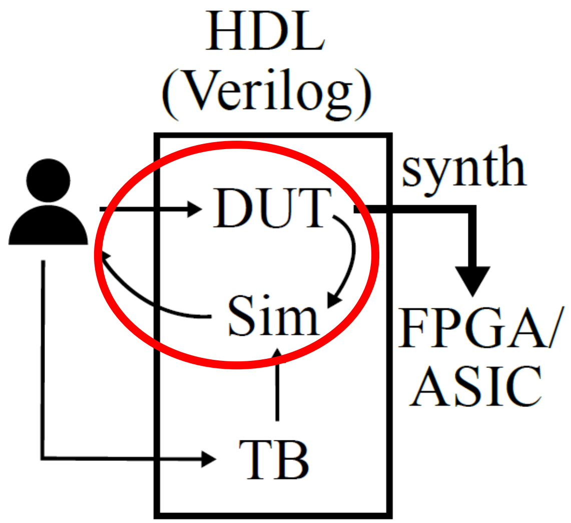
Cornell University
Computer Systems Laboratory

Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks

Shunning Jiang, Berkin Ilbeyi, Christopher Batten
School of Electrical and Computer Engineering
Cornell University

THE TRADITIONAL FLOW



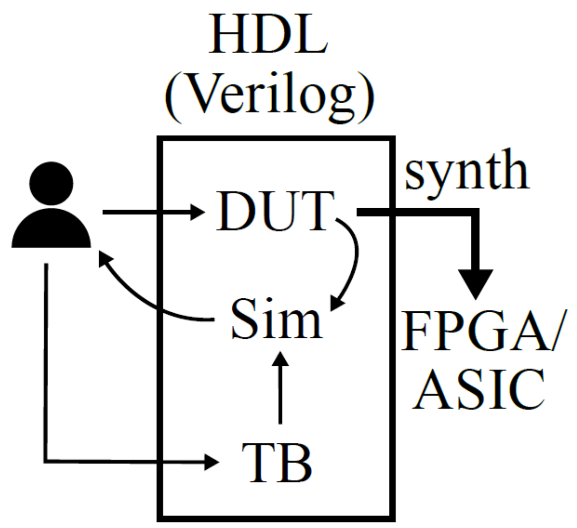
Traditional hardware description language
- Example: Verilog

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

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

* HDL: hardware description language
* DUT: design under test
* TB: test bench
* synth: synthesis

