# SimCommand and Chiseltest: High-Performance RTL Testbench APIs

Vighnesh Iyer, Young-Jin Park, Oliver Yu, Andy Lin, Kevin Laeufer, Bora Nikolic
UC Berkeley

## Motivation

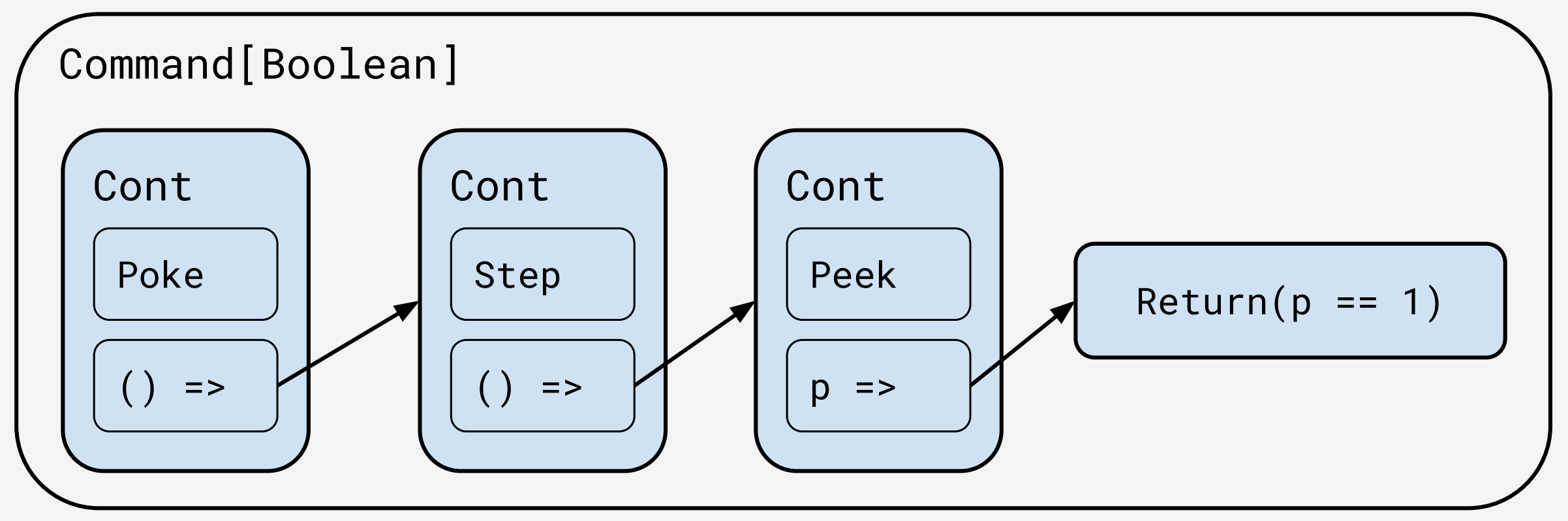
- Leverage general-purpose languages and their libraries, build systems, IDEs, and test frameworks
- Move beyond SystemVerilog + UVM without compromising on features or performance

