CASCADE

A Just-in-Time Compiler for FPGAs

ASPLOS 2019

vmware®

Opportunities and Challenges

Performance:

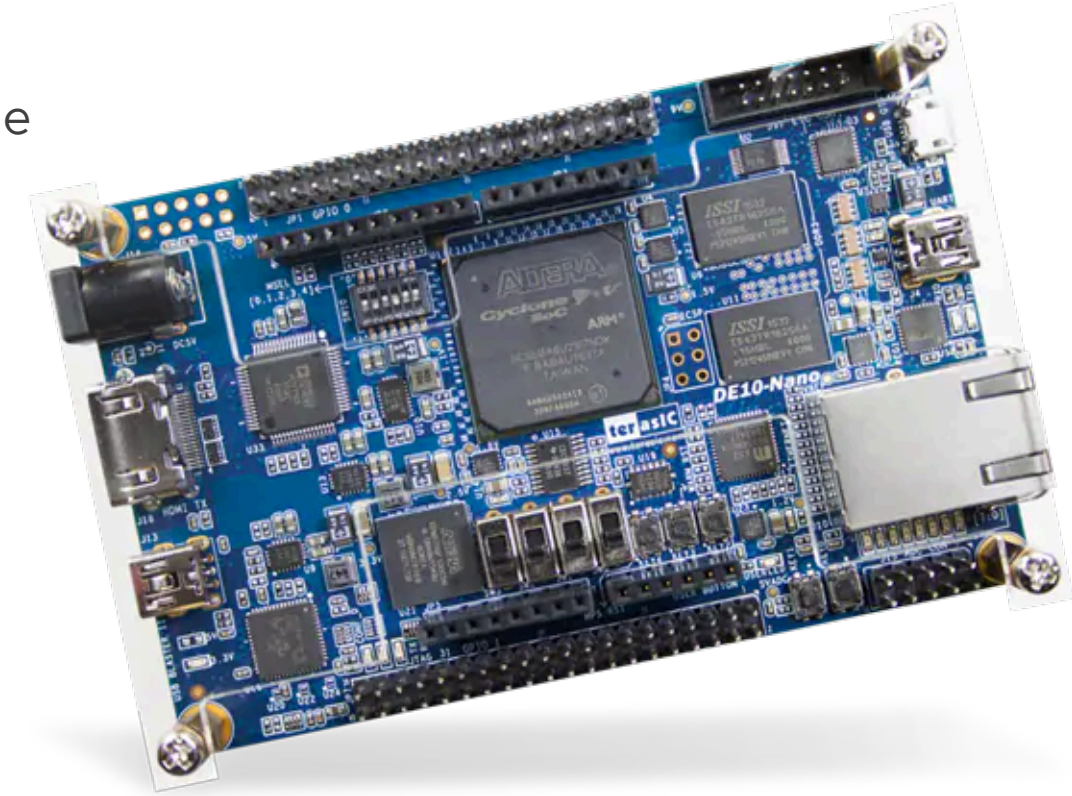
- Faster than CPUs by several orders of magnitude
- Lower cost and time to market than ASICs

Availability:

- Increasingly common in the data-center

Difficult to Program:

- Compilation can take hours or longer!
- Expensive, baroque IDEs!
- No open source alternatives!

