Just-in-Time Compilation for Verilog

A New Technique for Improving the FPGA Programming Experience

Eric Schkufza
VMware Research
Palo Alto, CA
eschkufza@vmware.com

Michael Wei
VMware Research
Palo Alto, CA
mwei@vmware.com

Christopher J. Rossbach
UT Austin and VMware Research
Austin, TX
rossbach@cs.utexas.edu

Abstract

FPGAs offer compelling acceleration opportunities for modern applications. However compilation for FPGAs is painfully slow, potentially requiring hours or longer. We approach this problem with a solution from the software domain: the use of a JIT. Code is executed immediately in a software simulator, and compilation is performed in the background. When finished, the code is moved into hardware, and from the user's perspective it simply gets faster. We have embodied these ideas in Cascade: the first JIT compiler for Verilog. Cascade reduces the time between initiating compilation and running code to less than a second, and enables generic printf debugging from hardware. Cascade preserves program performance to within 3× in a debugging environment, and has minimal effect on a finalized design. Crucially, these properties hold even for programs that perform side effects on connected IO devices. A user study demonstrates the value to experts and non-experts alike: Cascade encourages more frequent compilation, and reduces the time to produce working hardware designs.

CCS Concepts • **Hardware → Reconfigurable logic and FPGAs**; • **Software and its engineering → Just-in-time compilers**.

Keywords Cascade, Just-in-Time, JIT, Compiler, FPGA, Verilog

ACM Reference Format:
Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-in-Time Compilation for Verilog A New Technique for Improving the FPGA Programming Experience. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3297858.3304010

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304010

1 Introduction

Reprogrammable hardware (FPGAs) offer compelling acceleration opportunities for high-performance applications across a wide variety of domains [19, 20, 27, 41, 42, 44, 46, 57, 60, 66, 82, 85, 90]. FPGAs can exceed the performance of general-purpose CPUs by several orders of magnitude [21, 71] and offer dramatically lower cost and time to market than ASICs. In coming years, FPGA density and clock rates are projected to grow steadily while manufacturing costs decline. As a result, hardware vendors have announced plans for server-class processors with on-die FPGA fabric [10], and cloud providers have begun to roll out support for virtual machines with FPGA accelerators and application development frameworks [29].

While the benefits are substantial, so too are the costs. Programming an FPGA is a difficult task. Writing code in a *hardware description language* (HDL) requires a substantially different mental model than it does for a Von Neumann architecture. The rapid growth of Domain Specific Languages with HDL backends [16, 24, 56] has begun to address this issue. But regardless of frontend language, HDL must ultimately be compiled to an *executable bitstream* format which can be consumed by an FPGA. The key open problem we address in this paper is that this is an extremely slow process. Trivial programs can take several minutes to compile, and complex designs can take hours or longer.

We believe that compiler overhead is a serious obstacle to unlocking the potential of FPGAs as a commodity technology. First, long compile times greatly weaken the effectiveness of the compile-test-debug cycle. Second, large-scale deployments of FPGAs are likely to consist of FPGAs of varying sizes, architectures, and even vendors. Requiring developers to maintain, build, optimize, and test on every possible configuration makes it impossible to distribute FPGA logic at the same scale as software executables. For software developers used to the ability to rapidly prototype changes to their code, this is a serious barrier to entry. It diminishes interest in experimenting with FPGAs, and keeps the total number of active hardware developers much lower than it should be.

The obvious solution is to improve the compiler. However there are two reasons why this is not possible. First, the source is unavailable. FPGA hardware and toolchains are produced almost exclusively by two major manufacturers, Intel and Xilinx, and neither has a commercial incentive

```
 1: module Rol(
 2:   input  wire [7:0] x,
 3:   output wire [7:0] y
 4: );
 5:   assign y = (x == 8'h80) ? 1 : (x<<1);
 6: endmodule
```

```
 1: module Main(
 2:   input  wire       clk,
 3:   input  wire [3:0] pad, // dn/up  = 1/0
 4:   output wire [7:0] led  // on/off = 1/0
 5: );
 6:   reg [7:0] cnt = 1;
 7:   Rol r(.x(cnt));
 8:   always @(posedge clk)
 9:     if (pad == 0)
10:       cnt <= r.y;
11:     else
12:       $display(cnt); // unsynthesizable!
13:       $finish;       // unsynthesizable!
14:   assign led = cnt;
15: endmodule
```

Figure 1. A Verilog implementation of the running example.

to open their platforms to developers. Open source initiatives have begun to change this [3, 5]. However most are in their infancy and support a single target at best. The second reason is that compilation for FPGAs is theoretically hard. Transforming HDL into a bitstream is a two-step process. The first involves translation to a *register-transfer level* (RTL) style *intermediate representation* (IR) and the second involves *lowering* (generating a mapping from) that IR onto FPGA fabric. Crucially, this amounts to constraint satisfaction, a known NP-hard problem for which no fast general-purpose solution method exists. While constraint solvers have improved dramatically in the past decade and continue to do so, it is unlikely that a polynomial-time HDL compiler is on its way.

Instead, the current practice is to rely on *hardware simulation*. Running HDL in a simulator does not require a lengthy compilation. However it does have serious drawbacks. First, most FPGA programs involve IO peripherals which must be replaced by software proxies. Building and guaranteeing the correctness of those proxies (assuming the simulation environment supports them — most don't) distracts from the goal of producing a working hardware design. Second, because compilation is NP-hard, functional correctness does not guarantee that a program can be successfully lowered onto an FPGA. Instead, programmers must experiment with many functionally correct designs, attempting a lengthy compilation for each, before arriving at one which works on their architecture. Finally, software debugging techniques such as printf statements cannot be used once a program has left simulation. When bugs inevitably appear in production code running in hardware, they can be difficult to track down.

In this paper, we take a new approach. Rather than attempt to reduce the latency of the compiler, we propose a strategy for *hiding it* behind a simulator in a *just-in-time* (JIT) environment. The key idea is to use a sequence of transformations guided entirely by the syntax of Verilog to translate a program into many small pieces. Importantly, almost no user annotation is required. The pieces are organized into an IR which expresses a distributed system and supports communication between hardware and software. Pieces which interact directly with IO peripherals are automatically replaced by pre-compiled standard components and the remaining pieces begin execution in a software simulator while a potentially lengthy compilation is initiated for each in the background. As these compilations finish, the pieces transition from software to hardware. From the user's point of view, the code runs immediately and simply gets faster over time.

We have implemented these ideas in an open-source system called Cascade[1], the first JIT compiler for Verilog. Cascade reduces the time between initiating compilation and running code to less than a second, preserves program performance to within 3× in a debugging environment, and has minimal effect on a finalized design. In addition to tightening the compile-test-debug cycle, Cascade also improves portability and expressiveness. Automatically mapping IO peripherals onto pre-compiled standard components reduces the burden of porting a program from one architecture to another, and allows programmers to test their code in the same environment as they intend to release it. No IO proxies or simulators are necessary. Furthermore, the use of a runtime which supports communication between software and hardware allows Cascade to support printf-style debugging primitives even after a program has been migrated to hardware. The effect is substantial. We demonstrate through a user study that Cascade encourages more frequent compilation and reduces the time required for developers to produce working hardware designs.

To summarize, our key contribution is a compilation framework which supports a novel programming experience with strong implications for the way that hardware development is taught and carried out in practice. For developers who insist on extracting the fastest designs out of the smallest amount of fabric, there is no substitute for the traditional HDL design flow. However, for developers who are willing to sacrifice a small amount of runtime performance or spatial overhead, Cascade transforms HDL development into something which much more closely resembles writing JavaScript or Python. In short, we take the first steps towards bridging the gap between programming software and programming hardware.