## SimCommand Overview

**Goal:** A Scala-embedded testbench API with fork/join support and performance parity with SystemVerilog testbenches

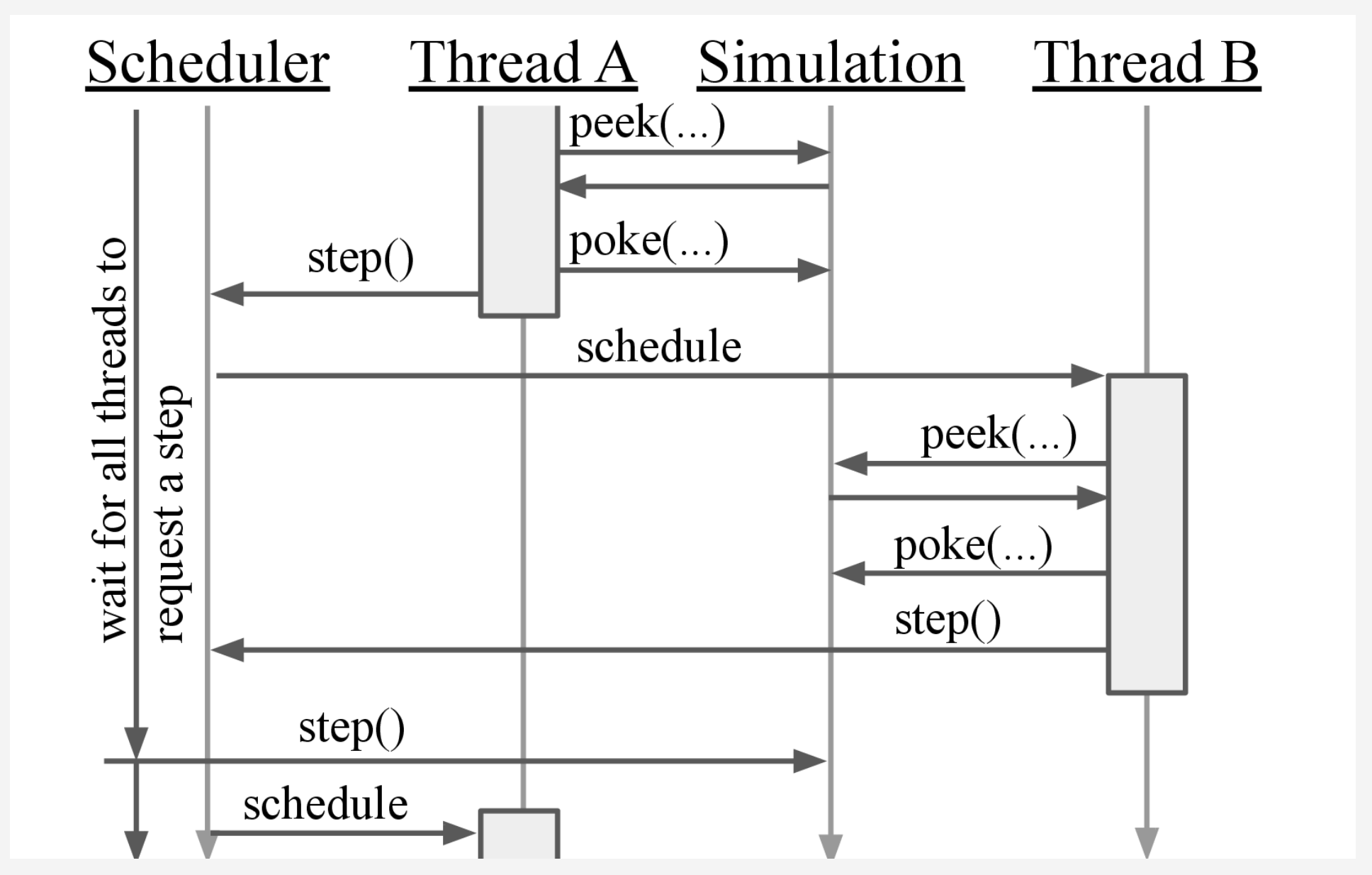
- Separation of description and interpretation of testbench actions
- Threads are just pointers to continuations → fast thread pause/resume
- A `Command[R]` is a *description* of testbench actions which terminates with a value of type R

```
val program: Command[Boolean] = for {
    _ ← poke(dut.enq.valid, 1.B)
    _ ← step(1)
    p ← peek(dut.deq.valid)
} yield p.litValue ═ 1
```



SimCommand is **> 10x faster** than chiseltest's built-in fork/join threading

## SimCommand Interpreter



- On each timestep
  - Run every thread until a step, join, or return
  - Collect any new threads spawned
  - Repeat until a fixpoint is reached
- Step the clock
- Repeat until the main thread returns

## SimCommand Example - UART

```
def sendBit(bit: Int, io: Bool): Command[Unit] = for {
    _ ← poke(io, bit.B)
    _ ← step(cyclesPerBit)
} yield ()

def sendByte(byte: Int, io: Bool): Command[Unit] = for {
    _ ← sendBit(0)
    _ ← concat((0 until 8).map(
        i ⇒ sendBit((byte >> i) & 0×1, io)))
    _ ← sendBit(1)
} yield ()
```

