute different designs—but unfortunately, fitting large designs onto these chips requires very slow Place and Route algorithms that often take hours or days.

Although running designs on real hardware is typically preferable for performance reasons, the slowness of Place and Route means that software simulation is often the first step for debugging a design. There are 2 broad categories of simulators: ones that interpret a circuit description at runtime and execute similarly to a scripting language (interpreters), and simulators that process a circuit description to produce code that, when compiled, will simulate the circuit (static simulators). Unfortunately, interpreters typically are not used outside of very small scale designs due to their very poor performance.

The industry standard in static simulation is currently the Verilator simulator [6], which reads in Verilog (the most widely used hardware description language), and produces C++ which must be compiled to produce a simulation program. Innovation in this area was lacking for many years, possibly due to the complexity of parsing and simulating Verilog, but the recent development of CoreIR[2] and other new circuit intermediate representations (IR) has caused an increase in development of new simulators.

This project focuses on simulating CoreIR-based circuits. CoreIR already has both an interpreter and a static simulator like Verilator that produces C++ code. However, the interpreter is quite slow, and neither Verilator nor the CoreIR static simulator have easy-to-use facilities to extract intermediate state or to set watchpoints in a circuit. This is the motivation for a JIT-based simulator that combines the performance of static simulators with the usefulness of interpreters.

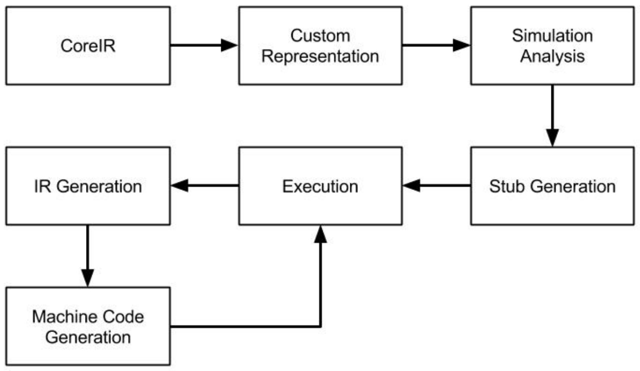

Figure 1: Simulation flow chart

This project relates to CS242's themes of Performance (for obvious reasons), and Expressiveness, because the JIT simulator aims to help circuit designers debug in a more natural and productive way than existing solutions.

2 APPROACH

2.1 Overview

The defining aspects of our project's simulator are its use of a hierarchical strategy for simulation, and its use of a JIT backend to lazily generate code. This section will first cover how a circuit is represented and analyzed, before moving on to how LLVM is used to generate and execute code. Figure 1 has a high-level overview of the simulator's operation and roughly corresponds to the layout of this section. The entire project is written in C++, since both CoreIR and LLVM's native APIs are written in C++. The code for the project can be found here [4], which is undoubtedly a more complete description of the simulator's approach than this report can be.

2.2 Circuit Representation

One of the first design decisions in our project was to have a custom representation for the circuit, rather than building the project around an existing IR. The primary reason behind this decision was to reduce the complexity and special case handling needed in the simulation analysis and code generation phases. In the authors' experiences, most of the complexity in writing a simulator comes from handling various edge cases within the IR, and when these edge cases spread through the whole project, debugging quickly becomes a nightmare.

In particular, CoreIR is a very general IR by design, so it can be used in many ways that are simply not useful to the simulator. This means the relationship between 2 pieces of a circuit may be obscured through several layers of indirection which leads to inefficiencies during code generation. Another pain point comes from handling array types. In CoreIR, a single wire value can hold a single bit of information, which is represented by the `Bit` class. This means a circuit that operates on 32-bit integers instead operates on `Arrays` of 32 `Bits` from CoreIR's perspective. This becomes particularly complicated when different bits in a given array refer to separate values in a circuit. For example an array of 32 `Bits` could have the upper 16 `Bits` connected to the output of one adder, and the lower 16 `Bits` connected to the output of another adder. The simplest way to solve this issue in a simulator is just to simulate on a `Bit`-by-`Bit` or single wire level, but this is very detrimental to performance since you cannot take advantage of the 64 bits in a native machine word.

Therefore our project's direct use of CoreIR is confined to loading in the CoreIR representation, and constructing a new, much simpler, and more restrictive representation from it. One of the key limitations of the simulator's representation is that it does not support nested types. CoreIR supports the concept of arrays of arrays, but there is no natural relation from this concept to machine words, so we use CoreIR's `flattentypes` pass to flatten arrays of arrays in the design into multiple single-level arrays. The simulator's representation is organized as follows:

At the top level is the `Definition` class. This corresponds to a module (in CoreIR terms) or class (in Magma terms) in the original circuit design. A definition can be instantiated to an `Instance`, which represents an actual instance of the module that exists somewhere in the design. A `Definition` is either a primitive, meaning it holds no instances of modules within it but has some externally defined behavior, or a list of `Instances` that are connected together and define the `Definition`'s behavior. The semantics of the behavior of various primitives is designed to match the corresponding CoreIR primitives. For example, CoreIR contains an "add" primitive that adds 2 N-bit numbers together to produce an N-bit result, a "reg" primitive that stores an N-bit value, etc. Both `Definitions` and `Instances` contain an `IFace` (interface) that defines the inputs and outputs for the attached `Definition` or `Instance`. These components of an `IFace` are grouped into `Sources` and `Sinks`. For `Instances`, the inputs are the `Sinks` and the outputs are the `Sources`; for `Definitions`, the opposite is true (while this may seem counterintuitive, remember that within the context of a `Definition`, the `Definition`'s inputs are the original `Sources` of all the values used by the `Definition`'s `Instances`, and the `Definition`'s outputs are the `Sinks` that accept the values generated by the `Instances`).

