MAMBA: CLOSING THE PERFORMANCE GAP IN PRODUCTIVE HARDWARE DEVELOPMENT FRAMEWORKS

Shunning Jiang, Berkin Ilbeyi, Christopher Batten
School of Electrical and Computer Engineering
Cornell University
Traditional hardware description language
- Example: Verilog

✓ Fast edit-debug-sim loop
✓ Single language for design and testbench

✗ Difficult to parameterize
✗ Require specific ways to build powerful testbench

* HDL: hardware description language
* DUT: design under test
* TB: test bench
* synth: synthesis
Traditional hardware description language - Example: Verilog

- Fast edit-debug-sim loop
- Single language for design and testbench
- Difficult to parameterize
- Require specific ways to build powerful testbench

Chisel → Verilog
SystemVerilog
C++ → Verilog → PyMTL → Verilog

~12 GRAD STUDENTS TAPE OUT CELERITY IN 9 MONTHS

**Hardware Preprocessing Framework (HPF)**

Traditional hardware description language
- Example: Verilog

- ✓ Fast edit-debug-sim loop
- ✓ Single language for design and testbench
- ✗ Difficult to parameterize
- ✗ Require specific ways to build powerful testbench

Hardware preprocessing framework (HPF)
- Example: Genesis2

- ✓ Better parametrization with insignificant coding style change
- ✗ Multiple languages create semantic gap
- ✗ Still difficult to build powerful testbench
**Hardware Generation Framework (HGF)**

**Traditional hardware description language**
- Example: Verilog

- ✔ Fast edit-debug-sim loop
- ✔ Single language for design and testbench
- ✗ Difficult to parameterize
- ✗ Require specific ways to build powerful testbench

**Hardware preprocessing framework (HPF)**
- Example: Genesis2

- ✔ Better parametrization with insignificant coding style change
- ✗ Multiple languages create semantic gap
- ✗ Still difficult to build powerful testbench

**Hardware generation framework (HGF)**
- Example: Chisel

- ✔ Powerful parametrization
- ✔ Single language for design
- ✗ Slower edit-debug-sim loop
- ✗ Yet still difficult to build powerful testbench (can only generate simple testbench)
Hardware Generation and Simulation Framework (HGSF)

- Example: PyMTL

- Powerful parametrization
- Single language for design and testbench
- Powerful testbench (unleash Python’s full power!)
- Fast edit-sim-debug loop
Hardware generation and simulation framework (HGSF)
- Example: PyMTL

✓ Powerful parametrization
✓ Single language for design and testbench
✓ Powerful testbench (unleash Python's full power!)
✓ Fast edit-sim-debug loop

Sad fact: The loop is only fast when simulating a small amount of cycles on a small design!
Closing the performance gap in HGSFs

- Understanding the performance gap
- Background on tracing JIT compiler
- Co-optimizing the JIT and the HGSF
- Mamba performance

Hardware generation and simulation framework (HGSF)
- Example: PyMTL
We implement a 64-bit radix-four iterative divider to the same level of detail in all frameworks using control/datapath split.

Higher is better.

Log scale – the gap is larger than it seems.
• CVS is 20X faster than Icarus
• Verilator requires C++ testbench, only works with synthesizable code, takes time to compile, but is 200+X faster than Icarus
Chisel (HGF) generates Verilog and simulates Verilog – the same performance!
• Using CPython interpreter, Python-based HGSFs are much slower than CVS and even 10X slower than Icarus
• Simply applying unmodified PyPy JIT interpreter brings ~10X speedup for Python-based HGSFs, but they are still significantly slower than CVS
Hybrid C/C++ cosimulation improves the performance but:
- Only works with a subset of code
- May require the user to work with C/C++ and Python at the same time
Hybrid C/C++ cosimulation improves the performance but:
• Only works with a subset of code
• May require the user to work with C/C++ and Python at the same time.
SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER

We need an HGSF that provides fast simulation performance within a single high-level language.
Closing the performance gap in HGSFs

- Understanding the performance gap
- Background on tracing JIT compiler
- Co-optimizing the JIT and the HGSF
- Mamba performance

Hardware generation and simulation framework (HGSF)
- Example: PyMTL
Dynamic languages provide vast productivity features. As a result, they require interpreter. (e.g. CPython)
Dynamic languages provide vast productivity features. As a result, they require interpreter. (e.g. CPython)

However, interpreters are slow.