## SimCommand Channels

Channels enable deterministic thread-to-thread communication, similar to SystemVerilog mailboxes or golang channels.

```
ch ← makeChannel[Integer](size=10)
_ ← put(ch, 1)
t ← fork(for { v ← getBlocking(ch) } yield v)
v ← join(t)
```

- Threads can block on a channel having data to receive
- Channels enable UVM-style port/export message sharing

## SimCommand Imperative Interpreter

- Older purely recursive interpreter used no mutable data structures and required unnecessary allocations
- New imperative interpreter maintains an explicit heap-allocated stack per thread and **improves performance by 30%**

## SimCommand Tail Recursion Primitive

- Naive monadic recursion can blow up the stack
- One solution is *trampolining* where the stack is dumped to the heap on every recursive call.

```
def tailRecM[S, R](s: S)
  (f: S ⇒ Command[Either[S, R]]): Command[R] = {
  f(s).flatMap {
    case Left(value) ⇒ tailRecM(value)(f)
    case Right(value) ⇒ lift(value)
  }
}
```

However, trampolining is slow. A recursion *primitive* that elides trampolining demonstrates **400% performance improvement**.

## Testing Commands with Commands

- Often, we want to test a Command *itself* (i.e. unit test a VIP)
- Traditionally, this would be difficult without running RTL simulation with a DUT
- However, since Commands are *pure descriptions*, they can be interpreted in RTL simulation *or* stand alone

```
// bindings mimic DUT IOs
val uartPin: Bool = binding(init=1.B)
val test = for {
  _ ← fork(sendByte(0×8d.U, uartPin))
  t ← fork(receiveByte(uartPin))
  v ← join(t)
} yield v
assert(run(test) ═ 0×8d)
```

## Thread Order Dependency Detection

- Multi-threaded chiseltest has support for detecting read-after-write conditions through combinational paths in the DUT from 2 different threads
- SimCommand now also supports dependency detection and will throw an exception unless thread order is manually defined. The performance overhead is marginal.

```
test(new Queue(UInt(8.W), flow=true)) { dut ⇒
  val test = for {
    e ← fork(enqueue(dut.enq, 4.U))
    d ← fork(dequeue(dut.deq))
    _ ← joinall(e, d)
  }.run() // will throw an exception
}
```

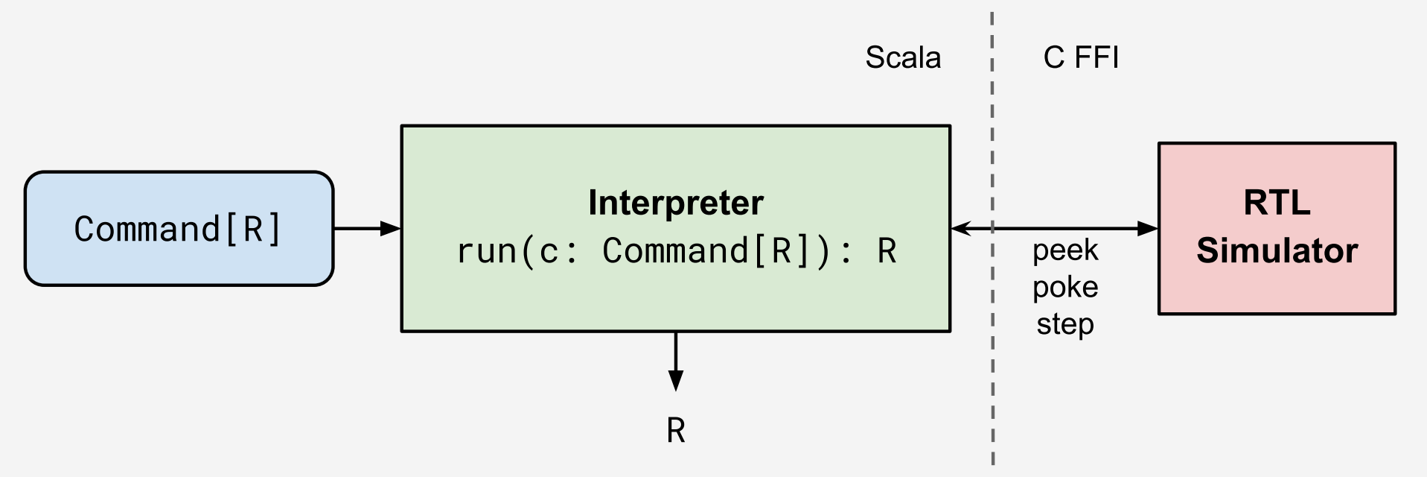
## Opportunistic Cycle Skipping

```
val t1 = for {
  _ ← step(10)
}
val t2 = for {
  _ ← step(1)
  _ ← step(5)
}
```

```
for {
  _ ← fork(t1)
  _ ← fork(t2)
  _ ← joinall(t1, t2)
}
```