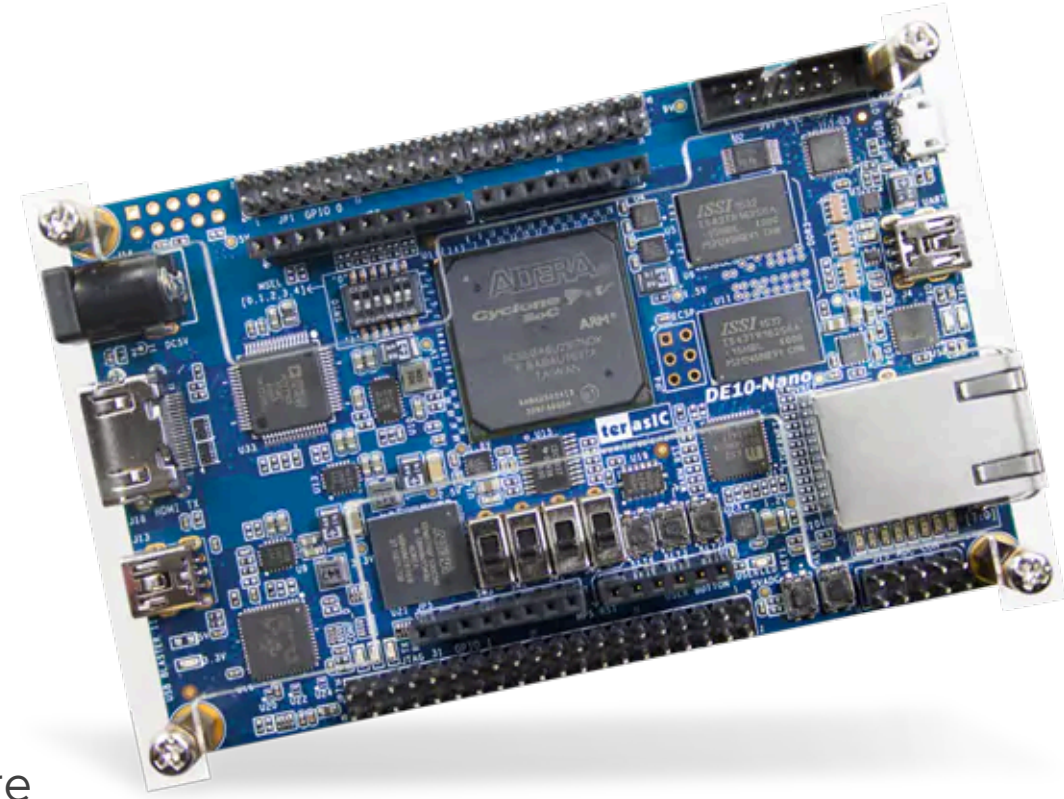


Just-in-Time Compilation

1. Run Code in a Simulator:
   - No waiting for the compiler
   - Shortened compile-test-debug cycle

2. Compile in the Background:
   - Treat closed-source compilers as a black box
   - Replace the IDE with a light-weight front-end

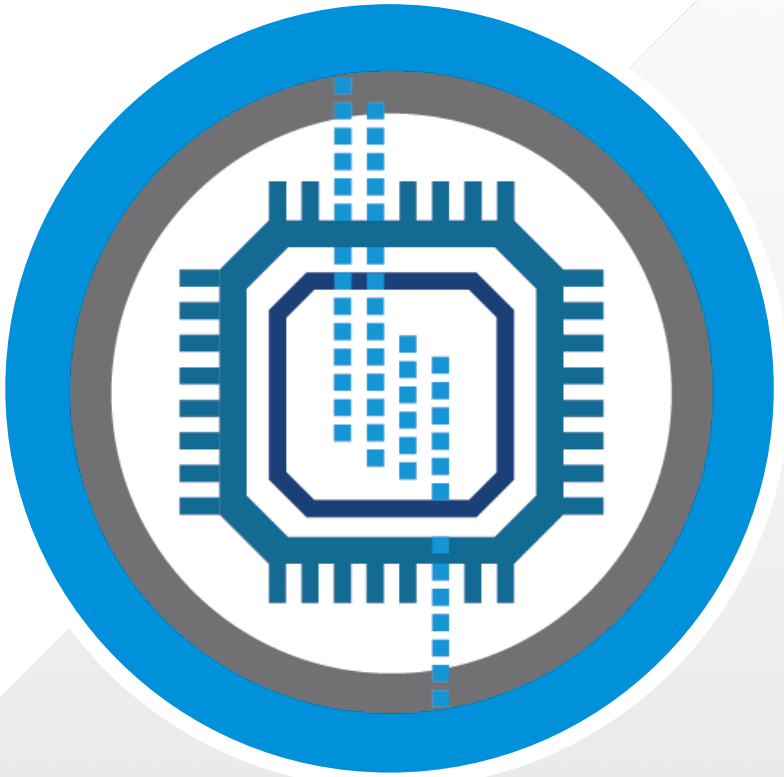
3. Transition to Hardware When Finished:
   - Runtime manages low-level details
   - Code gets faster over time as it reaches hardware



CASCADE:

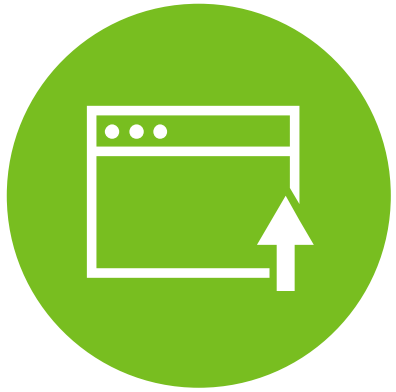
The First Just-in-Time Compiler for FPGAs

https://github.com/vmware/cascade



vmware®

Design Goals

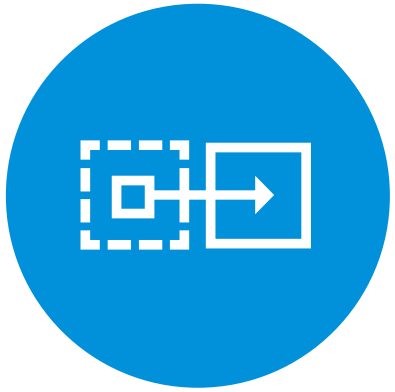


Interactivity

Run code immediately as it's written.

Expressiveness

Printf-style debugging from hardware.



Portability

Write code once. Run it on many platforms.