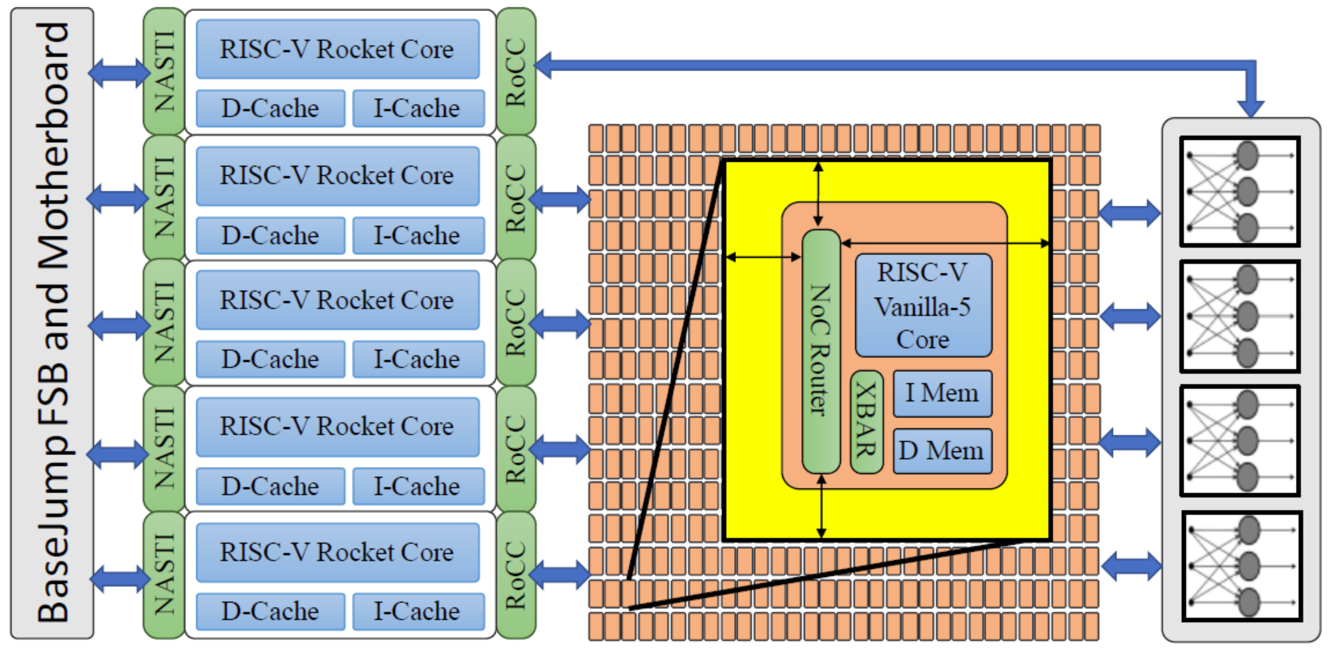
~12 GRAD STUDENTS TAPED OUT CELERITY IN 9 MONTHS



Traditional hardware description language
- Example: Verilog

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

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

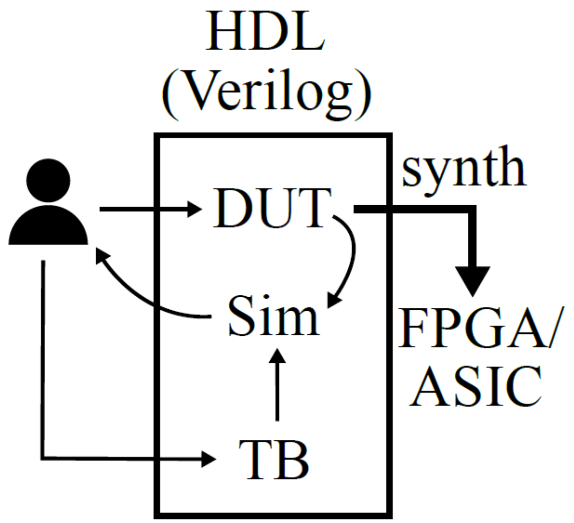


General-Purpose Tier | **Massively Parallel Tier** | **Specialization Tier**

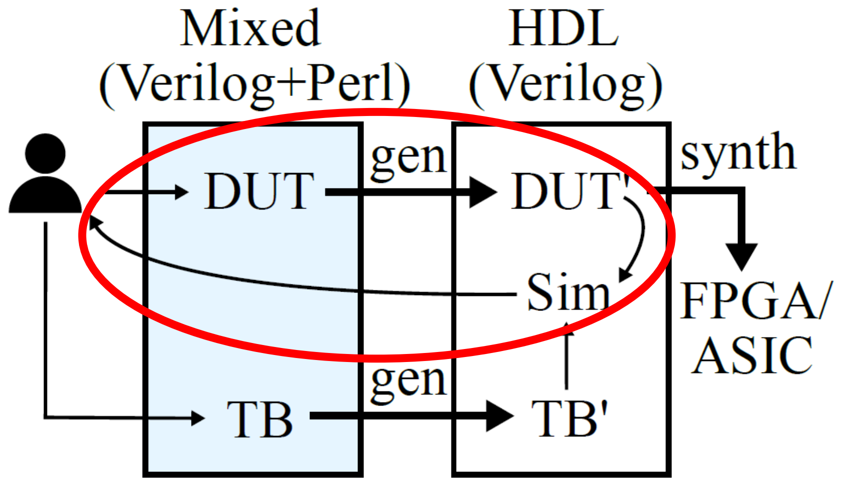
Chisel → Verilog | SystemVerilog | **C++** → Verilog → **PyMTL** → Verilog

Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael B. Taylor. **"The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips."** *IEEE Micro*, 38(2):30–41, Mar/Apr. 2018. (special issue for top picks from HOTCHIPS-29)

Hardware Preprocessing Framework (HPF)



Traditional hardware description language
- Example: Verilog



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

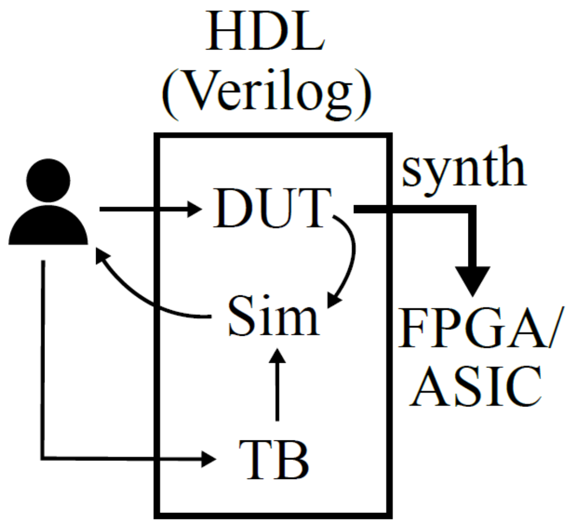
✓ Fast edit-debug-sim loop
✓ Single language for design and testbench

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

✓ Better parametrization with insignificant coding style change

X Multiple languages create semantic gap
X Still difficult to build powerful testbench

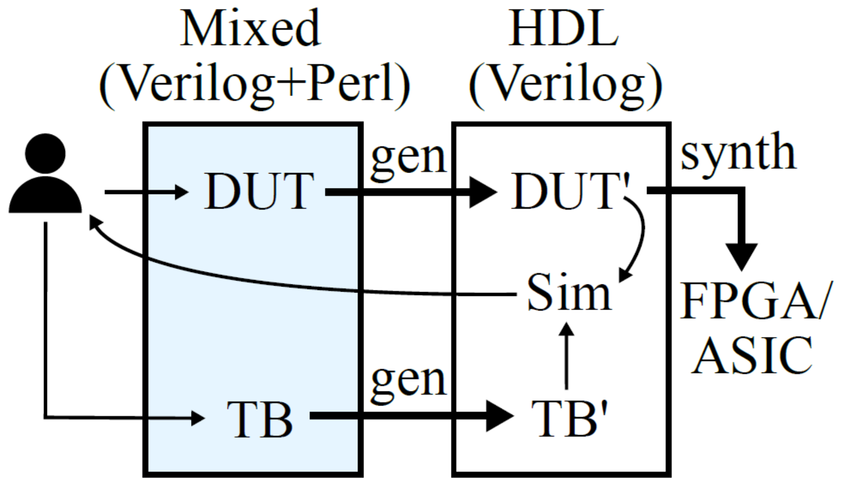
HARDWARE GENERATION FRAMEWORK (HGF)