[1] https://github.com/vmware/cascade

```
1: procedure EVAL(e)
2:     if e is an update then
3:         perform sequential update
4:     else
5:         evaluate combinational logic
6:     end if
7:     enqueue new events
8: end procedure
```

```
1: procedure REFERENCESCHEDULER
2:     while ⊤ do
3:         if ∃ activated events then
4:             EVAL(any activated event)
5:         else if ∃ update events then
6:             activate all update events
7:         else
8:             advance time t; schedule recurring events
9:         end if
10:    end while
11: end procedure
```

Figure 2. The Verilog reference scheduler, shown simplified.

```
1: module Rol ... endmodule
// next eval'ed declaration here ...
```

```
1: module Main();
2:   Clock clk();   // implicitly
3:   Pad#(4) pad(); //  provided by
4:   Led#(8) led(); //  environment
5:
6:   reg [7:0] cnt = 1;
7:   Rol r(.x(cnt));
8:   always @(posedge clk.val)
9:     if (pad.val == 0)
10:      cnt <= r.y;
11:    ...
14:  // next eval'd statement here ...
15: endmodule
```

```
CASCADE >>> assign led.val = □
```

Figure 3. The Cascade REPL-based user interface, shown with a partial implementation of the running example.

2 Background

We begin with a high level overview of HDLs and the tool flows typical of the hardware programming experience. We frame our discussion in terms of a running example.

2.1 Running Example

Consider an FPGA with a connected set of IO peripherals: four buttons and eight LEDs. The task is to animate the LEDs, and respond when a user presses one of the buttons. The LEDs should illuminate one at a time, in sequence: first, second, third, etc., and then the first again, after the eighth. If the user presses a button, the animation should pause. This task is a deliberate simplification of a representative application. Even still, it requires synchronous and asynchronous event handling, and computation that combines user inputs with internal state. More importantly, the task is complicated by the diversity of platforms on which it might be deployed. Debugging code in a simulator with proxies to represent the IO peripherals is no substitute for running it in the target environment and verifying that the buttons and LEDs indeed work as intended.

2.2 Verilog

A Verilog [6] implementation of the running example is shown in Figure 1. Verilog is one of two standard HDLs which are used to program FPGAs. The alternative, VHDL [7], is essentially isomorphic. The code is organized hierarchically in units called *modules* (`Rol`, `Main`), whose interfaces are defined in terms of input/output ports (`x`, `y`, `clk`, `pad`, `led`). The inputs to the *root* (top-most) module (`Main`) correspond to IO peripherals. Modules can consist of nested modules, arbitrary width wires and registers, and logic gates.

A module with a single assignment (`Rol`) produces the desired animation: when `x` changes, `y` is assigned the next value in the sequence, a one bit rotation to the left. The state of the program is held in a register, `cnt` (`Main:6`), which is connected to an instance of `Rol` (`Main:7`) and used to drive the LEDs (`Main:14`). The value of `cnt` is only updated to the output of `r` (`Main:10`) when the clock transitions from 0 to 1 (`Main:8`) and none of the buttons are pressed (`Main:9`).

2.3 Synthesizable Core

The language constructs discussed so far are part of the *synthesizable core* of Verilog. They describe computation which can be lowered on to the physical circuitry of an FPGA. Outside of that core are *system tasks* such as print statements (`Main:12`) and shutdown directives (`Main:13`). In Figure 1, they have been used to print the state of the program and terminate execution whenever the user presses a button, perhaps as part of a debugging session. While invaluable to a developer, there is no general purpose way for a compiler to preserve system tasks in a release environment. It is not uncommon to deploy an FPGA in a setting where there is no terminal, or there is no kernel to signal.

2.4 Design Flow

The design flow for the code in Figure 1 would typically begin with the use of a software simulator [73, 76]. For programs whose only IO interaction is with a clock, this is an effective way to catch bugs early. Simulation begins in seconds and

```
 1: module Main(
 2:   input  wire       clk,
 3:   input  wire [3:0] pad_val,
 4:   output wire [7:0] led_val,
 5:   output wire [7:0] r_x,
 6:   input  wire [7:0] r_y
 7: );
 8:   reg [7:0] cnt = 1;
 9:   assign r_x = cnt;
10:   always @(posedge clk_val)
11:     if (pad_val == 0)
12:       cnt <= r_y;
13:     else
14:       $display(cnt);
15:       $finish;
16:   assign led_val = cnt;
17: endmodule
```

Figure 4. Cascade's distributed system IR. Modules are transformed into stand-alone Verilog subprograms.

unsynthesizable Verilog is useful for diagnosing logic errors. However as with most programs, the running example involves IO peripherals. As a result, simulation would only be possible if the user was willing to implement software proxies, a task which is time consuming and error prone.

The next step would be the use of a *synthesis* tool [39, 88, 89] to transform the program into an RTL-like IR consisting of wires, logic gates, registers, and state machines. Synthesis can take from minutes to hours depending on the aggressiveness of the optimizations (eg. state machine minimization) which are applied. Unsynthesizable code is deleted and further debugging still requires the use of proxies. While the resulting code would be closer to what would be lowered onto hardware, it would now need to be run in a waveform viewer [33].

The last step would be the use of a *place and route* tool to lower the RTL onto the FPGA fabric, establish connections between top-level input/outputs and peripheral IO devices, and guarantee that the critical path through the resulting circuit does not violate the timing requirements of the device's clock. As with synthesis, this process can take an hour or longer. Once finished, a *programming* tool would be used to reconfigure the FPGA. This process is straightforward and requires less than a millisecond to complete.

Finally, the program could be tested in hardware. As buggy behavior was detected and repaired, the design flow would be restarted from the beginning. To summarize, compilation is slow, code is not portable, and the developer is hampered by the fact that foundational (eg. printf-style) debugging facilities are confined to a different environment than the one in which code is deployed.

2.5 Simulation Reference Model

The reason the code in Figure 1 can be run in so many environments (simulator, waveform viewer, or FPGA) is because the semantics of Verilog are defined abstractly in terms of the reference scheduling algorithm shown (simplified) in Figure 2. The scheduler uses an unordered queue to determine the interleaving of two types of events: *evaluation* (combinational logic) and *update* (sequential logic). Evaluations correspond to changes to stateless components such as logic gates, wires, or system tasks (a change in `cnt` triggers an evaluation of `r.x`, or the rising edge of `clk` may trigger a print statement) and updates correspond to changes to stateful components such as registers (assigning the value of `r.y` to `cnt`). Events are performed *in any order* but only once *activated* (placed on the queue). Evaluations are always active, whereas updates are activated when there are no other active events. When the queue is emptied the system is said to be in an *observable state*. The logical time is advanced, and some events such as the global clock tick are placed back on the queue.

Intuitively, it may be useful for a developer to think of a Verilog program in terms of its hardware realization. Evaluations appear to take place continuously, and updates appear to take place simultaneously whenever their trigger (eg. `posedge clk`) is satisfied. However, any system that produces the same sequence of observable states, whether it be a software simulator, an executable bitstream, or the system described in this paper which transitions freely between the two, is a well-formed model for Verilog.

3 Cascade

We now describe Cascade, the first JIT compiler for Verilog. Cascade is based on the following design goals which are derived from the shortcomings of the hardware development process. We defer an extended discussion of anti-goals to Section 7.

Interactivity Code with IO side effects should run immediately, and a user should be able to modify a running program.

Portability Code written on one platform should run on another with little or no modification.

Expressiveness Unsynthesizable Verilog should remain active after a program has moved to hardware.

Performance Users may trade native performance for expressiveness, but not be forced to sacrifice it for interactivity.

3.1 User Interface

Cascade's user interface, a *Read-Eval-Print-Loop* (REPL) similar to a Python interpreter [83], is shown in Figure 3 with a nearly identical copy of the code from Figure 1. Verilog is lexed, parsed, and type-checked one line at a time, and errors are reported to the user. Code which passes these checks is

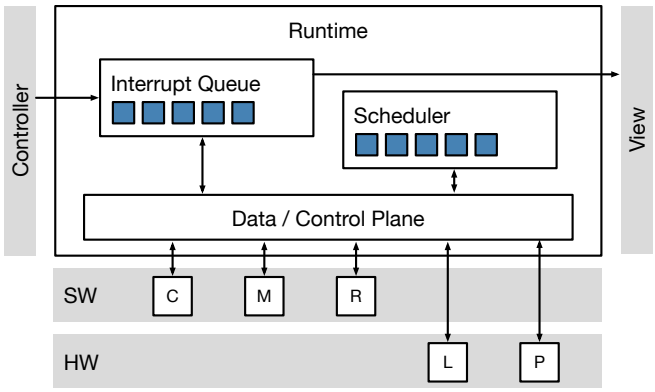

Figure 5. The Cascade runtime architecture.

integrated into the user's program: module declarations are placed in the outer scope, and statements are inserted at the end of a root module which is implicitly instantiated when Cascade begins execution. Code begins execution as soon as it is placed in an instantiated module and IO side effects are visible immediately. Cascade can also be run in batch mode with input provided through a file. The process is the same.