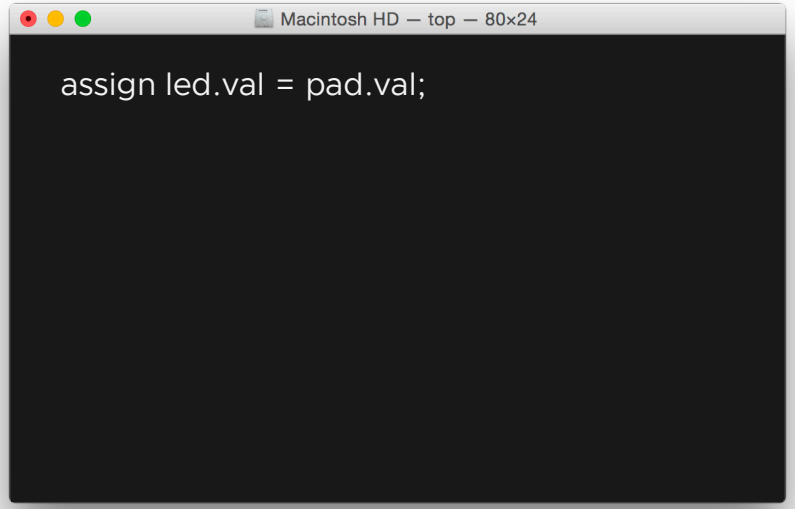
Performance

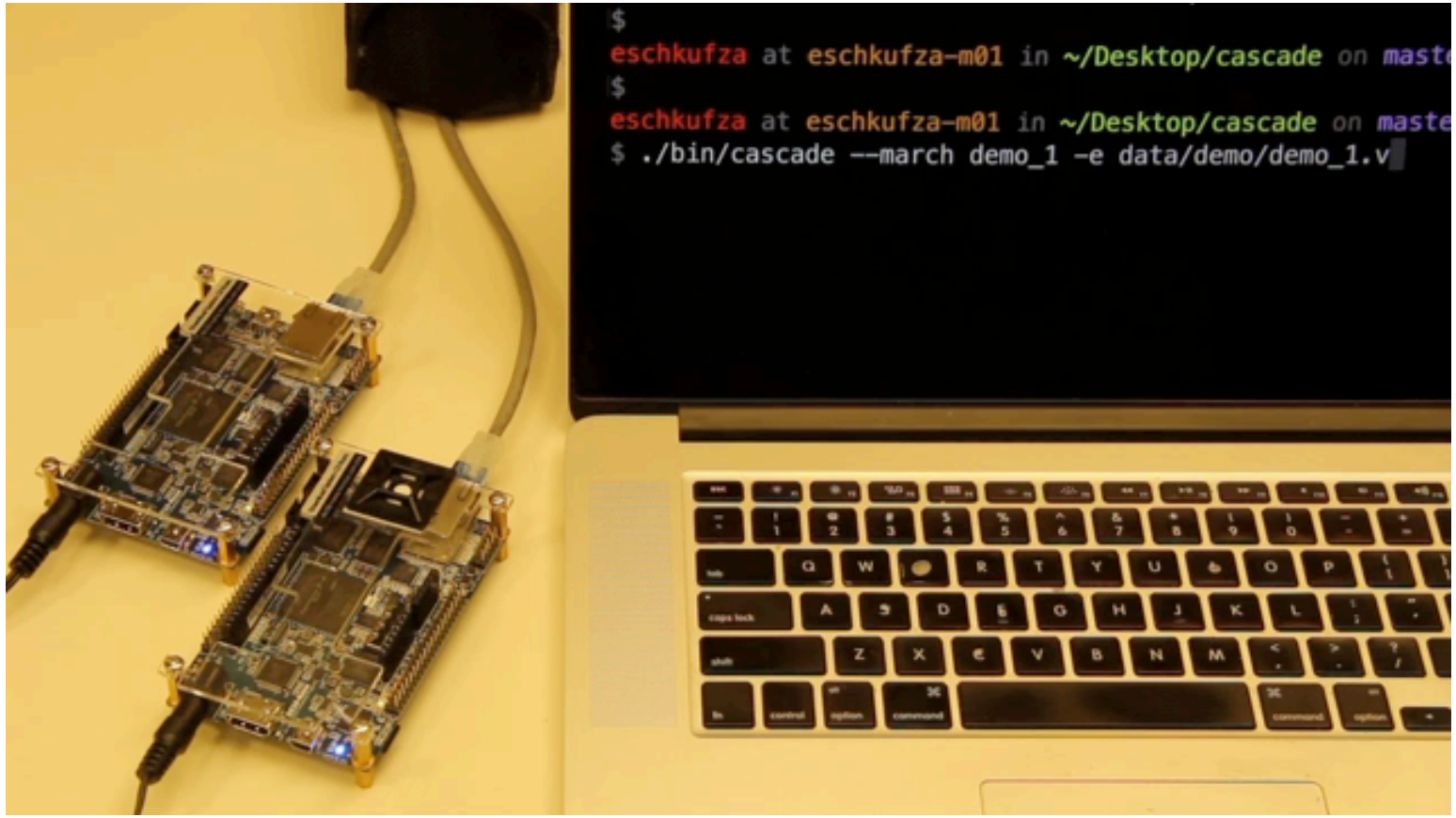
Don't pay for what you don't use.

Basic User Interaction

Code is compiled continuously.

Side effects are immediately visible in hardware.


Macintosh HD — top — 80×24
assign led.val = pad.val;


$
eschkufza at eschkufza-m01 in ~/Desktop/cascade on maste
$
eschkufza at eschkufza-m01 in ~/Desktop/cascade on maste
$ ./bin/cascade --march demo_1 -e data/demo/demo_1.v