Traditional hardware description language
- Example: Verilog

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

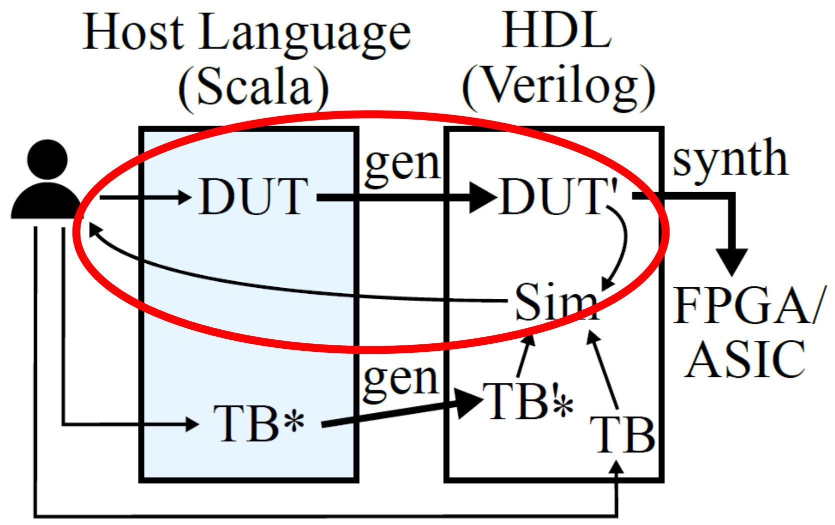
X Difficult to parameterize
X Require specific ways to build powerful testbench



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

✓ Better parametrization with insignificant coding style change

X Multiple languages create semantic gap
X Still difficult to build powerful testbench

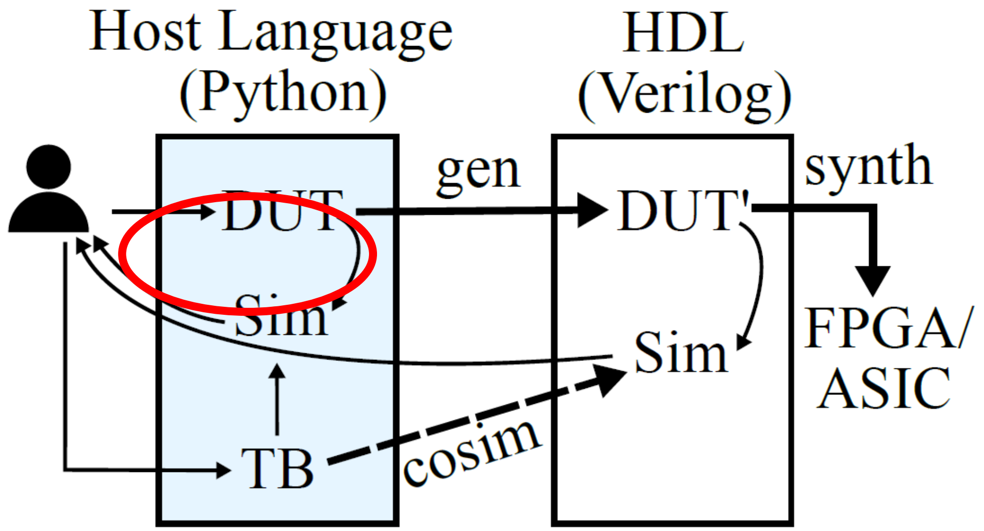


Hardware *generation* framework (HGF)
- Example: Chisel

✓ Powerful parametrization
✓ Single language for design

X Slower edit-debug-sim loop
X Yet still difficult to build powerful testbench (can only generate simple testbench)

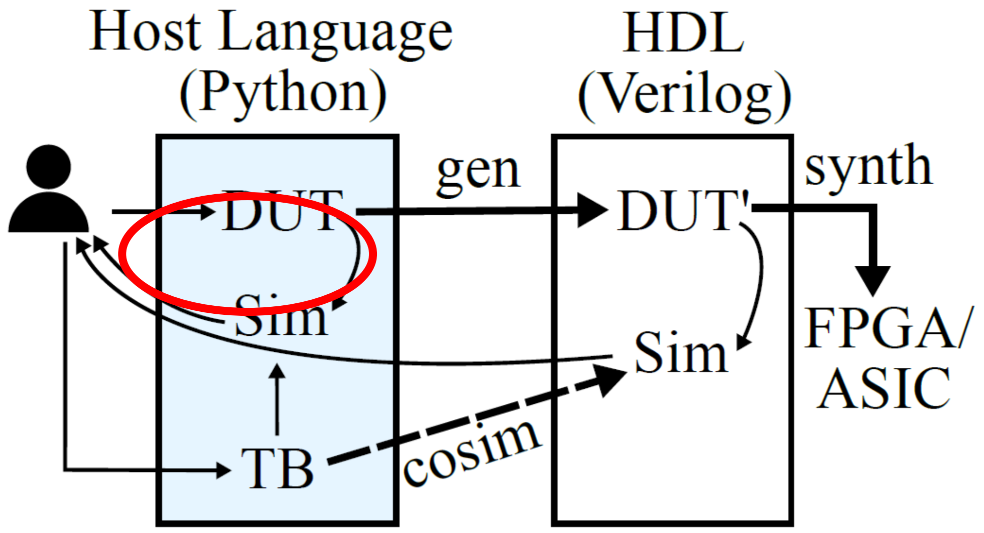
HARDWARE GENERATION AND SIMULATION FRAMEWORK (HGSF)