3.2 Standard Library

The only difference between the code Figures 1 and 3 is Cascade's treatment of IO peripherals, which are represented as pre-defined types: `Clock`, `Pad`, and `Led`. These modules are implicitly declared and instantiated when Cascade begins execution, along with whatever other types (eg. `GPIO`, `Reset`) are supported by the user's hardware environment. Several other types supported by all environments (`Memory`, `FIFO`, etc, not shown) may be instantiated at the user's discretion. The Verilog parameterization syntax (`#(n)`) is similar to C++ templates, and used to indicate object width (ie. four buttons, eight LEDs). This design supports **portability** by casting IO configuration as a target-specific implementation detail which can be managed by the compiler. Additionally, it allows Cascade to treat IO peripherals identically to user logic.

3.3 Intermediate Representation

Cascade uses the syntax of Verilog to manage programs at the module granularity. An IR expresses a distributed system composed of Verilog *subprograms* with a constrained protocol. Each subprogram represents a single module whose execution and communication are mediated by messages sent over a data/control plane.

When the user eval's code which instantiates a new module or places a statement at the end of the root module, Cascade uses a static analysis to identify the set of variables accessed by modules other than the one in which they were defined (in Figure 3, `clk.val`, `led.val`, `pad.val`, `r.x`, and `r.y`). Verilog does not allow naming through pointers,

```
1:  procedure EvalAll(E, t)
2:      while events e of type t in E's queue do
3:          Eval(e)
4:      end while
5:  end procedure
```

```
1:  procedure CascadeScheduler
2:      while ⊤ do
3:          if ∃ engine E with evaluation events then
4:              EvalAll(E, evaluation)
5:          else if ∃ engine E with update events then
6:              for all E with update events do
7:                  EvalAll(E, update)
8:              end for
9:          else
10:             service interrupts; end step for all engines
11:             advance time t
12:         end if
13:     end while
14:     end for all engines
15: end procedure
```

Figure 6. The Cascade scheduler.

so this process is tractable, sound, and complete. Cascade then modifies the subprogram source for the modules those variables appear in. Figure 4 shows the transformation for `Main`. First, the variables are promoted to input/outputs and renamed (`r.x` becomes `r_x`). This provides the invariant that no module names a variable outside of its syntactic scope. Next, nested instantiations are replaced by assignments (`Main:9`). The result is that while Verilog's logical structure is hierarchical (`main` contains an instance of `Rol`), Cascade's IR is *flat* (`main` and that instance are peers).

The runtime state of a subprogram (recall that instantiated code begin execution immediately) is represented by a data structure known as an *engine*. Subprograms start as quickly compiled, low-performance, software simulated engines. Over time they are replaced by slowly compiled, high-performance FPGA resident hardware engines. If a subprogram is modified, its engine is transitioned back to software, and the process is repeated. Specifics, and considerations for standard library engines, are discussed in Section 4. Being agnostic to whether engines are located in hardware or software, and being able to transition freely between the two is the mechanism by which Cascade supports **interactivity**.

3.4 Runtime

The Cascade runtime is shown in Figure 5 during an execution of the running example. Boxes `C` through `P` represent engines for the five modules `clk` through `pad`. Some are in software, others in hardware, but to the user, it *appears*

```
 1: struct Engine {
 2:   virtual State* get_state() = 0;
 3:   virtual void set_state(State* s) = 0;
 4:
 5:   virtual void read(Event* e) = 0;
 6:   virtual void write(Event* e) = 0;
 7:
 8:   virtual bool there_are_updates() = 0;
 9:   virtual void update() = 0;
10:   virtual bool there_are_evals() = 0;
11:   virtual void evaluate() = 0;
12:   virtual void end_step();
13:   virtual void end();
14:
15:   virtual void display(String* s) = 0;
16:   virtual void finish() = 0;
17:
18:   virtual void forward(Core* c);
19:   virtual void open_loop(int steps);
20: };
```

Figure 7. The Cascade target-specific engine ABI.

as though they are all in hardware. The user interacts with Cascade through a controller and observes program outputs through a view, which collectively form the REPL. The user's input, system task side effects, and runtime events are stored on an ordered interrupt queue, and a scheduler is used to orchestrate program execution by sending messages across the control/data plane.

The Cascade scheduler is shown in Figure 6. While formally equivalent to the reference, it has several structural differences. First, the scheduler batches events at the module granularity. If an engine has at least one active evaluation, the scheduler requests that it perform them all. If at least one engine has at least one active update event, it requests that all such engines perform them all. Second, the propagation of events generated by these computations takes place only when a batch has completed rather than as they become available. Finally, because eval'ing new code can affect program semantics, it is crucial that it happen when it cannot result in undefined behavior. This is guaranteed to be true in between time steps, when the event queue is empty, and the system is in an observable state. Cascade uses this window to update its IR by creating new engines in response to module instantiations, and rebuilding engines based on new read/write patterns between modules. The replacement of software engines with hardware engines as they become available, as well as interrupt handling (ie. passing `display` events to the view, or terminating in response to `finish`), and rescheduling recurring events like the global clock tick, take place during this window as well.

3.5 Target-Specific Engine ABI

Cascade is able to remain agnostic about where engines are located by imposing a constrained protocol on its IR. This protocol is captured by the *Application Binary Interface* (ABI) shown in Figure 7. Creating new implementations of this class is the mechanism by which developers can extend Cascade's support for new backend targets (we discuss two such implementations in Section 5). Importantly, *this is not* a user-exposed interface. The implementation details of target-specific engines are deliberately hidden from Verilog programmers inside Cascade's runtime.

Engines must support `get` and `set` methods so the runtime can manage their internal state (e.g. when `Main`'s engine transitions from software to hardware, `cnt` must be preserved rather than reset it to 1, as this would disturb the LED animation). Again, the absence of pointers implies the algorithm for identifying this state is tractable, sound, and complete. The `there_are_updates`, `there_are_evals`, `update`, and `evaluate` methods are invoked by the Cascade scheduler (lines 3–7), and the two optional `end_step` and `end` methods are invoked when the interrupt queue is empty (line 10), and on shutdown (line 14) respectively (e.g. this is how the standard clock re-queues its tick event as in Section 2.5). Engines must also support `read` and `write` methods, which are used to broadcast and discover changes to subprogram input/outputs which result from evaluations and updates. Finally, `display` and `finish` methods are used to notify the runtime of system task evaluation. Requiring these methods of all engines, enables **expressiveness** by providing support for unsynthesizable Verilog even from hardware.

4 Performance

Cascade's performance is a function of its runtime overhead and time spent performing engine computation and communication. A depiction is shown in Figure 8, which sets aside the running example and shows the runtime (top), a single software engine (middle), and a single hardware engine (bottom). Time proceeds left to right and computation moves between engines as the runtime invokes their ABIs through the data/control plane. In this section, we describe the optimizations Cascade uses to achieve **performance**, that is, to minimize communication and runtime overhead, and maximize the amount of computation in fast FPGA fabric.