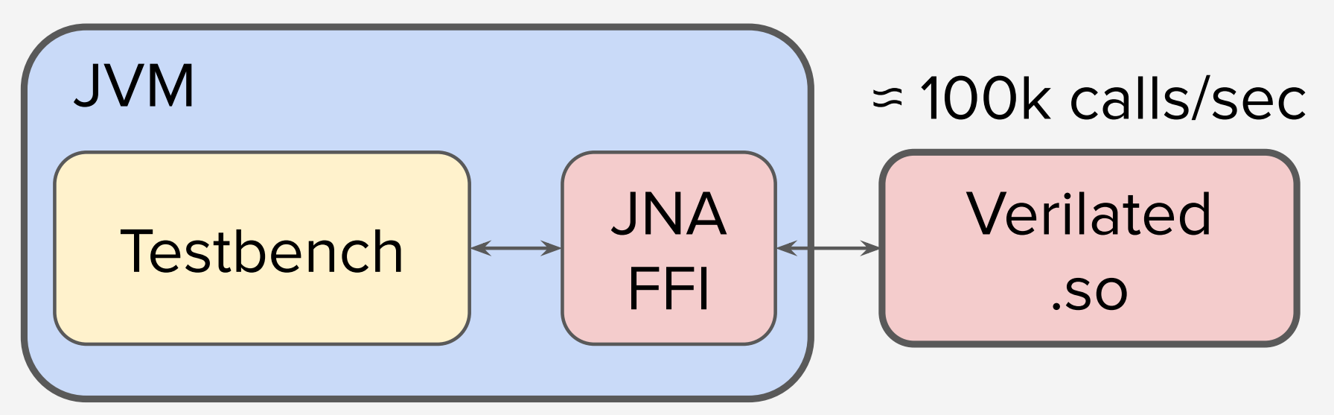
- The interpreter will step as many cycles as possible at a time (in this example: 1, 4, 5) and will only wake up threads that are ready to run
- A system-level testbench that took 25 minutes with chiseltest, now takes 5 seconds (faster than SystemVerilog + VCS too)