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)

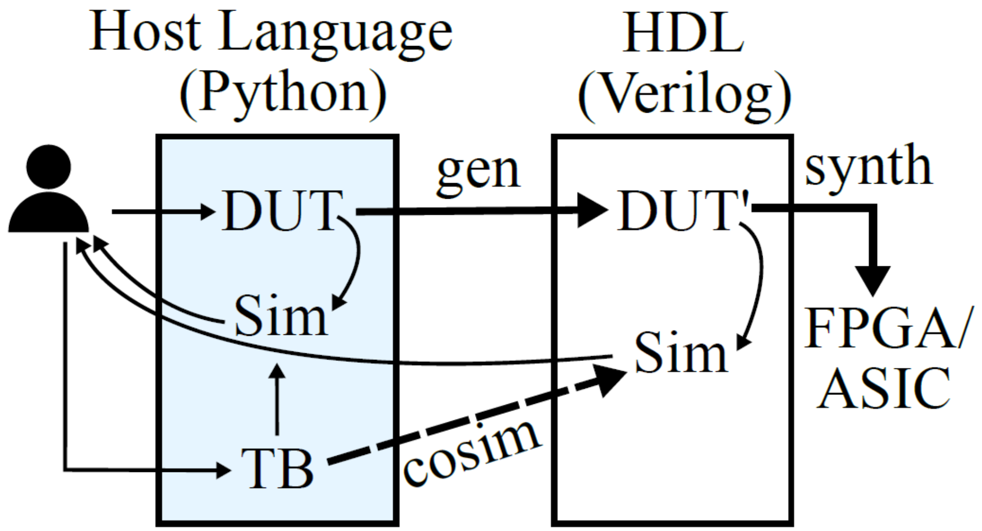


Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

- ✓ 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!