4.1 Goals

Hardware and software engines occupy different clock domains: software operates in GHz, and FPGAs in MHz. Further, the number of cycles a software engine takes to process an ABI request may be very different than for a hardware engine (e.g. thousands of CPU instructions versus a single FPGA clock tick). We define Cascade's performance in terms of

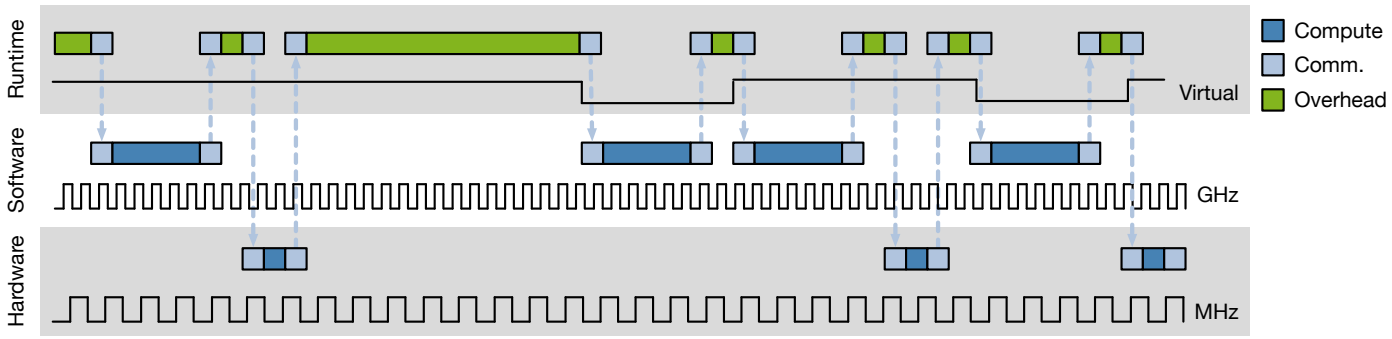

Figure 8. Cascade measures performance in terms of an aperiodic virtual clock defined over multiple physical clock domains.

its *virtual clock*, the *average* rate at which it can dispatch iterations of its scheduling loop (variable amounts of user interaction and ABI requests per iteration imply aperiodicity). Because the standard library's clock is just another engine, every two iterations of the scheduler correspond to a single virtual tick (up on the first, down on the second). Cascade's goal is to produce a virtual clock rate that matches the physical clock rate of the user's FPGA. Figure 9 shows the process for doing so.

4.2 User Logic

Returning to the running example, user logic (modules `Main` and `r`) begin execution in separate software engines (Figure 9.1). Because Cascade's IR is flat, all communication passes through the data/control plane, even though `r`'s input/outputs are referenced exclusively by `Main`. The first optimization that Cascade performs is to inline user logic into a single subprogram. Verilog does not allow dynamic allocation of modules, so the process is tractable, sound, and complete. A new engine is allocated for the inlined subprogram (Figure 9.2), it inherits state and control from the old engines, and the number of `read` and `write` requests sent across the data/control plane, along with the number of `evaluate` and `update` requests required for the event queue to fixed point, are significantly reduced. At the same time, Cascade creates a new hardware engine which begins the process of compiling the inlined subprogram in the background. When compilation is complete, the hardware engine inherits state and control from the inlined software engine (Figure 9.3). From this point on, nearly all ABI requests are processed at hardware speed.

4.3 Standard Library Components

Standard library components with IO side effects must be placed in hardware as soon as they are instantiated, as emulating their behavior in software doesn't make sense (Figure 9.1). This means the time to compile them to hardware can't be hidden by simulation. To address this, Cascade maintains a small catalog of pre-compiled engines for the modules in its standard library. While this allows a program in any compilation state (Figure 9.1–3) to generate IO side effects immediately, interacting with those pre-compiled engines still requires data/control plane communication. Worse, once user logic has migrated to hardware (Figure 9.3), this overhead can represent a majority of Cascade's total runtime.

For these components, inlining is insufficient for eliminating the overhead. There is no source to inline; they are pre-compiled code which respond to requests *as though* they were user logic. The solution is to observe that *if* they were inlined into the user logic engine (Figure 9.3), it would become the single entry and exit point for all runtime/hardware communication. As a result, engines may support ABI forwarding (Figure 7). If so, the runtime can cease direct interaction with standard components and trust the user logic engine to respond to requests on behalf of itself and any standard components it contains (eg. by recursively invoking `evaluate` requests on those engines, or responding true to `there_are_updates` requests if it or those engines have updates). With this (Figure 9.4), the only obstacle to pure hardware performance becomes the interaction with the runtime's virtual clock.

4.4 Open-Loop Scheduling

Processing ABI requests in hardware can be done in a single FPGA clock tick (Section 5). Nonetheless, sending *even one* message between hardware and software per scheduler iteration can be prohibitive. The bandwidth to sustain a virtual clock rate in a typical FPGA range (10–100 MHz) would be an order of magnitude greater (0.1–1 GB/s), a value unlikely to be achieved outside of a very high-performance setting. The key to overcoming this limit is to relax the requirement of direct communication on every scheduler iteration.

Observe that any program in the state shown in Figure 9.4 will exhibit the same schedule. Every iteration, the clock reports `there_are_updates`, an `update` causes a tick, and the user logic alternates between invocations of `update` and `evaluate` until both `there_are_updates` and also `there_are_evals` return false. Thereafter, the process repeats. To take advantage of this, hardware engines may support the `open_loop` request (Figure 7) which tells an

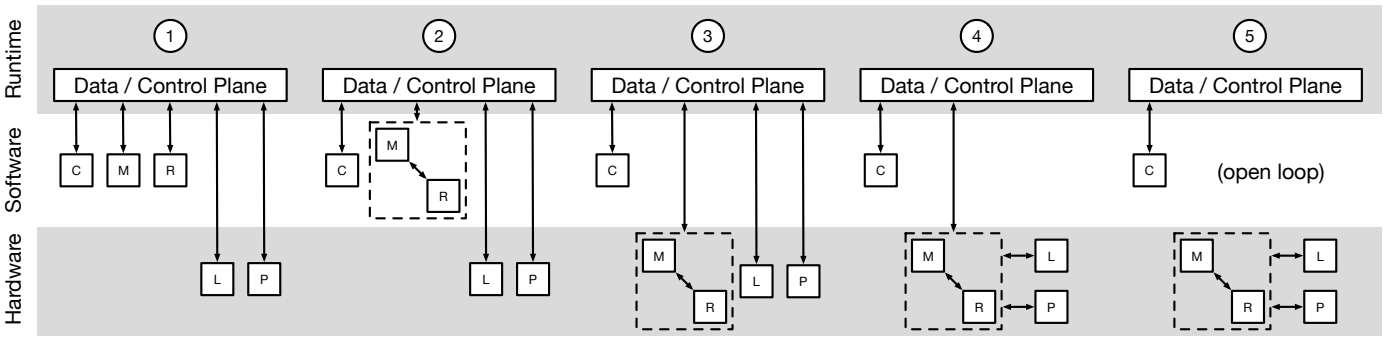

Figure 9. Cascade's optimization flow. Engines transition from software to hardware, and reduced interaction with the runtime.

engine to simulate as many iterations as possible of the schedule described above. Control remains in the engine either until an upper limit of iterations is reached, or the evaluation of a system task requires runtime intervention (Figure 9.5).

Because placing control in an engine stalls the runtime, adaptive profiling is used to choose an iteration limit which allows the engine to relinquish control on a regular basis (typically a small number of seconds). Cascade does its best to transition to open loop quickly and stay there for as long as possible. However, whenever a user interaction causes an update to program logic, engines must be moved back into software and the process started anew.

4.5 Native Mode

Open-loop scheduling can achieve virtual clock rates within a small constant of native performance (Section 6). However, applications which do not use unsynthesizable Verilog and are no longer undergoing active modification are eligible for one final optimization. Placing Cascade in native mode causes it to compile the user's program exactly as written with an off-the-shelf toolchain. This sacrifices **interactivity**, but achieves full native performance, and is appropriate for applications which are no longer being actively debugged.

5 Target-Specific Details

We conclude our discussion of Cascade with implementation notes for simulator-based software engines and FPGA-resident hardware engines. This material in particularly low-level and provided for the sake of completeness. Readers who wish to return later may safely skip ahead to the evaluation (Section 6).

5.1 Software Engines

Software engines use a cycle-accurate event-driven simulation strategy similar to iVerilog [73]. The Verilog source for a subprogram is held in an in-memory AST data structure along with values for its stateful elements. Cascade computes data dependencies at compile-time and uses a lazy evaluation strategy for AST nodes to reduce the overhead of recomputing outputs in response to changes to subprogram inputs. Software engines inhabit the same process as the runtime; communication and interrupt scheduling take place through the heap.

5.2 Hardware Engines

Hardware engines translate the Verilog source for a subprogram into code which can be compiled by a blackbox toolchain such as Quartus [39] or Vivado [89]. The code uses an AXI-style memory-mapped IO protocol to interact with a software stub which inhabits the same process as the runtime and mediates communication and interrupt scheduling. We describe these transformation by example. The effect on the code in Figure 4, after inlining `r`, is shown in Figure 10.