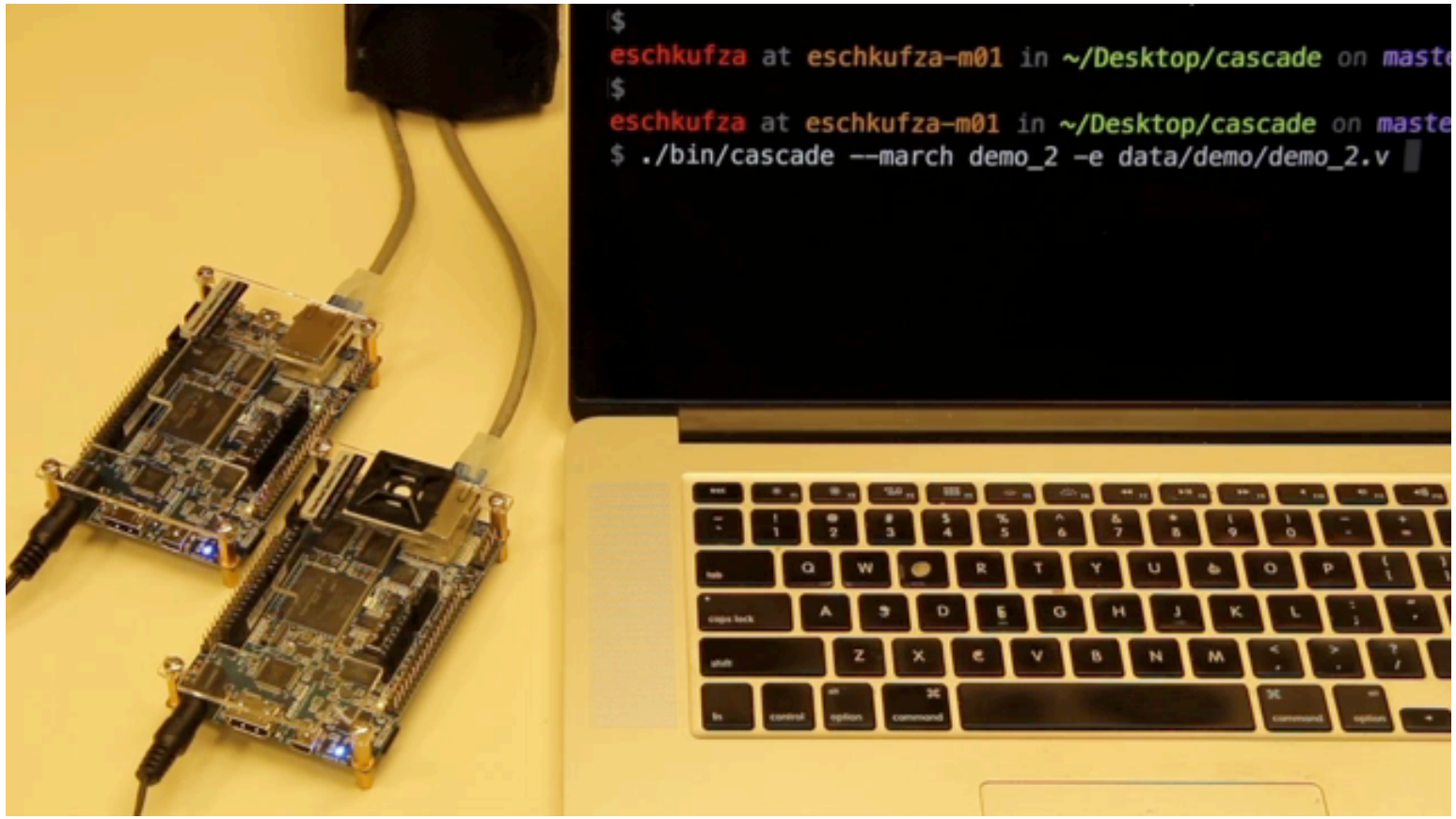
Hardware printf

A message is printed to the terminal when the user interacts with the buttons.

Macintosh HD — top — 80×24

```
assign led.val = pad.val;

reg r = 0;
always @(posedge pad.val) begin
  r <= !r;
  $display(r ? “tick” : “tock”);
end
```



Why Cascade Works

```
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
```

Simulation Reference Model:

- Verilog defined in terms of scheduler algorithm
- Any system which produces the same set of side effects is a well-defined model for Verilog!

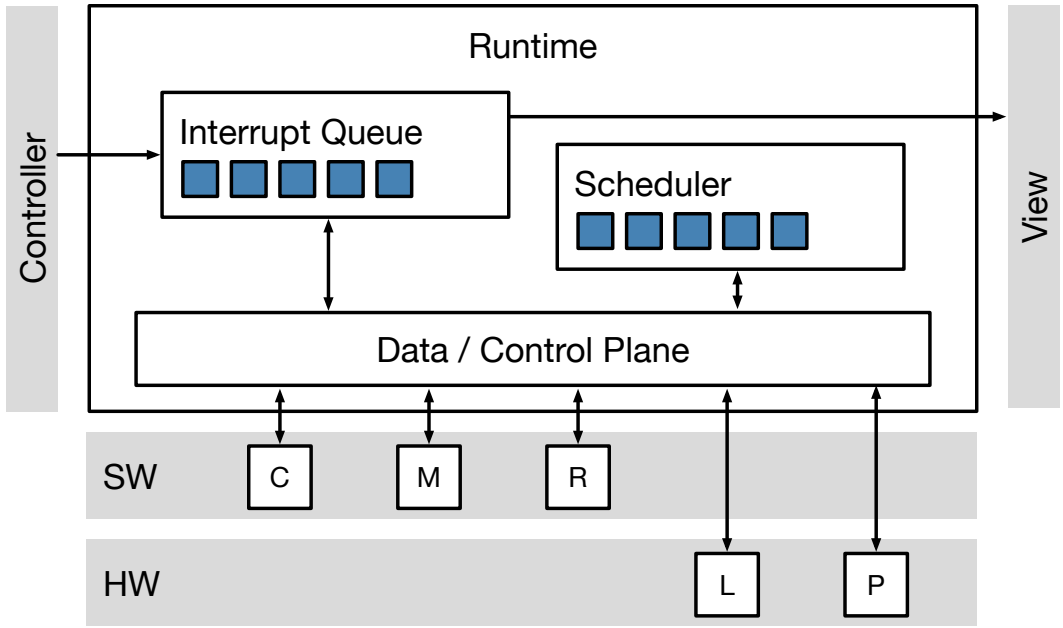
Small-Step Semantics:

- Events processed one at a time
- Observable sequence points give illusion that they take place simultaneously

Safe Windows:

- Well-defined places where state is stable
- Soundly swap implementations, eval new code, emit print statements, etc…

Runtime Architecture



Distributed-System-Like IR:

- Composed of Verilog sub-programs
- Communication over a constrained protocol

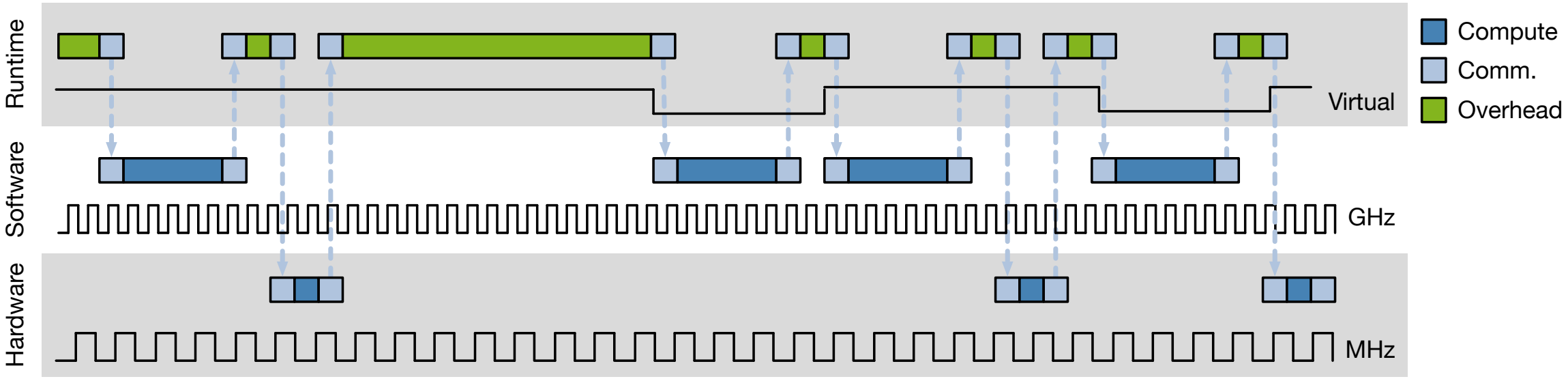
Monadic Semantics:

- Allows dynamic swapping of implementations
- Agnostic to whether modules are located in hardware or software

Data/Control Plane:

- Module state stored in implementation-specific “engines”
- Communication via message passing interface