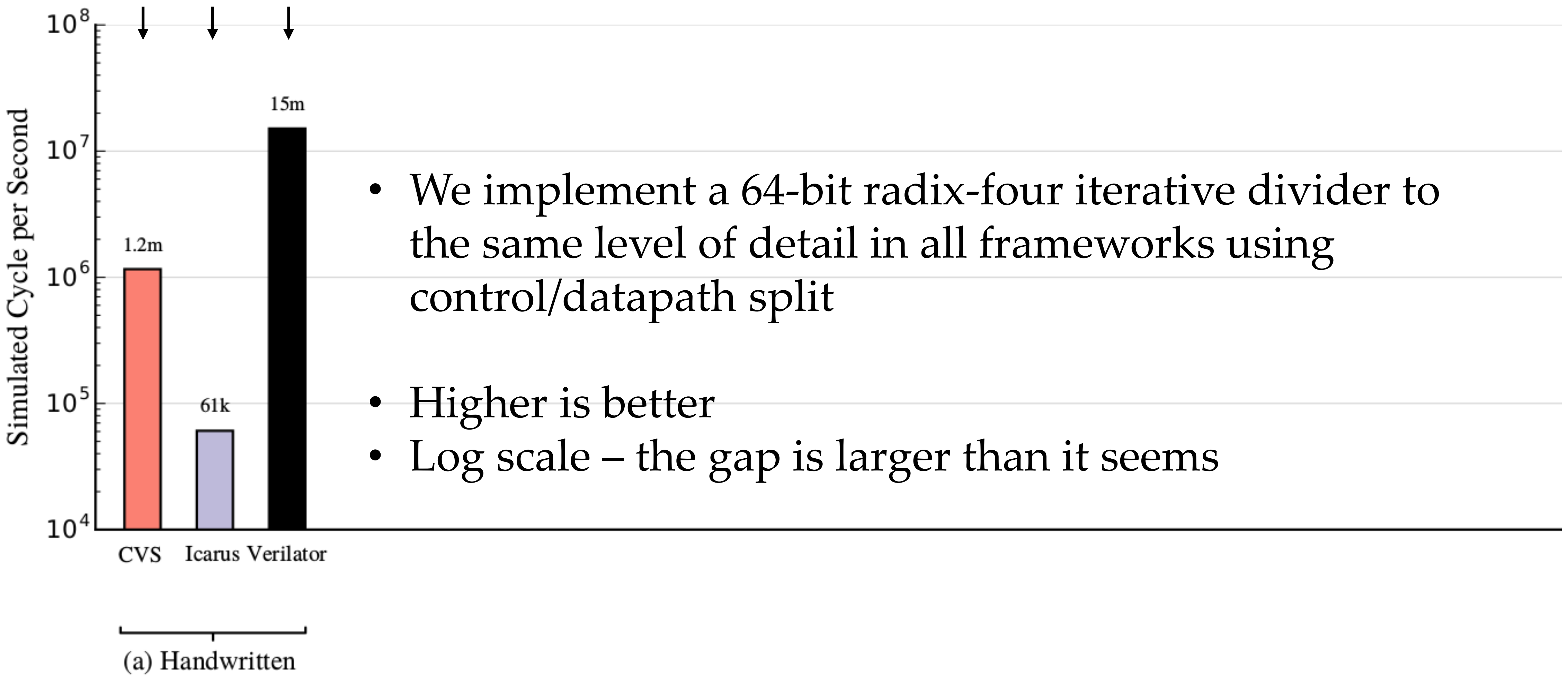
CLOSING THE PERFORMANCE GAP IN HGSFS



Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

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

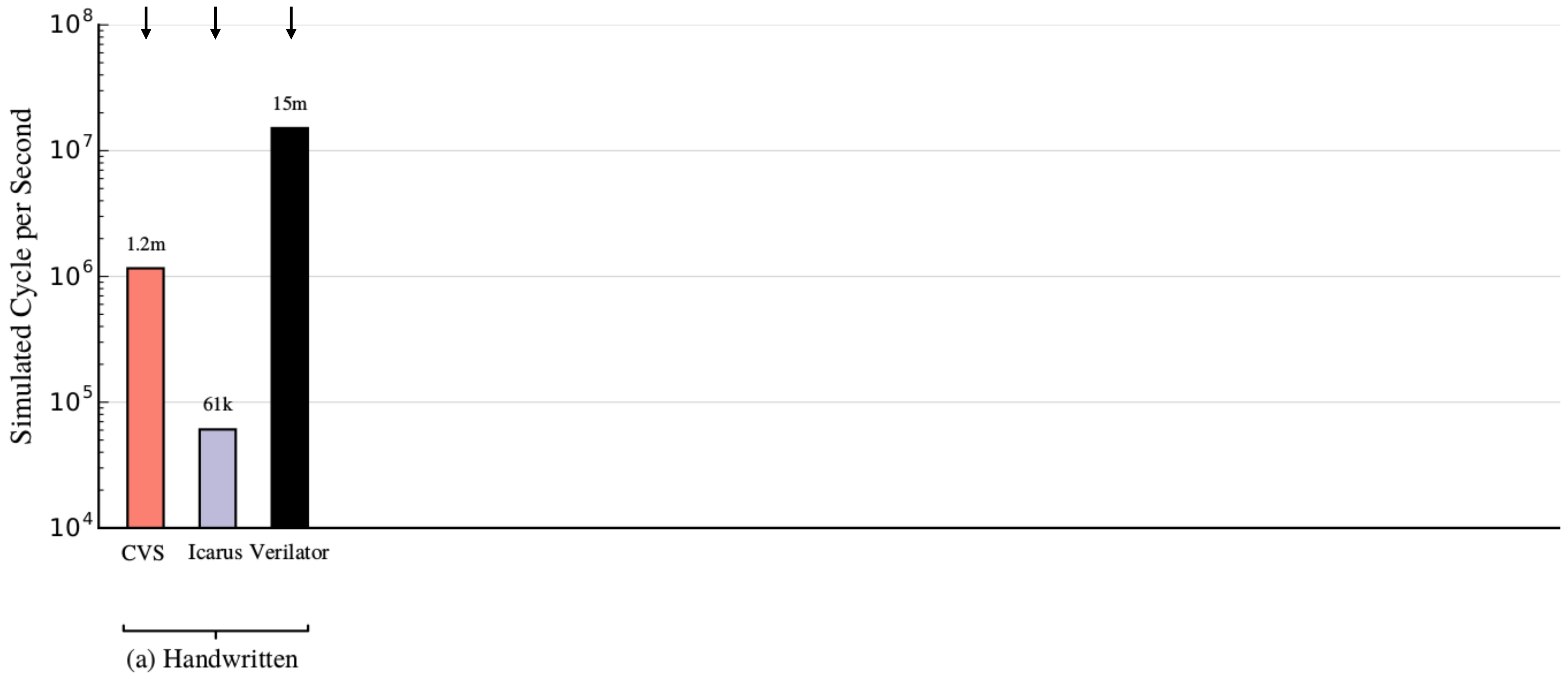
Simulation performance of 64-bit iterative divider



- We implement a 64-bit radix-four iterative divider to the same level of detail in all frameworks using control/datapath split

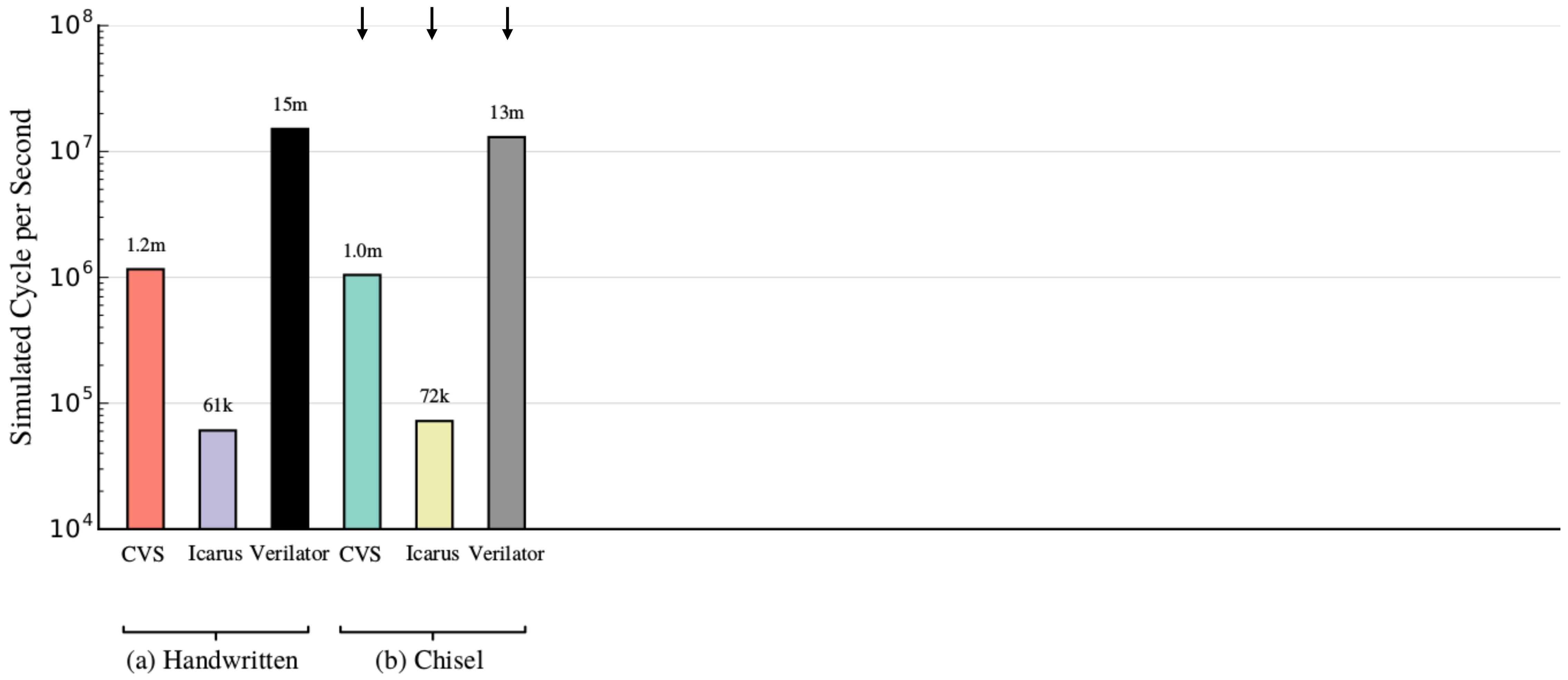
- Higher is better
- Log scale – the gap is larger than it seems