`Sources` and `Sinks` both store their bitwidth as well as a name for debugging purposes. `Sinks` also hold a list of slices of `Sources` that encodes the bits that drive a given `Sink`. For example, a 4-bit `Sink`'s first 2 bits could be driven by the 4th and 5th bits of a 6-bit input to a `Definition`, and the last 2 bits could be driven by the 2nd and 3rd bits of an `Instance`'s 4-bit output. As an optimization, when connecting a `Sink` to a list of `Sources`, the representation groups together adjacent bits. For example, Magma sometimes has a tendency to connect values `Bit` by `Bit`, when in reality a `Sink` could be a direct copy of a `Source`, instead of N 1 bit slices of the `Source`. This optimization improves the performance and clarity of the generated code.

Memory management in the representation is currently a bit fragile; currently, `Definitions` "own" all the memory within them, and there are numerous pointers shared around to link together connected values and instances. The only real issue with this approach is the danger of pointer invalidation if and when values are accidentally moved due to containers being extended. Possible future approaches include a `Context` object that owns all allocated data and then hands out unique IDs that can be used to retrieve values (or perhaps raw pointers for performance).

2.3 Simulation Analysis

After the circuit representation is constructed from CoreIR's representation, the simulator analyzes the circuit and stores information that is necessary for code generation. Before those details are presented, we will first explore the difference between hierarchical and the traditional "flat" simulation strategies. Note that in all the simulation analysis discussion, it is assumed that a circuit only has 1 clock. This is not a realistic assumption for a production-grade simulator, but currently no existing CoreIR-based simulator handles more than 1 clock and the semantics of multiple clocks are poorly defined in CoreIR itself, since no design exists for CoreIR that uses more than one clock. The design of this simulator does not preclude eventually supporting multiple clocks in the future, but this feature is out of the scope of our project.

In a traditional simulator (static or interpreter-based), all the hierarchy in a circuit design is removed by flattening the entire circuit into a single module, which only contains instances of primitives. This process is akin to inlining all of a program's code into a single function. The reasoning behind this approach is primarily simplicity: the behavior of a given primitive in the flattened module is much easier to reason about than the behavior of an instance of a module that may include other modules within it, particularly when modules contain state within them. Any given flattened circuit can be reasoned about as a collection of state, combined with some set of combinational logic that performs a function on the state. Each time the clock is toggled, the collection of state is updated and a new output is computed by the combinational logic. Simulators that use flattened designs typically separate out the state and the combinational logic, and then order the combinational logic so it can be correctly and efficiently executed in serial on a CPU.

Conversely, this simulator uses a hierarchical strategy where, instead of flattening, each `Definition` is analyzed to produce a `compute_output` and an `update_state` function that computes the output based on the current state and inputs, and updates the state based on the current inputs, respectively (credits to Ross Daly for outlining the idea of compute output and update state at the beginning of the quarter). These functions are composed from one another: this means the top-level definition's `update_state` function will call the `update_state` functions of every instance within itself, and so on. For example, consider the circuit in Figure 3. The `update_state` function for this simple circuit will call the `update_state` function of the Counter in Figure 2, which will call the `update_state` function of the Register, which is a primitive (notice that the Add primitives do not have `update_state` called on them because they hold no state).

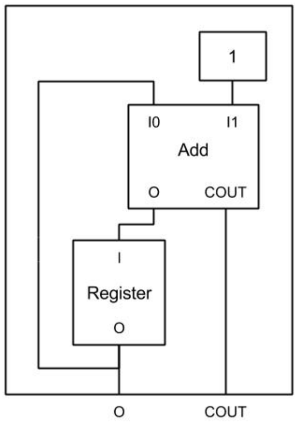

Figure 2: The Magma Counter Circuit.

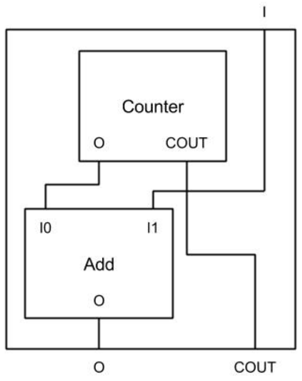

Figure 3: A Simple Circuit.