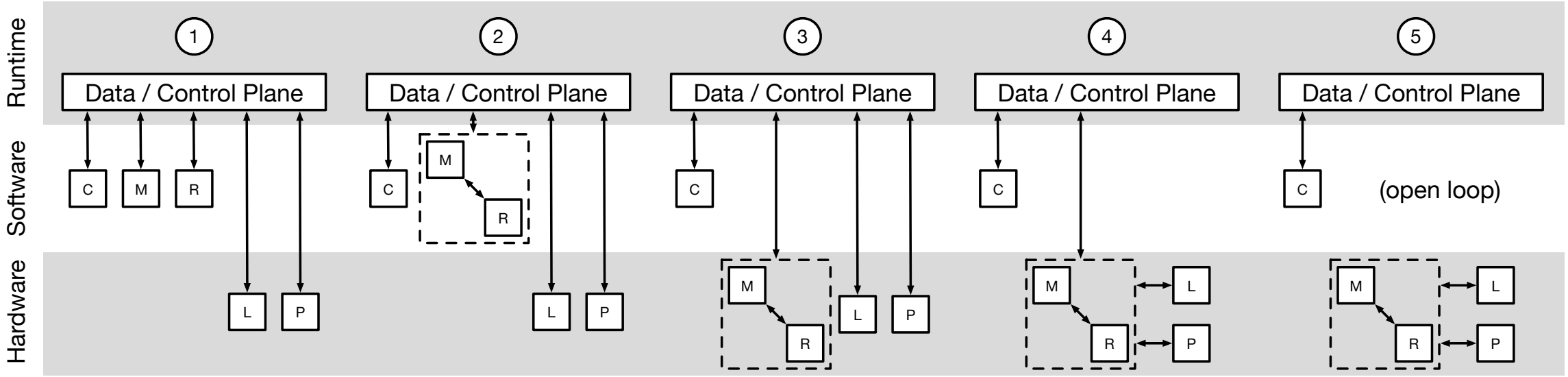
Performance Contract



Multiple Clock Domains:

- Compute at different frequencies, losses due to overhead
- Maximize the number of scheduler iterations per second (virtual clock frequency)