Just-in-time (JIT) compiler addresses the performance gap.
def max(a, b):
    if a > b:
        return a
    else:
        return b

# This is a hot loop
for i in xrange(10000000):
    ... = max( ..., ... )

# The first trace is generated
# when integers are passed as args
# and a is actually greater than b
guard_type(a, int)  # type check
guard_type(b, int)  # type check
c = int_gt(a, b)    # check if a>b
guard_true(c)
return(a)
# This is a hot loop
for i in xrange(10000000):
    ... = max(..., ...)

def max(a, b):
    if a > b:
        return a
    else:
        return b

# The first trace is generated
# when integers are passed as args
# and a is actually greater than b
guard_type(a, int)  # type check
guard_type(b, int)  # type check
c = int_gt(a, b)    # check if a>b
guard_true(c)
return(a)

# bridge out of guard_true(c)
# The second trace is generated
# when guard_true(c) fails
return(b)
# This is a hot loop
for i in xrange(10000000):
    ... = max(..., ...)

def max(a, b):
    if a > b:
        return a
    else:
        return b

# bridge out of guard_type(a, int)
# The third trace is generated
# when floats are passed as args
guard_type(a, float)  # type check
guard_type(b, float)  # type check
c = float_gt(a, b)    # check if a>b
guard_true(c)
return(a)

# bridge out of guard_true(c)
# The second trace is generated
# when guard_true(c) fails
return(b)
Closing the performance gap in HGSFs

- Understanding the performance gap
- Background on tracing JIT compiler
- Co-optimizing the JIT and the HGSF
- Mamba performance

Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL
**CHALLENGES OF HGSFs ON TRACING JIT**

- By nature, event-driven simulation is bad for tracing JIT
- Control flows in logic blocks turn into guards that fail often
- Emulating fix-width data types using Python’s seamless BigInt is not the most efficient
- …
CHALLENGES: EVENT-DRIVEN SIMULATION

- Every signal value change check is a frequently failing guard
- Event-driven simulation’s inner loop is a bad pattern for tracing JIT
Event-driven simulation’s inner loop is a bad pattern for tracing JIT

```python
class num_cycles = 1000000
for i in xrange(num_cycles):
    while not event_queue.empty():
        block = event_queue.pop()
        block()
```
- **Event-driven simulation’s inner loop is a bad pattern for tracing JIT**

```python
num_cycles = 1000000
for i in xrange(num_cycles):
    while not event_queue.empty():
        block = event_queue.pop()
        block()
```

- # The first trace is for blk1
  ```python
guard_equal(block, blk1)
< execute the code of blk1 >
jump_to_loop(while_loop)
```

- # The second trace is for blk2
  ```python
guard_equal(block, blk2)
< execute the code of blk2 >
jump_to_loop(while_loop)
```

- # The third trace is for blk3
  ```python
guard_equal(block, blk3)
< execute the code of blk3 >
jump_to_loop(while_loop)
```
Event-driven simulation’s inner loop is a bad pattern for tracing JIT

N-th block will fail N-1 times to find the trace. In total it is O(N^2) for N blocks and is the scaling bottleneck.
Challenges: Emulating Fix-Width Data Types

- Emulating fix-width data types using Python integer is not the most efficient
  - Python seamlessly promote integer to BigInt when overflowing 63-bit
  - However, each overflow is a guard failure
  - A 100-bit signal can either be BigInt or integer

- We actually know each signal’s bitwidth during elaboration!
- How can we tell JIT engine this information?
**Mamba**