The port declaration on lines 1–8 is typical of AXI and replaces the original. `CLK` is the native FPGA clock, `RW` indicates a write or read request at address `ADDR`, `IN` and `OUT` are the buses for those requests, and `WAIT` is asserted when the FPGA requires more than one cycle to return a result. A kernel module and top-level connections (not shown) map the code's address space into software memory and guarantee that C-style dereferences in the stub produce the appropriate interactions with the resulting hardware. The shorthand `<LATCH>`, `<OLOOP>`, etc., represents checks for write requests to distinguished addresses, which serve as an RPC mechanism.

Auxiliary variables are introduced on lines 9–13. `_vars` is a storage array with one element for each of `Main`'s inputs (`clk_val` and `pad_val`), stateful elements (`cnt`), and instance of a variable in a display statement (`cnt` again), `_umask` and `_tmask` are used for tracking updates and system tasks, `_oloop` and `_itrs` are used while running in open-loop mode, and the set of shadow variables `_nvars`, `_numask`, etc., are used to store values for their counterparts on the (n)ext `update` request. Mappings between these variables and the names in the original code appear on lines 15–18. The text of the original program appears on lines 20–27, only slightly modified. Update targets are replaced by their shadow variable counterparts (`_nvars[2]`), and the corresponding bit in the update mask is toggled. System tasks are treated similarly. Values which appear in display

```
 1: module Main(
 2:   input  wire        CLK,
 3:   input  wire        RW,
 4:   input  wire [31:0] ADDR,
 5:   input  wire [31:0] IN,
 6:   output wire [31:0] OUT,
 7:   output wire        WAIT
 8: );
 9:   reg [31:0] _vars [3:0];
10:   reg [31:0] _nvars [3:0];
11:   reg _umask = 0, _numask = 0;
12:   reg [ 1:0] _tmask = 0, _ntmask = 0;
13:   reg [31:0] _oloop = 0, _itrs = 0;
14:
15:   wire clk_val = _vars[0];
16:   wire [3:0] pad_val = _vars[1];
17:   wire [7:0] led_val;
18:   wire [7:0] cnt = _vars[2];
19:
20:   always @(posedge clk_val)
21:     if (pad_val == 0)
22:       _nvars[2] <= pad_val << 1;
23:       _numask <= ~_umask;
24:     else
25:       _nvars[3] <= cnt;
26:       _ntmask <= ~_tmask;
27:   assign led_val = cnt;
28:   wire _updates = _umask ^ _numask;
29:   wire _latch = <LATCH> |
30:     (_updates & _oloop);
31:   wire _tasks = _tmask ^ _ntmask;
32:   wire _clear = <CLEAR>;
33:   wire _otick = _oloop & !_tasks;
34:
35:   always @(posedge CLK)
36:     _umask <= _latch ? _numask : _umask;
37:     _tmask <= _clear ? _ntmask : _tmask;
38:     _oloop <= <OLOOP> ? IN :
39:       _otick ? (_oloop-1) :
40:       _tasks ? 0 : _oloop;
41:     _itrs <= <OLOOP> ? 0 :
42:       _otick ? (_itrs+1) : _itrs;
43:     _vars[0] <= _otick ? (_vars[0]+1) :
44:       <SET 0> ? IN : _vars[0];
45:     _vars[1] <= <SET 1> ? IN : _vars[1];
46:     _vars[2] <= <SET 2> ? IN :
47:       _latch ? _nvars[2] : _vars[2];
48:
49:   assign WAIT = _oloop;
50:   always @(*)
51:     case (ADDR)
52:       0: OUT = clk_val;
53:       // cases omitted ...
54: endmodule
```

Figure 10. Verilog code generated by the hardware engine associated with the inlined user logic from the running example.

statements are saved (`_nvars[3]`), and the bit in the task mask that corresponds to each system task is toggled.

The remaining code supports the Engine ABI. `read` and `write`, and `get_state` and `set_state` are defined in terms of memory dereferences. Lines 49–53 provide access to any of the subprogram's outputs, stateful elements, or variables which appear in a display statement, and lines 43–47 provide write access to its inputs and stateful elements. `there_are_updates` is defined in terms of a read of the `_updates` variable (line 28), which becomes true when one or more shadow variables are changed, and `update` is defined in terms of a write to the `_latch` variable, which synchronizes those variables with their counterparts and clears the update mask (lines 36 and 46). The definition of `evaluate` involves reading the subprogram's output variables and checking the `_tasks` variable (line 31) which becomes true when one or more tasks are triggered. If any of those tasks are display statements, the values of their arguments at the time of triggering are read out (lines 49-53), formatted in the software stub, and forwarded to the runtime. Thereafter, writing the `_clear` variable (line 32) resets the task mask. Writing the `_oloop` variable (line 38) places the code into the control loop described in Section 4.4. Control alternates between toggling the clock variable (line 38) and triggering updates (line 29) until either the target number of iterations is achieved or a task is triggered (lines 33 and 38).

Hardware engines may also establish ABI forwarding (not shown) for the standard components they contain. For combinational elements such as the pads and LEDs in the running example, this involves promoting the two subprogram variables `led_val` and `pad_val` to subprogram input/outputs, and connecting them to the corresponding IO peripherals.

6 Evaluation

We evaluated Cascade using a combination of real-world application and user study. All of the experiments described in this section were performed using the initial open-source release of Cascade (Version 1.0). The release consists of 25,000 lines of C++ code, along with several thousand lines of target-specific Verilog. As an experimental platform, we used an Intel Cyclone V SoC device [9] which consists of an 800 MHz dual core ARM processor, a reprogrammable fabric of 110K logic elements with a 50 MHz clock, and 1 GB of shared DDR3 memory. Cascade's runtime and software engines were configured to run on the ARM cores, and its hardware

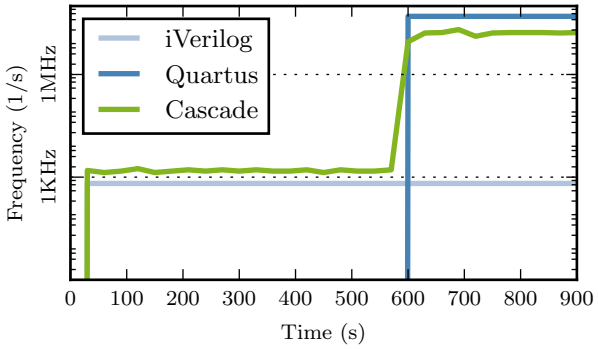

Figure 11. Proof of work performance benchmark.

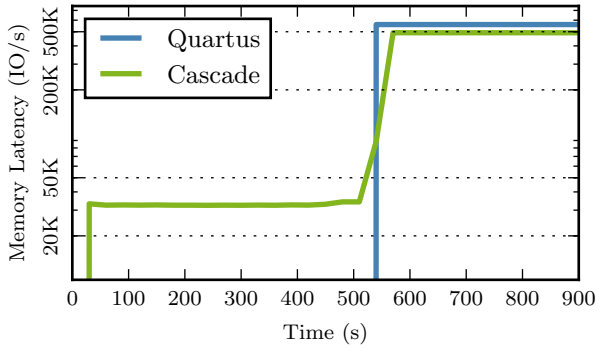

Figure 12. Streaming regular expression IO/s benchmark.

engines on the FPGA. Compilation for the code generated by Cascade's hardware engines was performed using Intel's Quartus Lite compiler (Version 17.0). In order to isolate Cascade's performance properties, the compiler was run on a separate networked server consisting of a four core 2.5 GHz Core i7 with 8 GB of DDR3 memory. In all cases, Cascade's software behavior was compute bound, exhibiting 100% CPU utilization with a memory footprint of less than 10 MB.

6.1 Proof of Work

We used Cascade to run a standard Verilog implementation of the SHA-256 proof of work consensus algorithm used in bitcoin mining [4]. The algorithm combines a block of data with a nonce, applies several rounds of SHA-256 hashing, and repeats until it finds a nonce which produces a hash less than a target value. The algorithm is typical of applications which can benefit from FPGA acceleration: it is embarrassingly parallel, deeply pipelineable, and its design may change suddenly, say, as the proof of work protocol evolves over time.