Now that the concepts of `compute_output` and `update_state` have been defined, the simulator's analysis is left with the problem of how to order the execution of instances in `compute_output` and `update_state` so that no instance is executed before any of the instances it depends on. For example, the circuit in Figure 3 relies on the Counter's output before the Adder's `compute_output` function can be executed. This example circuit's Counter also brings up a problem with the original definition of `compute_output` and `update_state`. At the start of the project, `compute_output` and `update_state` were defined to expect all of their inputs to be available at the time of their execution. Unfortunately, closer examination of the Counter in Figure 2 shows this is impossible: calling `compute_output` on the Register would require `compute_output` to be called on the Adder first, but the Adder's output depends on the Register's output. The solution to this problem is fairly obvious: the semantics of a Register indicate that its output is not dependent on its input, because the input is only used to update the internal state on the clock's rising edge. Therefore the Register's `compute_output` does not require the Register's input to be available, and the correct order is for the Register to be executed before the Adder. This means each definition needs to have 2 additional pieces of information in addition to `compute_output` and `update_state`: the subset (not necessarily proper) of the inputs required for `compute_output` and `update_state` respectively.

The simulator computes all this information as follows (note that `Definitions` are analyzed in a bottom-up order, meaning when analyzing a given `Definition`, the `Definitions` of all its `Instances` have already been analyzed): first, the simulator gathers all stateful instances and places them in a list so they can be called by the `update_state` function; then, for each of these stateful instances, the simulator does a depth-first traversal of all the instances whose outputs the stateful instance depends on for updating state (this is where the list of inputs required for `update_state` is used). For example, the Counter circuit's only stateful instance is the Register, and the Register's output depends on the Adder's output, which depends on the Register's output, so (somewhat confusingly) the Register's dependencies for updating its state are itself and the Adders' output. These instances are then gathered into a list of "stateful dependencies", and are topologically sorted, so that no instance will ever be called before its inputs are ready. The `update_state` function can then operate by calling `compute_output` on each of the stateful dependencies, followed by calling `update_state` for each stateful instance. Additionally, if any of the stateful instances trace back to one of the `Definition`'s inputs, that input is marked as required for `update_state`.

The `compute_output` information is calculated in a very similar way: instead of starting with the stateful instances and tracing back, the `Definition`'s sinks are traced back with a depth-first traversal and every instance is added to a list of "output dependencies," which are then topologically sorted. Any inputs of the definition that are traced to from the outputs are marked as required for `compute_output`.

This somewhat complex process raises a question: why is hierarchical simulation useful? There are several reasons. First, breaking the simulation up into multiple functions is useful for the recompilation required by debugging, since much smaller parts of the code would then need to be recompiled. Second, retaining the hierarchy of the original design is useful for debugging, since a potentially lossy and very complicated mapping from components in the original design to components in the flattened design would otherwise be required to find intermediate values. Additionally, although all these function calls may at first appear to have a large overhead compared to a flattened approach, the core simulation functions can potentially be inlined, leaving only debugging functions to retain the hierarchy, and drawing upon the best of both worlds.

2.4 Compiler Stub Generation

After circuit analysis is finished, instead of generating the IR up front, LLVM's callback manager and indirect stubs manager are used to create stubs for each `compute_output`, `update_state` and debug function. These stubs pause execution when called, jump into a compile callback that generates IR, and then update the stub to point to the machine code after the IR has been compiled. This introduces some slight overhead since any function call comes with the additional level of indirection of these stubs, but the additional overhead could be fixed by using LLVM's patch sites feature. This would allow any call to the stub to be updated with a direct call after the stub has been compiled; however, this feature is currently experimental in LLVM, so it was passed over.

The motivation for lazily generating code and IR like this is to avoid paying a cost for features of the simulator that are not used. For example, a user of the simulator only interested in the state of the circuit can simply repeatedly call `update_state` and examine the saved state, and if users do not require debugging features then no time will be spent compiling the debug functions until they are called.

2.5 LLVM IR Generation

Compared to simulation analysis, generating the actual IR is relatively straightforward. `compute_output` is generated for each definition by simply iterating through the sorted list of output dependencies calculated during analysis, and calling the `compute_output` function for each instance, unless it is a primitive. If a given instance is a primitive, then code generation calls a predefined lambda function which outputs the IR for a given primitive, for example, this is the IR generation code for the `compute_output` "function" of CoreIR's mux primitive:

```
[](auto &env, auto &args, auto &inst)
{
  llvm::Value *lhs = args[0];
  llvm::Value *rhs = args[1];
  llvm::Value *sel = args[2];

  llvm::Value *if_cond =
    env.getIRBuilder().CreateICmpEQ(sel,
                                    llvm::ConstantInt::
     get(env.getContext(), llvm::APInt(1, 0)),
                                    "ifcond");

  llvm::Value *result =
    env.getIRBuilder().CreateSelect(if_cond, lhs, rhs, "
     result");

  return std::vector<llvm::Value *> { result };
}
```

Notice that this does not actually generate a function that performs the `compute_output` operation; it simply inserts instructions directly into the current `Definition`'s basic block. This was done to improve performance, since a fully flattened design will be fully inlined already with this approach, and also to improve usability. One goal was to make sure adding primitives would be as quick and easy as possible, so we wanted someone implementing primitives to write as little boilerplate as possible. If a primitive does need to call a function, it can still do so, and there is a separate infrastructure for the primitive to define an LLVM module with helper functions it can call out to.