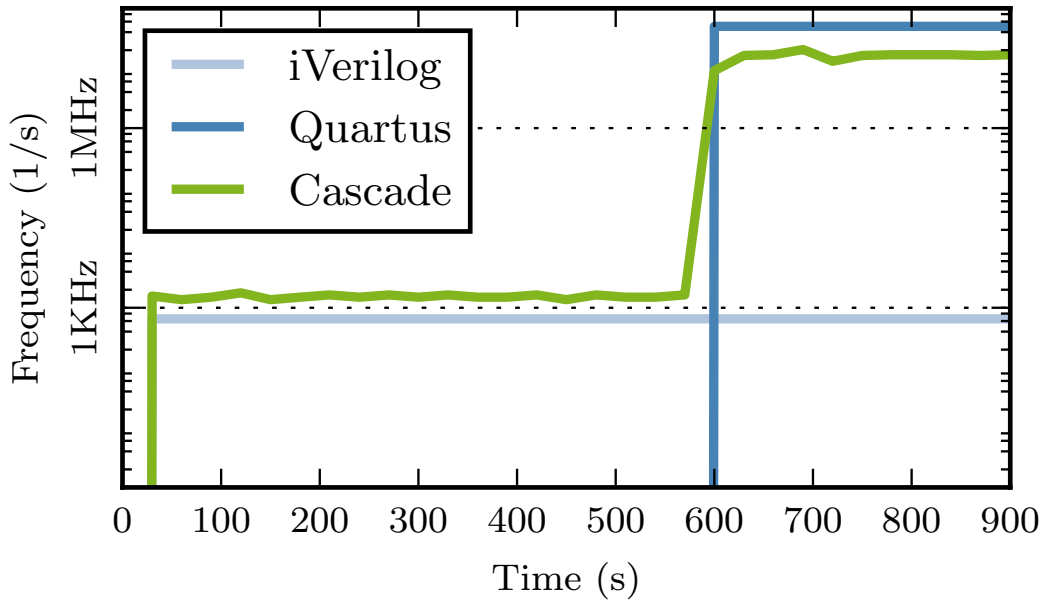
Optimization Pipeline

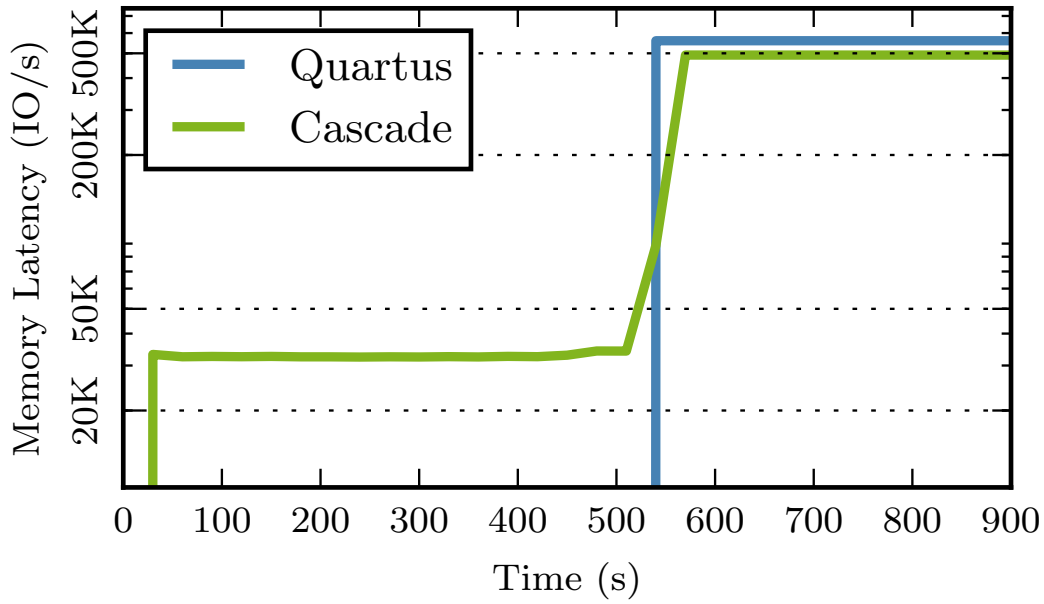


High-Level Concepts:

- Gradual transition from software to hardware
- Dynamic inlining removes overhead of runtime-mediated communication
- Open-loop execution removes overhead of runtime-mediated scheduling

Experiments

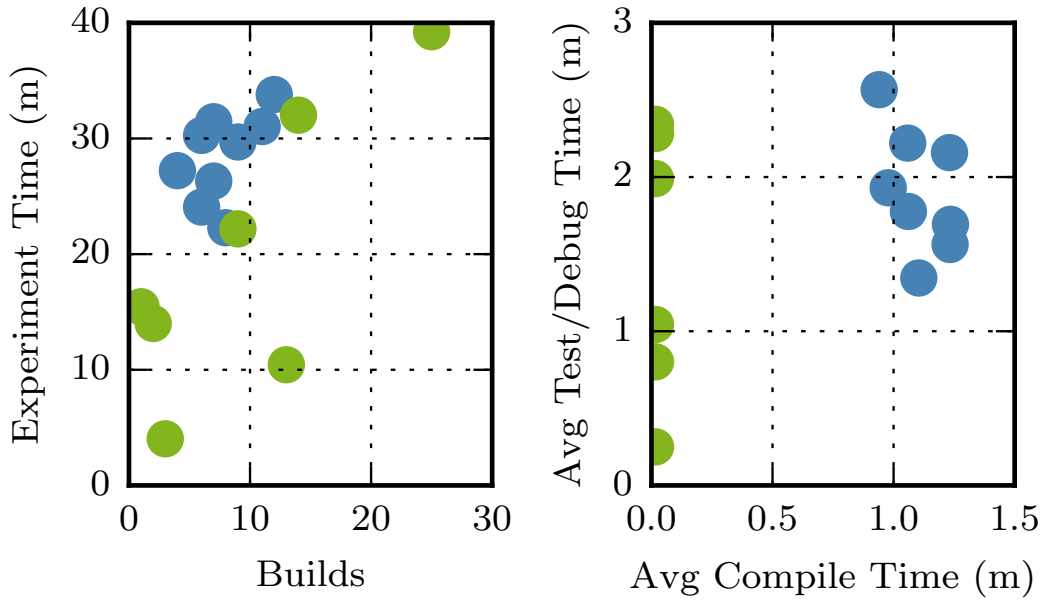




Benchmarks:

- Compute bound (bitcoin miner) and communication bound (regular expression streaming)
- Code begins execution with performance comparable to simulator, approaches native performance in the limit, retains support for unsynthesizable Verilog

User Study



Debugging Task:

- Users given spec and buggy code
- Asked to produce a working solution

Sample Set:

- 20 participants ranging from novice to expert
- Half given Cascade, half given Quartus

Observations:

- Users performed more compilation cycles and finished task faster when using Cascade
- Spent same time testing and debugging
- Encourages more compilation, not sloppy coding

Thank You

https://github.com/vmware/cascade

https://research.vmware.com

vmware®