Figure 11 compares Cascade against Intel's Quartus compiler, and the open source iVerilog simulator [73]. Clock rate over time is shown on a log scale. iVerilog began execution in under one second, but its performance was limited to a virtual clock rate of 650 Hz. Quartus was able to lower the design onto the FPGA and achieve the full 50 MHz native performance, but only after ten minutes of compilation. Cascade was able to achieve the best of both worlds. Cascade began execution in under one second, and achieved a 2.4× faster virtual clock rate through simulation, while performing hardware compilation in the background. When compilation was finished and control was transitioned to open-loop hardware execution, Cascade was able to achieve a virtual clock rate within 2.9× of the native clock while still providing support for unsynthesizable Verilog. The spatial overhead of the bitstream generated by Cascade's hardware engine was small but noticeable: 2.9× that of a direct compilation using Quartus, mainly due to support for `get_state` and `set_state` ABI requests. In native mode, Cascade's performance and spatial requirements were identical to Quartus's.

6.2 Regular Expression Streaming

We used Cascade to run a streaming regular expression matching benchmark generated by a tool similar to the Snort packet sniffer [80] or an SQL query accelerator [40]. In contrast to the previous example, this benchmark also involved an IO peripheral: a FIFO queue used to deliver bytes from the host device to the matching logic. While a real-world application would batch its computation to mask communication overhead, we modified the benchmark to process one byte at a time. This allowed us to measure Cascade's ability to match the memory latency to an IO peripheral provided by the Quartus compiler.

Figure 12 compares Cascade against Intel's Quartus compiler. IO operations per second (tokens consumed) are plotted against time on a log scale. No comparison is given to iVerilog as it does not provide support for interactions with IO peripherals. The implementations are identical, with one exception: the Quartus implementation used the FIFO IP provided by the Quartus IDE, while the Cascade implementation used the FIFO data structure provided by Cascade's standard library. In both cases, host to FPGA transport took place over a memory mapped IO bus [8]. The details were distinct, but not meaningfully different with respect to performance.

Cascade began execution in under one second and achieved an IO latency of 32 KIO/s through simulation. In the same amount of time required for the Quartus implementation to finish compiling (9.5 minutes), Cascade was able to transition to open-loop hardware execution and achieve an IO latency of 492 KIO/s, nearly identical to the 560 KIO/s of the Quartus implementation. In this case, the spatial overhead of the bitstream generated by Cascade's hardware engines was slightly larger (6.5×), though commensurate with that of similar research architectures [47]. As before, Cascade's performance and spatial requirements were identical to Quartus when run in native mode.

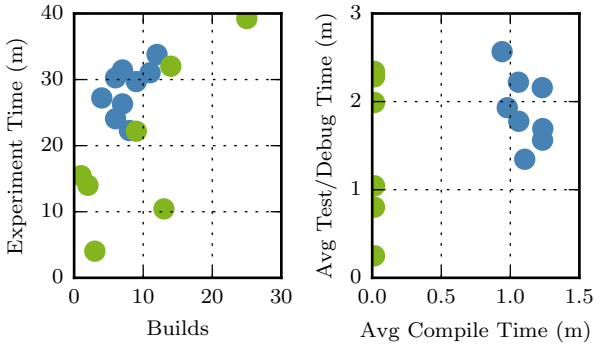

Figure 13. Timing results from user study (data points for the Quartus IDE shown in blue, for Cascade shown in green).

	mean	min	max
Lines of Verilog code	287	113	709
Always blocks	5	2	12
Blocking-assignments	57	28	132
Nonblocking-assignments	7	2	33
Display statements	11	1	32
Number of builds	27	1	123

Table 1. Aggregate statistics characterizing student solutions to Needleman-Wunsch using Cascade.

6.3 User Study

We used Cascade to perform a small user study (n= 20) to test whether JIT compilation can improve the hardware development process. Subjects were drawn from a pool of Computer Science PhD students and full time research staff at VMware, and familiarity with hardware programming was mixed, ranging from none to strong. Subjects were given a 30 minute primer on Verilog and taken to a workstation consisting of an FPGA with IO peripherals, and a hardware development environment. The control group's environment was the Quartus IDE, and the experiment group's was Cascade. In both cases the peripherals were the same: four buttons and strip of 72 individually addressable multi-colored LEDs. Also in both cases, the environments were pre-loaded with a small (50 line) program intended to produce a behavior described in the primer (e.g. pressing one button would cause the LEDs to turn red, pressing another would cause the LEDs to blink in sequence). The subjects were then told that the program contained one or more bugs which would prevent the FPGA from behaving as described. Their task was to fix the bugs and demonstrate a working program as quickly as possible.

Figure 13 summarizes the results of the study. For each participant, we recorded total number of compilations performed, time spent compiling, and time spent testing and debugging in between compilations. The left scatter plot compares number of compilations against time required to complete the task. On average, participants who used Cascade performed 43% more compilations, and completed the task 21% faster than those who used Quartus. Free responses indicated that Cascade users were less concerned with 'wasting time', and more likely to consider using an FPGA in the future. The right scatter plot compares time spent compiling against time spent in between compilations. Participants who used Cascade spent 67× less time compiling, but spent only slightly less time testing and debugging. This agreed with free responses which suggested that while Cascade encouraged faster compilation, it did not encourage sloppy thought.

6.4 UT Concurrency Class Study

We used Cascade as teaching tool for an honors undergraduate concurrency course taught at UT Austin in Fall 2018. The course was designed to provide students with hands-on experience with parallel hardware and programming models that expose concurrency. The assignment asked students to implement a well-known genomics algorithm: Needleman-Wunsch [15]. Students were tasked with comparing scalability with increasing problem size for sequential and parallel CPU implementations, as well as Cascade-based implementations running in software and hardware. For most students, the assignment was a first exposure to Verilog programming. Students were asked to use an instrumented build of Cascade which captured a complete history of compilations to a file that they were encouraged (but not required) to submit along with their completed assignments. Table 1 summarized the results for on analysis of 31 submissions, 23 of which were accompanied by log files.

Students wrote an average 287 lines of verilog (not normalized for boilerplate), tended toward solutions with a very small amount of sequential logic, and over-used blocking assignments (8× more than non-blocking in aggregate, some using none at all). Only 29% of the students arrived at pipelined solutions. In general, students relied overwhelmingly on printf support both in debugging and to verify their final design. Because log capture was not required, it is difficult to make conclusive statements about the impact of JIT support on development times. However, the logs we did collect reflect over 100 build cycles. Even the most conservative assessment of how much the development cycle was shortened based on this data should find it is substantial.

Anecdotally, students were frustrated with some infrastructure aspects of the project. Many submissions which ran correctly in simulation did not pass timing closure during the later phases of JIT compilation. This suggests an important tradeoff between rapid development and feedback on hardware-dependent issues, a topic we defer to future work. Despite this, there was a significant overall positive response.

The class featureed a final research project for which students could choose a combination of problem and platform: nearly 1/3 of the class chose to do further work with Cascade, more than any other combination of technologies.

7 Limitations

Before closing, we briefly consider Cascade's technical limitations and anti-goals that it does not seek to achieve.

7.1 Timing Critical Applications

Cascade presents the illusion that modifications to a running program produce immediately visible hardware side-effects. This is possible because Cascade abstracts away the details of how hardware-located standard library components interact with software-located user logic. For peripherals such as LEDs, the effects of the timing mismatch between the domains are negligible. For others such as FIFOs, back pressure (e.g. a `full` signal) is sufficient for guaranteeing that the data rate of the peripheral does not outstrip the compute throughput of Cascade's software engines. However for applications that use high-performance peripherals (eg. a giga-bit ethernet switch) it is unclear how to preserve higher-order program semantics such as QoS guarantees for compilation states in which user logic has not yet been migrated to hardware.

7.2 Non-Monotonic Language Features

The soundness of executing code immediately after it is eval'ed depends on the invariant that subsequent eval's do not affect the semantics of that code. This is the reason why Cascade's REPL gives users the ability to add code to a running program, but neither the ability to edit nor delete it. Supporting either feature would violate this property. The Verilog specification describes support for several language features that would violate this property for insertions as well. Specifically, it is syntactically legal to re-parameterize modules after they have been instantiated (this is akin to changing a C++ template parameter after an object is created). While Cascade does not support these features, they are deprecated, and will not appear in subsequent revisions of the specification.