After the `compute_output` IR has been generated for each instance within a `Definition`, the final outputs of the definition are packed together into an LLVM struct and returned. This is to support `Definitions` with multiple outputs, since LLVM functions cannot return multiple values directly. There seems to be very little overhead from packing values into structs: based on examination of the generated assembly, LLVM often removes the struct entirely and just returns multiple results in multiple registers.

The `update_state` function is generated in a very similar manner, except instead of iterating through the list of output dependencies, the list of stateful dependencies is used, and instead of packing the outputs in a struct, the `update_state` IR for each instance is produced at the end of the function.

One subtle aspect of code generation is how the storage of state is handled. The number of bytes of state required for each definition is recursively calculated based on their component instances, which ultimately determines the number of bytes of state required for the top-level definition (and therefore the whole circuit). This is then allocated in one buffer, which is passed into the `compute_output` and `update_state` functions. Each instance in a definition is given an offset relative to the start of the definition. For example, in a `Definition` that contained two 4-byte registers, one register would be given an offset of 0 and one would be given an offset of 4. Therefore, before calling `compute_output` or `update_state`, the current state pointer would be incremented by the instance's offset value, which ensures that no instance overwrites or reads the state of another instance. This approach guarantees that the state can be passed through the hierarchy as a single pointer to ensure minimal overhead. In a totally flattened design, each instance could be assigned a fixed address to store state at, which would likely be somewhat more efficient. However, the overhead of incrementing the pointer has been low enough thus far to not require exploring that optimization, especially since it only works on flattened circuits.

In order to simplify generation of IR, each LLVM function is wrapped in a `FunctionEnvironment` class that stores the currently declared variables and functions, which prevents duplicate declarations of external functions in a given module and allows values to be easily accessed. The `FunctionEnvironment` stores a map from `Source` to `llvm::Value`, so the values that a given `Sink` depends on can quickly be retrieved and passed as arguments to a given instance.

2.6 LLVM JIT Execution

The actual JIT compilation and execution uses LLVM's ORC APIs, based on the tutorial found here [3]. When ready to be compiled, LLVM IR is passed to a transform layer that performs optimizations (currently just the default set of -O2 optimizations). This optimized IR is then passed to the `SimpleCompiler` layer that generates native machine code, which then passes the IR to the runtime linking layer. This layer uses a dynamic symbol resolver to first look inside the JIT's compiled code for a symbol, and then calls an external lambda that simply looks through the host processes' symbols. This allows JITed code to call back to precompiled functions in the host process, for debugging or perhaps profiling purposes.

2.7 JIT Debug Support

Previous sections briefly alluded to generating "debug functions" alongside `update_state` and `compute_output`. The debug version of `update_state` is currently called `state_deps`, and performs the same `compute_output` calls on the stateful dependencies as `update_state`, but never actually calls `update_state`; so it essentially computes all the outputs `update_state` would have, without actually changing the state. Additionally, the `state_deps` function passes an instance id through the call stack, which tracks the current instance of the definition that is currently being executed. This is important since it allows the debugging code to differentiate between different calls to the same definition's `state_deps` function. The debug version of `compute_output` is called `output_deps` and is identical except for the introduction of the instance id parameter like in `state_deps`.

Neither of these functions is particularly useful unmodified, but together they allow access to every single intermediate value in the circuit (this is why the `state_deps` function is necessary, because otherwise it would be impossible to access intermediate values from a definition's state dependencies without updating state at the same time). After the IR for these functions is initially generated through the stub callbacks, the IR is saved for the duration of simulation to allow them to be repeatedly modified without needing to regenerate the initial IR over and over. To support this feature, an interface to remove code from the JIT was added, which simply rewrites the stub from pointing to the now-removed compiled code to instead point to a new compile callback. Additionally, before these functions are modified, the modules are cloned to preserve the original copy. The performance impact of this clone operation versus simply regenerating the whole function from scratch each time is likely worth investigating in the future.