- Mamba is a set of techniques that improve simulation performance by co-optimizing the meta-tracing JIT and the HGSF.
  - Goal:
    - Minimize the total number of generated traces
    - Minimize the total size of generated traces
    - Minimize the effect of having too many traces
<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF + Static Scheduling</strong></td>
<td>13×</td>
</tr>
</tbody>
</table>

```python
num_cycles = 1000000
for i in xrange(num_cycles):
    while not event_queue.empty():
        block = event_queue.pop()
        block()

for i in xrange(num_cycles):
    for block in static_schedule:
        block()
```
### Mamba Techniques/Performance (All with PyPy)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF</strong></td>
<td></td>
</tr>
<tr>
<td>+ Static Scheduling</td>
<td>13×</td>
</tr>
<tr>
<td>+ Schedule Unrolling</td>
<td>16×</td>
</tr>
</tbody>
</table>

```python
num_cycles = 1000000
for i in xrange(num_cycles):
    while not event_queue.empty():
        block = event_queue.pop()
        block()
```

```python
for i in xrange(num_cycles):
    for block in static_schedule:
        block()
```

```python
for i in xrange(num_cycles):
    block1(); block2(); block3();
    ...; blockN();
```
# Mamba Techniques/Performance (All with PyPy)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF</strong></td>
<td></td>
</tr>
<tr>
<td>+ Static Scheduling</td>
<td>13×</td>
</tr>
<tr>
<td>+ Schedule Unrolling</td>
<td>16×</td>
</tr>
<tr>
<td>+ Heuristic Toposort</td>
<td>18×</td>
</tr>
</tbody>
</table>

```python
num_cycles = 1000000
for i in xrange(num_cycles):
    while not event_queue.empty():
        block = event_queue.pop()
        block()

for i in xrange(num_cycles):
    for block in static_schedule:
        block()

for i in xrange(num_cycles):
    block1(); block2(); block3(); ...; blockN();

for i in xrange(num_cycles):
    block3(); block1(); block4(); block2(); ...
```
## Mamba Techniques/Performance (All with PyPy)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF</strong></td>
<td></td>
</tr>
<tr>
<td>+ Static Scheduling</td>
<td>13×</td>
</tr>
<tr>
<td>+ Schedule Unrolling</td>
<td>16×</td>
</tr>
<tr>
<td>+ Heuristic Toposort</td>
<td>18×</td>
</tr>
<tr>
<td>+ Trace Breaking</td>
<td>19×</td>
</tr>
</tbody>
</table>

```python
for i in xrange(num_cycles):
    block3();
    block1();
    jit_break_trace()
    block4();
    block2(); ...
```
"Letting the generate-purpose JIT recognize RTL simulation constructs" – As a proof of concept, we implement fix-bitwidth data types in RPython framework.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>24K CPS</td>
</tr>
<tr>
<td><strong>JIT-Aware HGSF</strong></td>
<td></td>
</tr>
<tr>
<td>+ Static Scheduling</td>
<td>13×</td>
</tr>
<tr>
<td>+ Schedule Unrolling</td>
<td>16×</td>
</tr>
<tr>
<td>+ Heuristic Toposort</td>
<td>18×</td>
</tr>
<tr>
<td>+ Trace Breaking</td>
<td>19×</td>
</tr>
<tr>
<td>+ Consolidation</td>
<td>27×</td>
</tr>
<tr>
<td><strong>HGSF-Aware JIT</strong></td>
<td></td>
</tr>
<tr>
<td>+ RPython Constructs</td>
<td>96×</td>
</tr>
</tbody>
</table>
We use Linux `perf` tool to identify microarchitectural bottlenecks.

For larger designs (unrolled into a huge loop body), the instruction TLB becomes the bottleneck.
CLOSING THE PERFORMANCE GAP IN HGSFs

- Understanding the performance gap
- Background on tracing JIT compiler
- Co-optimizing the JIT and the HGSF
- Mamba performance

Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL
CASE STUDY: SIMULATING RISC-V MULTICORE

Simulated Design:
- 1 / 2 / 4 / 8 / 16 / 32 RV32IM five-stage pipeline processors hooked up to a multi-port test memory
- No cache, no on-chip network, just 32 processors
- Running a parallel C++ matrix multiplication program

Competitors:
- Mamba
- Verilator, Icarus Verilog, CVS
- PyMTL, PyMTL-CSim
Simulating 1-core

Simulating 32-core

Average Cycle Per Second = \frac{\text{Simulated cycle}}{\text{Compilation time} + \text{Startup Overhead} + \text{Simulation time}}
PERFORMANCE (w/ Compilation and Startup Overheads)

Simulating 1-core

Simulating 32-core

Average Cycle Per Second (CPS)

Average Cycle Per Second = \frac{\text{Simulated cycle}}{\text{Compilation time} + \text{Startup Overhead} + \text{Simulation time}}
PERFORMANCE (W/ COMPILATION AND STARTUP OVERHEADS)

Simulating 1-core

Simulating 32-core

Average Cycle Per Second = \frac{\text{Compilation time} + \text{Startup Overhead} + \text{Simulation time}}{\text{Simulated cycle}}
PERFORMANCE (WITH COMPILATION AND STARTUP OVERHEADS)

Simulating 1-core

Simulating 32-core

Average Cycle Per Second = \frac{\text{Simulated cycle}}{\text{Compilation time} + \text{Startup Overhead} + \text{Simulation time}}
Deeply co-optimizing the HGSF and the underlying general-purpose JIT is the key to achieve an order of magnitude speedup.

Proposed techniques also shed light on performance optimizations in existing hardware generation and simulation frameworks.

- [https://github.com/cornell-brg/pymtl](https://github.com/cornell-brg/pymtl)

This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, and a donation from Intel.