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER



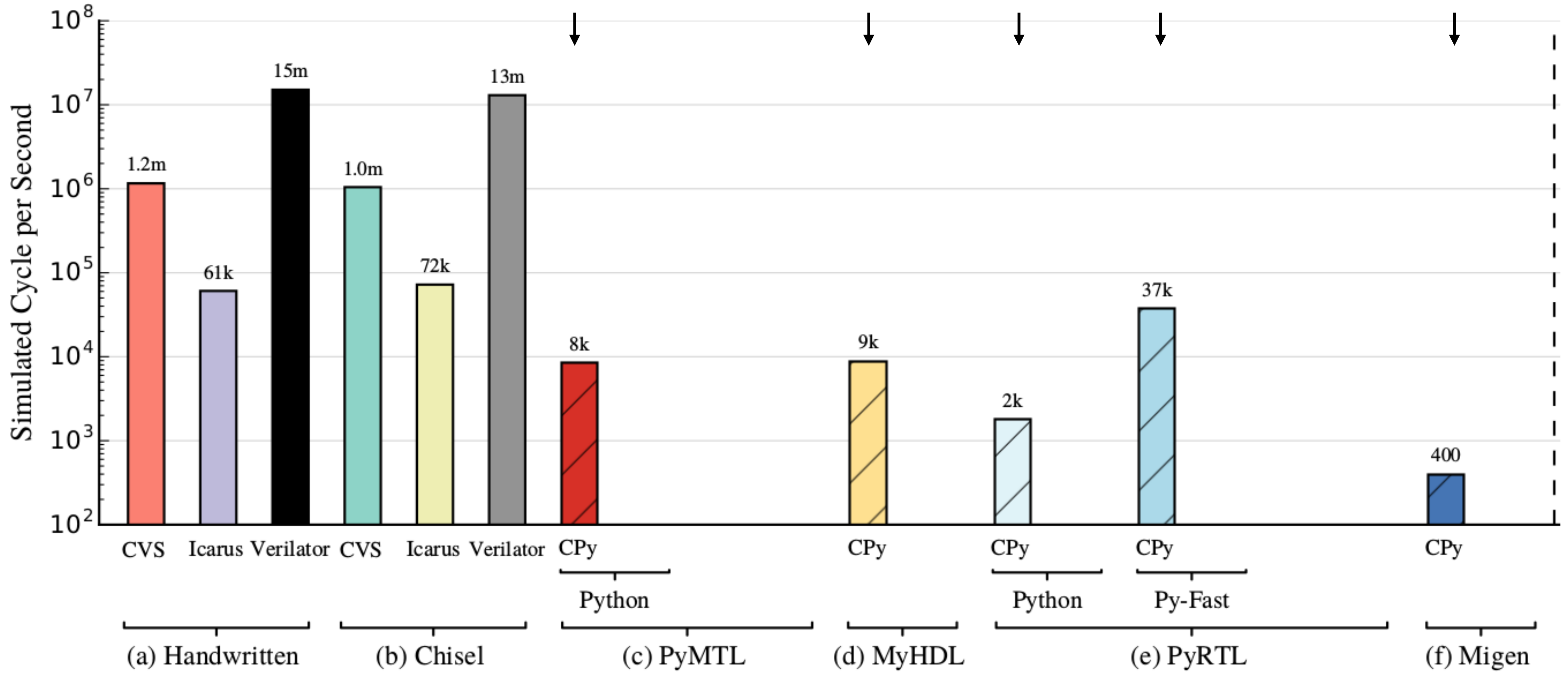
- CVS is 20X faster than Icarus
- Verilator requires C++ testbench, only works with synthesizable code, takes time to compile, but is 200+X faster than Icarus