The JIT frontend provides a `DebugInfo` interface to these functions, which allows clients to request specific intermediate values be extracted from a given instance's `state_deps` or `output_deps` function. After the `DebugInfo` is updated, the debug functions are regenerated to extract the given intermediate value with the given instance ID. Here is an example of a modified `output_deps` function for the circuit in Figure 3. This code extracts the output of the counter before the adder has incremented it by the input value:

```
define %global_simple_output_type.1
     @global_simple_output_deps(i4 %self.I, i8*
     %state_ptr, i64 %inst_offset) {
entry:
  %0 = add i64 %inst_offset, 0
  %inst0_output = call %global_Counter4_output_type
     @global_Counter4_output_deps(i8* %state_ptr, i64 %0)
  %inst0_COUT = extractvalue %global_Counter4_output_type
      %inst0_output, 0
  %inst0_O = extractvalue %global_Counter4_output_type
     %inst0_output, 1
  %1 = add i64 %inst_offset, 1
  %inst1_output = call %global_Add4_output_type
     @global_Add4_output_deps(i4 %inst0_O, i4 %self.I,
     i64 %1)
  %inst1_O = extractvalue %global_Add4_output_type
     %inst1_output, 0
  br label %debug_block

return:                                              ; preds
      = %inst_match, %debug_block
  %2 = insertvalue %global_simple_output_type.1 undef, i1
      %inst0_COUT, 0
  %3 = insertvalue %global_simple_output_type.1 %2, i4
     %inst1_O, 1
  ret %global_simple_output_type.1 %3

debug_block:                                         ; preds
      = %entry
  %4 = icmp eq i64 %inst_offset, 0
  br i1 %4, label %inst_match, label %return

inst_match:                                          ; preds
      = %debug_block
  store i4 %inst0_O, i4* inttoptr (i64 33075376 to i4*)
  br label %return
}
```

The debug passes operate by rewriting all jumps to the final return block of the function to instead jump to the debug block, which then compares the current instance id to the desired instance id; if they are equal,the debug block stores the given value in a fixed memory location determined by the client. This approach allows arbitrarily many of these debug blocks to be chained together, with the only overhead being the instance id comparison and a single store. The debug passes also support watchpoints with this strategy by simply adding an additional conditional to the debug block to only store a value if a condition is true.

Overall these debug functions allow intermediate state to be extracted without modifying or slowing down the core `update_state` and `compute_output` functions. For simply examining intermediate state, there is essentially no downside to this approach; however there is considerable overhead introduced for watchpoints, since both debug functions need to be called each time after the state is updated. Ideally, these changes could be integrated into a debug version of `update_state` that checked watchpoints and updated state, but there was not enough time to test that approach.

3 RESULTS

The utility of the JIT simulator was evaluated based on the quality of the generated IR, the performance of the generated code, and most importantly, the JIT's ability to facilitate debugging.

3.1 Generated LLVM IR

One of the goals of our project was to generate high quality LLVM IR to facilitate debugging. This means the IR should ideally be human-readable, so that a circuit designer can input their design into the JIT and immediately see a view of the straight-line code the simulator generates. Ideally, this would allow circuit designers to quickly find ambiguities and errors in their design exposed by the simulator's analysis of the circuit. The following is the generated IR for the `compute_output` function of a linebuffer memory included from CoreIR's standard library:

```
define %commonlib_LinebufferMem_output_type
     @commonlib_LinebufferMem_compute_output(i8*
     %state_ptr, i64 %inst_offset) {
entry:
  %0 = getelementptr inbounds i8, i8* %state_ptr, i64 20
  %1 = add i64 %inst_offset, 3
  %raddr_output = call %mantle_reg_output_type
     @mantle_reg_compute_output(i8* %0, i64 %1)
  %raddr_out = extractvalue %mantle_reg_output_type
     %raddr_output, 0
  %valid_cond = icmp ult i4 %raddr_out, 1
  br i1 %valid_cond, label %then, label %else

then:                               ; preds = %entry
  %2 = bitcast i8* %state_ptr to i16*
  %3 = zext i4 %raddr_out to i64
  %addr = getelementptr inbounds i16, i16* %2, i64 %3
  %rdata = load i16, i16* %addr
  br label %merge

else:                               ; preds = %entry
  br label %merge

merge:                              ; preds = %else, %then
  %mem_rdata = phi i16 [ %rdata, %then ], [ 0, %else ]
  %4 = getelementptr inbounds i8, i8* %state_ptr, i64 21
  %5 = add i64 %inst_offset, 7
  %waddr_output = call %mantle_reg_output_type
     @mantle_reg_compute_output(i8* %4, i64 %5)
  %waddr_out = extractvalue %mantle_reg_output_type
     %waddr_output, 0
  %veq_out = icmp ne i4 %raddr_out, %waddr_out
  br label %return

return:                                    ; preds = %merge
  %6 = insertvalue %commonlib_LinebufferMem_output_type
     undef, i16 %mem_rdata, 0
  %7 = insertvalue %commonlib_LinebufferMem_output_type
     %6, i1 %veq_out, 1
  ret %commonlib_LinebufferMem_output_type %7
}
```

This example highlights both the strengths and weaknesses of our current IR generation. Since LLVM allows naming of registers, every output of an instance is named with the convention "instance name_port name", such as "waddr_out" or "veq_out." The call instructions also show the hierarchy of the design at play, because they make it very clear when an instance is being evaluated. Unfortunately, there is also a good deal of baggage associated with function calls, namely the "extractvalue" and "insertvalue," instructions that are necessary to pack and unpack multiple return values from functions. Currently these clutter the overall flow of the IR significantly, so ideally this could be better hidden without an impact on performance. Another minor weakness of LLVM IR is that a given value in SSA form can only have 1 name, which means it is not possible to name instance inputs (since that would involve renaming the output). Ultimately, the current LLVM IR produces a reasonably useful view of the execution of a given part of the design, but some clutter and limitations of the current calling conventions mean there is still more work to be done here.