## Chiseltest ↔ RTL Simulator FFI

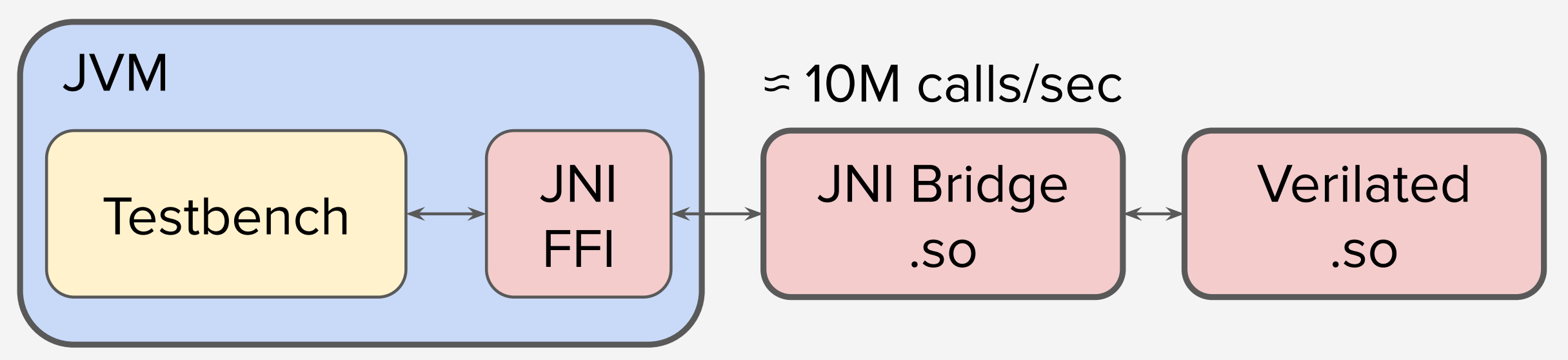


- The foreign function interface (FFI) linking chiseltest on the JVM and the RTL simulator (Verilator) in C++ is a bottleneck

**JNA**



**JNI**



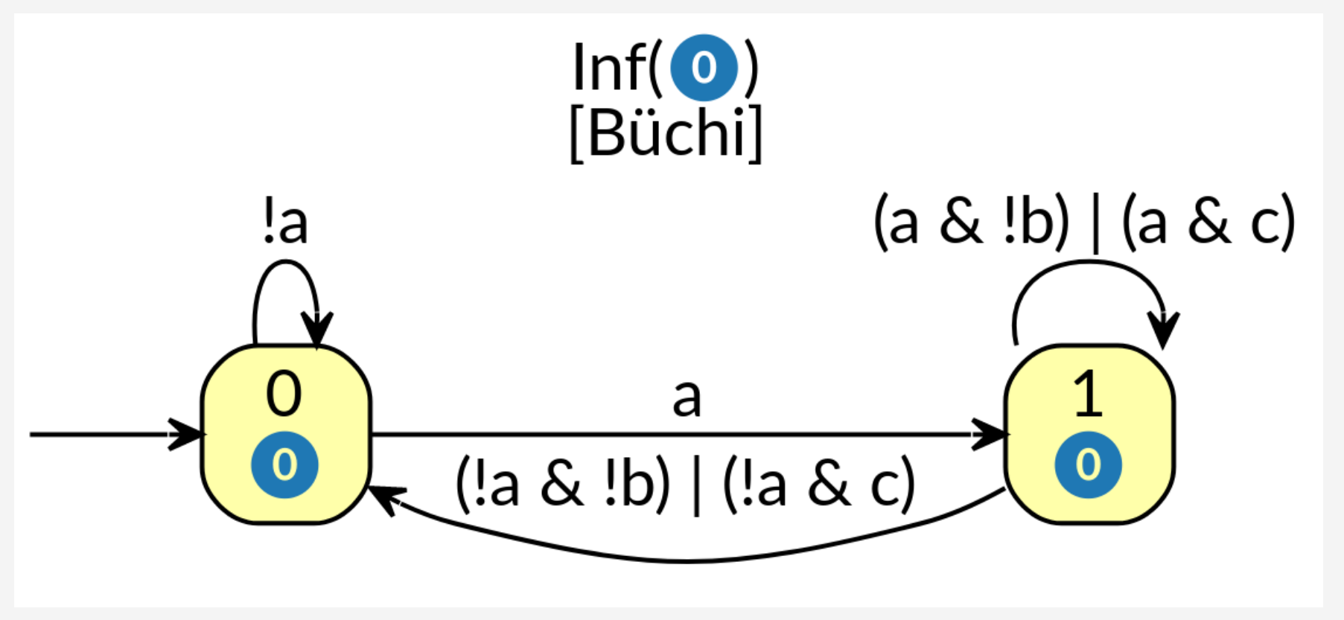
- Using JNI vs JNA as the FFI API **improves performance by 10x**
- Integrated into chiseltest

## "SVA" for Chisel

- Sequence library to describe temporal properties in Chisel
- Supports a subset of linear PSL (sequence combinators + bounded delays + LTL operators)
- 2 backends: naive monitor automata synthesis, SPOT-driven optimized automata synthesis

```
class Example extends Module {
  val a, b = Reg(Bool())
  val c = Reg(UInt(8.W))
  assert(a ⟾ b)
  assert(a ###1 b ⟾ (c > 15.U))
}
```

```
PSL: G((a & X b) -> X c)
```



github.com/ekiwi/chisel-sequences