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER



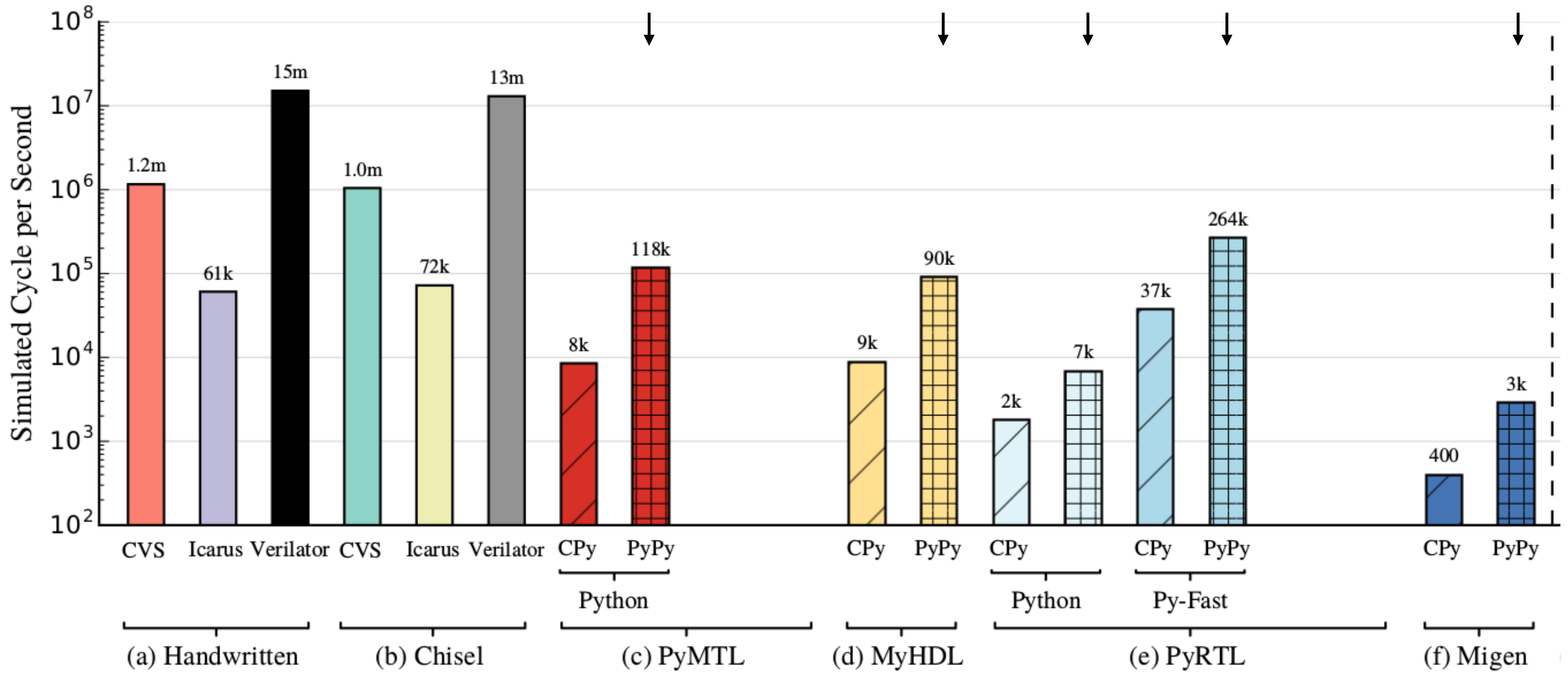
- Chisel (HGF) generates Verilog and simulates Verilog – the same performance!

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER



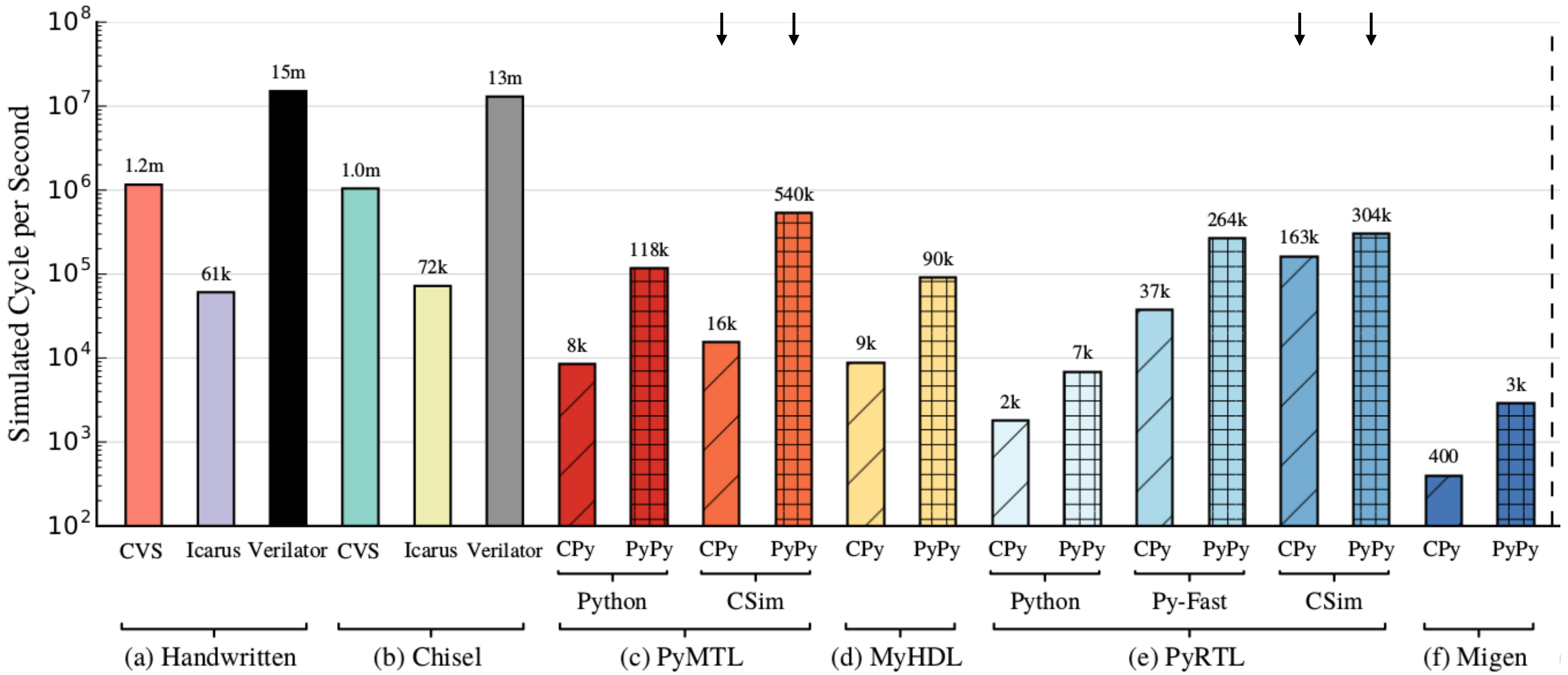
- Using CPython interpreter, Python-based HGSFs are much slower than CVS and even 10X slower than Icarus