3.2 Performance

The JIT simulator was evaluated on 3 main performance metrics: the startup time of the JIT, the performance-per-cycle of the JIT after all code has been compiled, and the memory overhead introduced by the JIT. Most of these results are based on running various simulators on the CoreIR Harris circuit, which implements a version of the Harris corner detection algorithm [1]. This is the largest currently available real-world circuit in CoreIR that the authors are aware of, so while larger and slower circuits would have been better for some of the performance analysis, this was not possible due to the relative immaturity of CoreIR. All the performance tests were conducted on Brennan's desktop running Gentoo Linux with a Intel 5820k processor.

3.2.1 Startup Time. While much of the initial design of the JIT simulator revolved around making sure that code generation was as fast as possible to reduce startup time, we found that in real-world testing on the circuits currently available in CoreIR, startup time was fast enough to never be a practical issue that impeded usage of the JIT. In our testing on the Harris circuit, startup time with optimizations enabled on the project and in LLVM was measured in the 0.25 - 0.28s range over 100,000 tests. This includes the time to load the circuit from CoreIR, perform graph analysis on the circuit, and generate all native IR and machine code upfront before execution. The CoreIR interpreter has a much faster startup time in the sub 0.1s range, since it only needs to perform graph analysis of the circuit. Both of these times are low enough to be totally insignificant for debugging or normal usage, especially since the JIT can dynamically change the produced code at runtime rather than requiring repeated restarts like a traditional simulator would. In comparison, Verilator takes on average 10 seconds to produce C++ for the Harris test, and then additional time is needed to compile and link that file (which is dependent on compiler choice and optimization levels).

Table 1: Comparison of Performance Per Cycle

Simulator	ms per cycle
JIT Simulator	0.00007331 ms
CoreIR Simulator	0.00006872 ms
CoreIR Interpreter	2.124 ms
Verilator	0.00006324 ms

3.2.2 Performance Per Cycle. Performance per cycle was compared between the JIT Simulator, the CoreIR Simulator, the CoreIR Interpreter, and Verilator. The tests were conducted by running the Harris circuit over 1 million cycles on each simulator and computing the average ms per cycle. The results are shown in Table 1.

The main takeaway from these results is that on the Harris circuit, the JIT simulator is within less than 10 percent of the speed of Verilator and the CoreIR Simulator, and the interpreter is several orders of magnitude slower than all the other options. One side note is that the CoreIR interpreter's extremely poor performance is not necessarily indicative of the poor performance of all interpreted strategies. In particular, the current CoreIR interpreter performs particularly poorly on the large amount of arithmetic computations in the Harris circuit because it performs all math on arbitrary width bit vectors, which currently are not particularly well optimized. Regardless, from a performance perspective, these results show that the JIT Simulator is able to provide all the key features of the interpreter for debugging with a fairly minimal impact to overall performance when compared to the traditional approaches. In fact, the native machine code produced by running Clang on the C++ file produced by Verilator is very similar to the native machine code produced by the JIT.

The main reason for the JIT's performance penalty compared to the statically compiled simulators is the interface used to call into the JITed code. In order to allow arbitrarily many arbitrary-width inputs to be passed into the JITed code, a wrapper is generated by the JIT that accepts a struct and then loads inputs from that struct into registers that are then passed into the actual simulation code. According to performance tests done in Valgrind, this extra step of needing to load the inputs from a struct rather than passing the inputs directly as can be done in the generated C++ code has a nontrivial performance impact. One possible solution to this problem is to allow getting a raw pointer to the simulation code that can be called directly by a user of the JIT if they know the input and outputs of the circuit they are working on beforehand. Unfortunately, this would result in a loss of flexibility since the user would need to recompile their frontend whenever they changed the interface to the top level circuit.

Ultimately, it seems plausible that the JIT could feasibly produce faster machine code than existing C++ based simulators, since it is capable of generating code at a lower level. In particular, LLVM's support for arbitrary integer widths could be leveraged in optimization passes that converted those large integers to SIMD operations where possible, with much less analysis than would be required on the C++ generated by Verilator or the CoreIR simulator.

3.2.3 Memory Overhead. The memory overhead of the JIT simulator is primarily important when compared with the CoreIR circuit interpreter. Much like the JIT, the CoreIR Interpreter allows any intermediate value in the circuit to be examined and watched for changes. However, it achieves this by simply saving all intermediate state in the entire circuit during the evaluation of each cycle. This works fine for small- to medium-sized circuits; however, it quickly breaks down for large circuits with thousands or millions of intermediate values. The JIT works around this problem as discussed earlier, by recompiling circuits to only save values of interest. Unfortunately, the JIT is not always strictly better than the interpreter in terms of memory usage, due to the overhead introduced by storing LLVM IR and native machine code.