8 Related Work

FPGAs are a mature technology with a long history as target of research. We offer a necessarily brief survey of that work.

8.1 Programming and Compilation

FPGAs are programmed at many levels of abstraction. They are often the target of domain specific languages [16, 24, 53, 56, 64, 70, 75, 79] or extensions that integrate FPGAs with high-level imperative languages [1, 13, 14, 25, 38, 48, 54–56]. Frameworks such as OpenCL [48] and Lime [14] or commercial high-level synthesis tools such as Xilinx AutoESL [25], transform C-style code into synthesizable RTL, or HDLs such as Verilog [6], VHDL [7], or BlueSpec [65]. For applications with strict runtime requirements, experts may target these lowest-level languages directly. Compilation at this level is a serious bottleneck and the primary focus of our work.

Many systems in the software domain seek to reduce the overhead of existing compilers. ccache [84], distcc [69], and icecream [32] are gcc frontends that minimize redundant recompilation of sub-components and execute non-interfering tasks simultaneously. Microsoft Cloudbuild [30], Google Bazel [12], and Vesta [37] are distributed caching build systems. These systems do not translate to the hardware domain, where whole-program compilation is the norm. Cascade is an instance of a JIT system. It makes multiple invocations of the compiler in the context of a runtime environment. JIT techniques are used in the compilers for many popular software languages including Java, JavaScript, Python, Matlab, and R.

8.2 Simulation

Hardware simulators can be classified into two partially overlapping categories: event-driven and cycle-accurate. High-fidelity simulators such as those provided by Quartus [39] and Vivado [89] operate at speeds on the order of 10–100 Hz, but are accurate with respect to asynchronous logic and multiple clock domains. Interpreted simulators such as iVerilog [73] do not offer all of these features, but are somewhat faster, approaching 1 KHz. Compiled simulators such as Verilator [76] can operate in the 10 KHz range. Cascade uses JIT compilation techniques to interpolate between these performance domains and native rates of 10 to 100 MHz.

8.3 Hardware-Software Partitioning

Palladium [2] allows users to actively migrate between simulation, accelerated simulation, and emulation environments at runtime. Lime [14] provides a high level language and runtime environment capable of dynamically repartitioning programs across hardware and software. Both systems are capable of moving code back and forth between software and hardware, but neither provides features similar to Cascade's native mode, neither provides a software-style environment for debugging, and neither allows execution of unsynthesizable Verilog in hardware. Unlike Lime, Cascade does not rely on higher-level language support. And unlike Palladium, Cascade does not require expensive hardware support. Instead, Cascade uses source-to-source translation to provide these features on low-cost development boards, and can easily be extended to new targets.

LEAP [11] is a compiler-supported OS for FPGAs that enables flexible partitioning of modules which communicate over OS managed latency insensitive channels. LEAP enables dynamic management of FPGA resources and hardware/-software partitioning of Bluespec modules using a compiler extension interface called *SoftServices*. SoftServices provide a

portability mechanism for service-specific functionality similar to Cascade's standard library support for clocks and IO (Section 3.2). Unlike LEAP, Cascade does not rely on programmer exposed interfaces or channel abstractions for dynamic partitioning of work between software and hardware.

8.4 Operating Systems and Virtualization

Coordinating software simulation and native execution in a runtime environment requires design choices which resemble operating system and virtualization primitives. Many of these have been explored in the context of FPGAs: spatial multiplexing [23, 31, 81, 86], task preemption [59], relocation [43], context switching [58, 72], and interleaved hardware-software execution [18, 34, 81, 86]. Several projects have extended these concepts to full-fledged operating systems for FPGA. These include ReconOS [61], Borph [77, 78], and MURAC [35]. Others have extended these concepts to FPGA hypervisors. These include CODEZERO [67], Zippy [68], TARTAN [63], and SCORE [28]. Chen et al. explore virtualization challenges that arise in a setting where FPGAs are a shared resource [22]. The work integrates Xilinx FPGAs in OpenStack [74] and Linux-KVM [51], and supports isolation across processes in different virtual machines. Amazon's EC2 F1 FPGA instances [29] are connected to each other through a dedicated isolated network such that sharing between instances, users, and accounts is not permitted. Microsoft Azure Olympus [62] servers are expected to follow a similar model. AmorphOS [47] provides an OS-level management layer to concurrently share FPGAs for acceleration among mutually distrustful processes, based on a compatibility layer that targets F1 [29] and Catapult [71].

8.5 Communication Models

Many connection strategies exist for exposing FPGAs as hardware accelerators. In coprocessor-coupled platform such as ZYNQ [26] and Arria [36] an FPGA is connected to a dedicated CPU which is tasked with mediating interaction with the host system. In host-coupled platforms, there is no coprocessor. Instead, FPGA fabric must be set aside for the implementation of a mediation layer such as a PCIe bridge or an Avalon Memory Bus [8]. Cascade's hardware engines are an instance of the latter strategy.

8.6 Abstraction and Compatibility

Cascade's runtime environment is an instance of an overlay system. FPGA overlays implement a virtualization layer to abstract a design from specific FPGA hardware [17, 87], enabling fast compilation times and lower deployment latency [45, 52], at the expense of reduced hardware utilization and performance. Examples of overlays include ZUMA [17], VirtualRC [50], and RCMW [49] which provide bitstream independence across different hardware and toolchains, and VForce [64] which enables the same application to be run on different reconfigurable supercomputers. AmorphOS [47] implements a compatibility layer at the OS interface.

9 Conclusion and Future Work

Compilation is a painfully slow part of the hardware design process and a major obstacle to the widespread adoption of FPGA technology. In this paper we presented Cascade, the first JIT compiler for Verilog. Cascade allows users to test and deploy code in the same environment, ignore the distinction between synthesizable and unsynthesizable Verilog, and to enjoy cross-platform portability, while requiring only minimal changes to their code. Cascade tightens the compile-test-debug cycle and allows users to modify programs as they are run. Side-effects on IO peripherals become visible immediately, debugging incurs no more than a 3× performance penalty, and near native performance is supported for finalized designs.

Future work will explore the development of dynamic optimization techniques which can produce performance and layout improvements by specializing a program to the input values it encounters at runtime. Future work will also consider the use of Cascade as a platform for FPGA virtualization. Specifically, multi-runtime aware backends could be used to temporarily and spatially multiplex FPGA fabric, and Cascade's ability to move programs back and forth between hardware and software could be used to bootstrap virtual machine migration for systems that use hardware accelerators.

Acknowledgments

We thank the VMware employees and interns who participated in our user study. We also thank the UT Austin CS 378H Fall 2018 students for being the first external group to use Cascade, and for their enthusiastic contributions to its open source code.

References

[1] [n. d.]. *AppArmor*. http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[2] [n. d.]. Cadence Palladium XP II Verification Computing Platform. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-xpii-tb.pdf. (Accessed January 2019).
[3] [n. d.]. Debian – Details of Package fpgatools. https://packages.debian.org/stretch/fpgatools. (Accessed July 2018).
[4] [n. d.]. FPGAMiner. https://github.com/fpgaminer/Open-Source-FPGA-Bitcoin-Miner. (Accessed July 2018).
[5] [n. d.]. SymbiFlow. https://symbiflow.github.io/. (Accessed July 2018).
[6] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–560.
[7] 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan 2009), c1–626.
[8] 2017. Avalon Interface Specifications.
[9] 2017. Device Handbook — Altera Cyclone V.

[10] 2017. *Intel unveils new Xeon chip with integrated FPGA, touts 20x performance boost - ExtremeTech.* https://www.extremetech.com/extreme/184828-intel-unveils-new-xeon-chip-with-integrated-fpga-touts-20x-performance-boost

[11] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 25–28.

[12] K Aehlig et al. 2016. Bazel: Correct, reproducible, fast builds for everyone. https://bazel.io

[13] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ron Sass, and David Andrews. 2006. Enabling a Uniform Programming Model across the Software/Hardware Boundary. FCCM '06.

[14] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 89–108.

[15] Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to Search for Similarities in Amino Acid Sequence of 2 Proteins. *Journal of molecular biology* 48 (04 1970), 443–53. https://doi.org/10.1016/0022-2836(70)90057-4

[16] 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.

[17] Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 93–96.

[18] Gordon J. Brebner. 1996. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*. Springer-Verlag, London, UK, UK, 327–336.

[19] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2015. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 94–97.

[20] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14)*. ACM, New York, NY, USA, 151–160.

[21] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture, In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture.

[22] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, New York, NY, USA, Article 3, 10 pages.

[23] Liang Chen, Thomas Marconi, and Tulika Mitra. 2012. Online Scheduling for Multi-core Shared Reconfigurable Fabric. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*. EDA Consortium, San Jose, CA, USA, 582–585.

[24] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *40th International Symposium on Computer Architecture*. ACM.

[25] Philippe Coussy and Adam Morawiec. 2008. *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media.

[26] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. 2014. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media.

[27] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 105–110.

[28] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. 2006. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems* 30, 6 (2006), 334–354.

[29] Amazon EC2. 2017. Amazon EC2 F1 Instances.

[30] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 11–20.

[31] W. Fu and K. Compton. 2008. Scheduling Intervals for Reconfigurable Computing. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*. 87–96.

[32] K Funk et al. 2016. icecream. https://github.com/icecc/icecream

[33] GNU. [n. d.]. GTKWave. http://gtkwave.sourceforge.net. (Accessed July 2018).

[34] Ivan Gonzalez, Sergio Lopez-Buedo, Gustavo Sutter, Diego Sanchez-Roman, Francisco J. Gomez-Arribas, and Javier Aracil. 2012. Virtualization of Reconfigurable Coprocessors in HPRC Systems with Multicore Architecture. *J. Syst. Archit.* 58, 6-7 (June 2012), 247–256.

[35] B. K. Hamilton, M. Inggs, and H. K. H. So. 2014. Scheduling Mixed-Architecture Processes in Tightly Coupled FPGA-CPU Reconfigurable Computers. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. 240–240.

[36] Arria V Device Handbook. 2012. Volume 1: Device Overview and Datasheet. (2012).

[37] Allan Heydon, Timothy Mann, Roy Levin, and Yuan Yu. 2006. *Software Configuration Management Using Vesta*. Springer.

[38] SRC Computers Inc. 2006. Carte Programming Environment.

[39] Intel. 2018. Intel Quartus Prime Software. https://www.altera.com/products/design-software/fpga-design/quartus-prime/download.html

[40] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *PVLDB* 10, 11 (2017), 1202–1213.

[41] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 425–438.

[42] Alexander Kaganov, Asif Lakhany, and Paul Chow. 2011. FPGA Acceleration of MultiFactor CDO Pricing. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2, Article 20 (May 2011), 17 pages.

[43] H. Kalte and M. Porrmann. 2005. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on*. 223–228.

[44] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 295–308.

[45] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *FPL*. IEEE, 1–8.

[46] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*. 1–4.

[47] Ahmed Khawaja, , Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. "Sharing, Protection and Compatibility for Reconfigurable Fabric with AmorphOS". In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. (OSDI)*. Carlsbad, CA.

[48] Khronos Group 2009. *The OpenCL Specification, Version 1.0.* Khronos Group.

[49] Robert Kirchgessner, Alan D. George, and Greg Stitt. 2015. Low-Overhead FPGA Middleware for Application Portability and Productivity. *ACM Trans. Reconfigurable Technol. Syst.* 8, 4, Article 21 (Sept. 2015), 22 pages.

[50] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. 2012. VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 205–208.

[51] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. 225–230.

[52] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. 2013. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL*. IEEE, 1–8.

[53] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 115–127. https://doi.org/10.1109/ISCA.2016.20

[54] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 296–311.

[55] James Lebak, Jeremy Kepner, Henry Hoffmann, and Edward Rutledge. 2005. Parallel VSIPL++: An open standard software library for high-performance parallel signal processing. *Proc. IEEE* 93, 2 (2005), 313–330.

[56] Ilia A. Lebedev, Christopher W. Fletcher, Shaoyi Cheng, James Martin, Austin Doupnik, Daniel Burke, Mingjie Lin, and John Wawrzynek. 2012. Exploring Many-Core Design Templates for FPGAs and ASICs. *Int. J. Reconfig. Comp.* 2012 (2012), 439141:1–439141:15.

[57] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL '11)*. IEEE Computer Society, Washington, DC, USA, 317–322.

[58] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. 2010. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. *J. Inf. Sci. Eng.* 26 (2010), 1289–1305.

[59] L. Levinson, R. Manner, M. Sessler, and H. Simmler. 2000. Preemptive multitasking on FPGAs. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. 301–302.

[60] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 476–488.

[61] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Trans. Embed. Comput. Syst.* 9, 1, Article 8 (Oct. 2009), 33 pages.

[62] Microsoft. 2017. Microsoft Azure Goes Back To Rack Servers With Project Olympus.

[63] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 163–174.

[64] Nicholas Moore, Albert Conti, Miriam Leeser, Benjamin Cordes, and Laurie Smith King. 2007. An extensible framework for application portability between reconfigurable supercomputing architectures.

[65] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 69–70.

[66] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 111–117.

[67] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell. 2013. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. 219–226.

[68] Christian Plessl and Marco Platzner. 2005. Zippy-A coarse-grained reconfigurable array with support for hardware virtualization. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*. IEEE, 213–218.

[69] M Pool et al. 2016. distcc: A free distributed C/C++ compiler system. https://github.com/distcc/distcc

[70] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 389–402.

[71] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *41st Annual International Symposium on Computer Architecture (ISCA)*.

[72] Kyle Rupnow, Wenyin Fu, and Katherine Compton. 2009. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *FCCM 2009, 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, USA, 5-7 April 2009, Proceedings*. 63–70.

[73] J. Russell and R. Cohn. 2012. *Icarus Verilog*. Book on Demand. https://books.google.co.uk/books?id=ZNanMQEACAAJ

[74] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012).

[75] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ning-Yi Xu, and Huazhong Yang. 2010. FPMR: MapReduce framework on FPGA.. In *FPGA* (2010-03-01), Peter Y. K. Cheung and John Wawrzynek (Eds.). ACM, 93–102.

[76] W Snyder, D Galbi, and P Wasson. 2018. Verilator. https://veripool.org/wiki/verilator

[77] Hayden Kwok-Hay So and Robert Brodersen. 2008. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH. *ACM Trans. Embed. Comput. Syst.* 7, 2, Article 14 (Jan. 2008), 28 pages.

[78] Hayden Kwok-Hay So and Robert W. Brodersen. 2007. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html

[79] Hayden Kwok-Hay So and John Wawrzynek. 2016. OLAF'16: Second International Workshop on Overlay Architectures for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 1–1.

[80] Haoyu Song, Todd S. Sproull, Michael Attig, and John W. Lockwood. 2005. Snort Offloader: A Reconfigurable Hardware NIDS Filter. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, August 24-26, 2005*. 493–498.

[81] C. Steiger, H. Walder, and M. Platzner. 2004. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11 (Nov 2004), 1393–1407.

[82] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 16–25.

[83] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. 2010. *Node.Js*. Betascript Publishing, Mauritius.

[84] A Tridgell, J Rosdahl, et al. 2016. ccache: A Fast C/C++ Compiler Cache. https://ccache.samba.org

[85] A. Tsutsui, T. Miyazaki, K. Yamada, and N. Ohta. 1995. Special Purpose FPGA for High-speed Digital Telecommunication Systems. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*. IEEE Computer Society, Washington, DC, USA, 486–491.

[86] G. Wassi, Mohamed El Amine Benkhelifa, G. Lawday, F. Verdier, and S. Garcia. 2014. Multi-shape tasks scheduling for online multitasking on FPGAs. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*. 1–7.

[87] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. 2014. Embedding FPGA overlays into configurable Systems-on-Chip: ReconOS meets ZUMA. In *ReConFig*. IEEE, 1–6.

[88] Clifford Wolf. [n. d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/. (Accessed July 2018).

[89] Xilinx. 2018. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html

[90] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. ACM, New York, NY, USA, 161–170.