The memory overhead was evaluated in several different ways: first the memory usage of the interpreter and the JIT simulator were compared before execution of the circuit. This was designed to measure the overhead of the circuit representations, without regard for the overhead of generated code by the JIT. We found that for most circuits, the JIT's circuit representation had a much lower overhead than the CoreIR interpreter's circuit representation, usually on the order of 25 percent. For example, the CoreIR interpreter used 4MB of ram to load in the Harris circuit while the JIT simulator used 1MB. This was to be expected, because the JIT simulator loads in the CoreIR representation, translates it to its own much simpler representation, and then frees the CoreIR representation before simulation begins.

The previous test was repeated after additionally having the JIT simulator generate IR and native machine code. As discussed in the Approach section, the JIT retains the IR version of the debug functions, so the overhead of this step is significant. This placed the JIT simulator at 25MB of memory used on the Harris example while the CoreIR interpreter of course remained at 4MB. As an additional test, we decided to not include the size of debug functions, which brought the JIT's memory usage down to around 3MB, showing the cost of saving LLVM IR was much more significant than expected. Even so, this is a relatively minor issue with the JIT, since the faster debugging provided by saving IR outweighs the fixed memory overhead in many applications. Additionally, a user of the JIT only pays the memory cost for the debugging features once those features are actually used, so if the user is only executing the circuit normally, the debug IR is never generated.

Finally, the memory usage after 1000 cycles of execution on the Harris circuit was compared (note that for this test the interpreter's rewind feature was disabled since it currently saves the state after every cycle). The JIT simulator used approximately 50MB of memory, due to additional space needed to store the state for the circuit, which is not allocated until execution. The interpreter's memory usage increased substantially up to 125MB, because every intermediate value needed to be saved as a dynamic width bit vector, which contains significant overhead. As an extension of this test, we expanded the Harris circuit to operate on 32-bit values instead of 16-bit values, and found that the JIT simulator actually reduced its memory usage slightly down to about 45 MBs (due to fewer truncation instructions needed in the generated code), and the interpreter's memory requirements more than doubled up to approximately 275 MB, since the size of all the intermediate states doubled.

Of course, strong conclusions cannot be drawn from these results without a much wider set of test circuits and machine configurations, as well as more precise measurements of memory usage. However, these results still seem to indicate that the JIT simulator has a significantly more scalable strategy than the interpreter in terms of memory usage.

3.3 Debugging

In order to show that the performance of the JIT simulator is beneficial, and even necessary, to support debugging a wide class of circuits, we created an example circuit in Magma, as shown below:

```
def DefineBWClassifier(width, height):
  class BWClassifier(Circuit):
    name = 'example'
    IO = ["R", In(Bits(8)), "G", In(Bits(8)), "B", In(
     Bits(8)), "F", Out(Bit), "O", Out(Bit), "CLK", In(
     Clock)]
    @classmethod
    def definition(circuit):
      counterbits = int(log2(width*height))
      conv = ToGrayScale()
      wire(circuit.R, conv.R)
      wire(circuit.G, conv.G)
      wire(circuit.B, conv.B)
      compare = ULE(8)

      # compare returns 1 if the grayscale value is below
      128 (in other words it is more black)
      wire(conv.O, compare.I0)
      wire(array(128, 8), compare.I1)
      n = Not()
      wire(compare.O, n.interface.ports['in'])

      white_counter = Counter(counterbits, has_ce=True)
      black_counter = Counter(counterbits, has_ce=True)

      wire(n.out, white_counter.CE)
      wire(compare.O, black_counter.CE)

      blackvwhite = ULE(counterbits)
      wire(white_counter.O, blackvwhite.I0)
      wire(black_counter.O, blackvwhite.I1)

      # Circuit Outputs 1 if more white than black
      wire(blackvwhite.O, circuit.O)

      total = Add(counterbits)
      wire(white_counter.O, total.I0)
      wire(black_counter.O, total.I1)
      totalcmp = EQ(counterbits)
      wire(array(width*height, counterbits), totalcmp.I0)
      wire(total.O, totalcmp.I1)
      wire(totalcmp, circuit.F)

    return BWClassifier
```

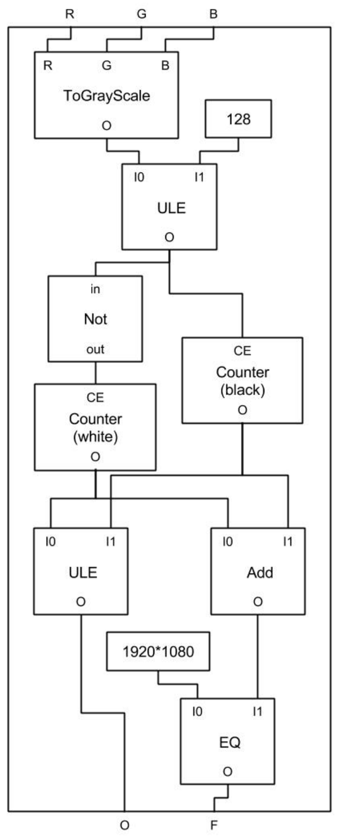

Figure 4: A diagram of the Black/White Classifier Circuit.

A graphical depiction of the circuit is also provided in Figure 4. During each cycle, this circuit takes the 8-bit RBG values of a pixel in an image, and converts this to a single 8-bit grayscale value (the code for `ToGrayScale` is omitted—it simply performs fixed point arithmetic to multiply the R, G and B values by some fraction and combines the result). After the grayscale value has been computed, it is compared against the midpoint value 128 to determine if the given pixel is more white or black, at which point the corresponding counter is enabled and the counter for black or white will increase by one. The running totals of each counter are continuously compared, and when every pixel has been counted, the F or "finished" signal becomes true, and the consumer of this circuit can use the value, perhaps as a feature in some larger machine learning project.

Unfortunately, the magma code for this circuit has a subtle but very significant error. The circuit itself is parameterized by the width and height of the input image, so the circuit knows how many bits to allocate to the counter to be able to count all the way up to the total resolution of the image (in case the entire image is determined to be black or white). However, the programmer for this circuit forgot to use the ceiling of the log base 2 rather than just the log base 2, so for any image whose total number of pixels is not a power of 2, this circuit will ignore the end of the image. As an aside, it seems plausible that magma should have warned the programmer when trying to fit the constant number of pixels into too small of an array, but unfortunately that is currently not the case and the above circuit will compile without error or warning.

Imagine a scenario where a designer is creating a circuit that processes 1920x1080 images. $1920 \cdot 1080$ is not a multiple of 2, so the error will be manifested in the designer's circuit. Let's assume that the designer notices something is wrong with the black white classifier and begins to debug. Assuming the designer did not have access to the JIT simulator, his debugging process would likely go as follows: he could attempt to run the circuit under the CoreIR Interpreter, and set a watchpoint for when the F signal of the black / white classifier turned to 1. When this watchpoint was triggered, the interpreter would allow him to examine the internal state of the classifier when it believes it has finished processing the circuit, and he could likely see that the sum of the counters is much less than the total size of the image, and that the $1920 \cdot 1080$ constant had been truncated. Unfortunately in order for this watchpoint to trigger, the interpreter would have to advance through approximately 1 million cycles, which on this circuit, even with a very simplified implementation of ToGrayScale takes around 40 minutes. Assuming the designer gave up on using the interpreter on his design due to its slow performance, he has 2 approaches: he could try reducing the size of the image, so the interpreter could operate on a smaller example. Unfortunately this could require a large number of changes across the design, especially in the circuit that loads the image and passes it to the classifier. Alternatively the designer could opt to build debugging assertions into his design using Verilator's debugging features; however, each new assertion requires recompiling the circuit, and introducing code into his design that is Verilator specific.

The JIT simulator makes this problem much simpler to solve. The designer can set a watchpoint on the F signal of the classifier, and the JIT will recompile the appropriate function to write a boolean out to some external address when the F signal becomes true. Each cycle the JIT just checks if this address contains true, and if so the simulator stops cycling an notifies the designer. At this point the designer can query any intermediate variables and the JIT will automatically recompile the correct output_deps function to extract the desired values. All these recompilations happen in less than half a second, so the designer is able to debug the circuit in a very interactive manner, allowing a similar experience to software debugging through GDB.

3.4 Conclusions

These results show that the JIT Simulator is able to provide debugging features that can be used to find and fix issues in real circuits, without needing to make significant concessions for usability or performance. In particular, the JIT simulator is able to provide all the features an interpreter based simulator would, but without the traditional performance cost. Additionally, the JIT simulator is capable of better interoperability with the existing statically compiled simulators, since ultimately they all produce machine code that uses the C calling conventions. The flexibility and performance of the JIT simulator will hopefully make it a useful tool in the future of agile hardware development.

ACKNOWLEDGMENTS

The authors would like to acknowledge Ross Daly and Leonard Truong for their work on the CoreIR circuit representation, Dillon Huff for his work on the CoreIR interpreter, and Professor Pat Hanrahan for his work on Magma and CoreIR.

WORK BY PROJECT MEMBERS

Equal work was performed by both project members.

REFERENCES

[1] CoreIR 2017. CoreIR example programs. (2017). Retrieved Dec 14, 2017 from https://github.com/StanfordAHA/CGRAMapper/tree/master/examples

[2] CoreIR 2017. CoreIR source code. (2017). Retrieved Dec 14, 2017 from https://github.com/rdaly525/coreir

[3] JIT 2017. Building a JIT in LLVM. (2017). Retrieved Dec 14, 2017 from https://llvm.org/docs/tutorial/BuildingAJIT1.html

[4] JITSim 2017. JIT Circuit Simulation with LLVM. (2017). Retrieved Dec 14, 2017 from https://github.com/shacklettbp/jitsim

[5] Stanford AHA 2017. ISTC/AHA! Intel Science and Technology Center for Agile Hardware. (2017). Retrieved Dec 14, 2017 from https://aha.stanford.edu/

[6] Verilator 2017. Introduction to Verilator. (2017). Retrieved Dec 14, 2017 from https://www.veripool.org/wiki/verilator