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER



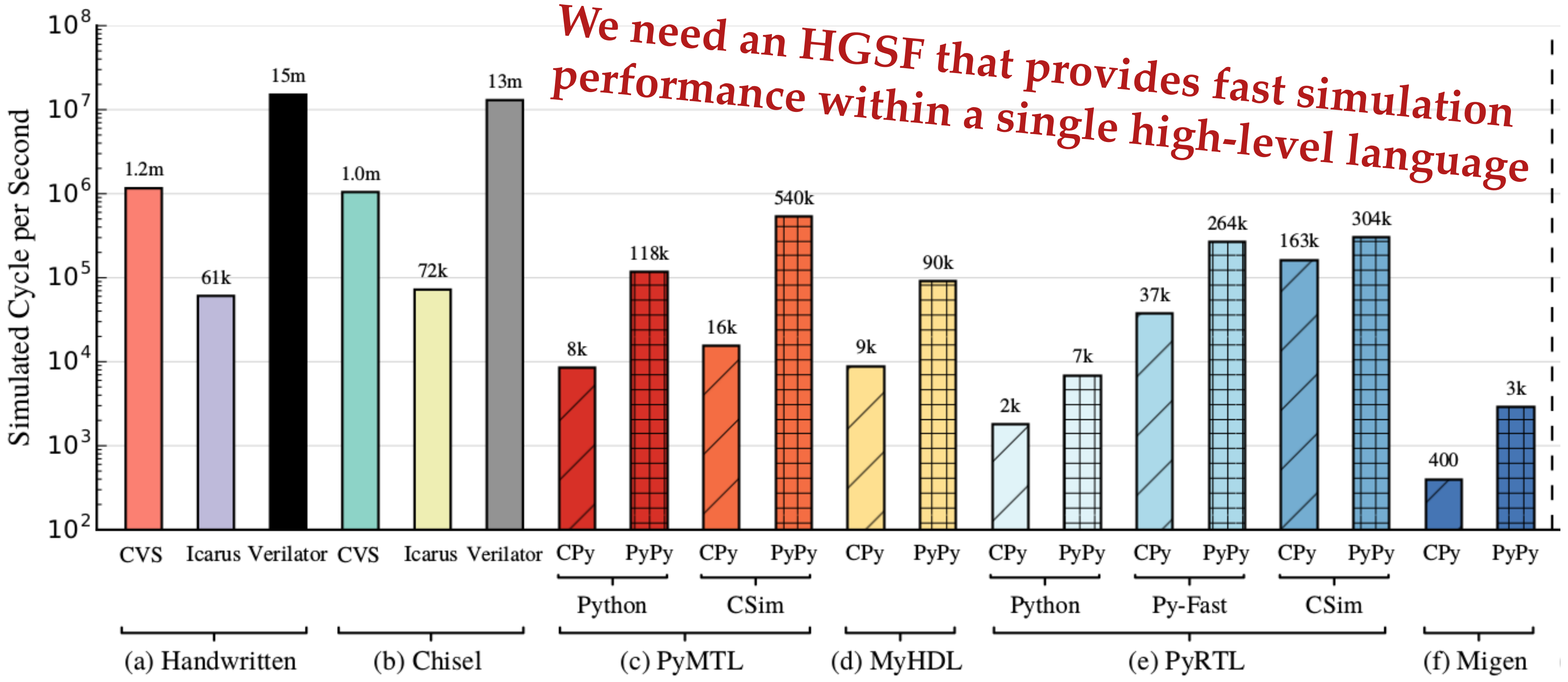
- Simply applying unmodified PyPy JIT interpreter brings ~10X speedup for Python-based HGSFs, but they are still significantly slower than CVS

Simulation performance of 64-bit iterative divider



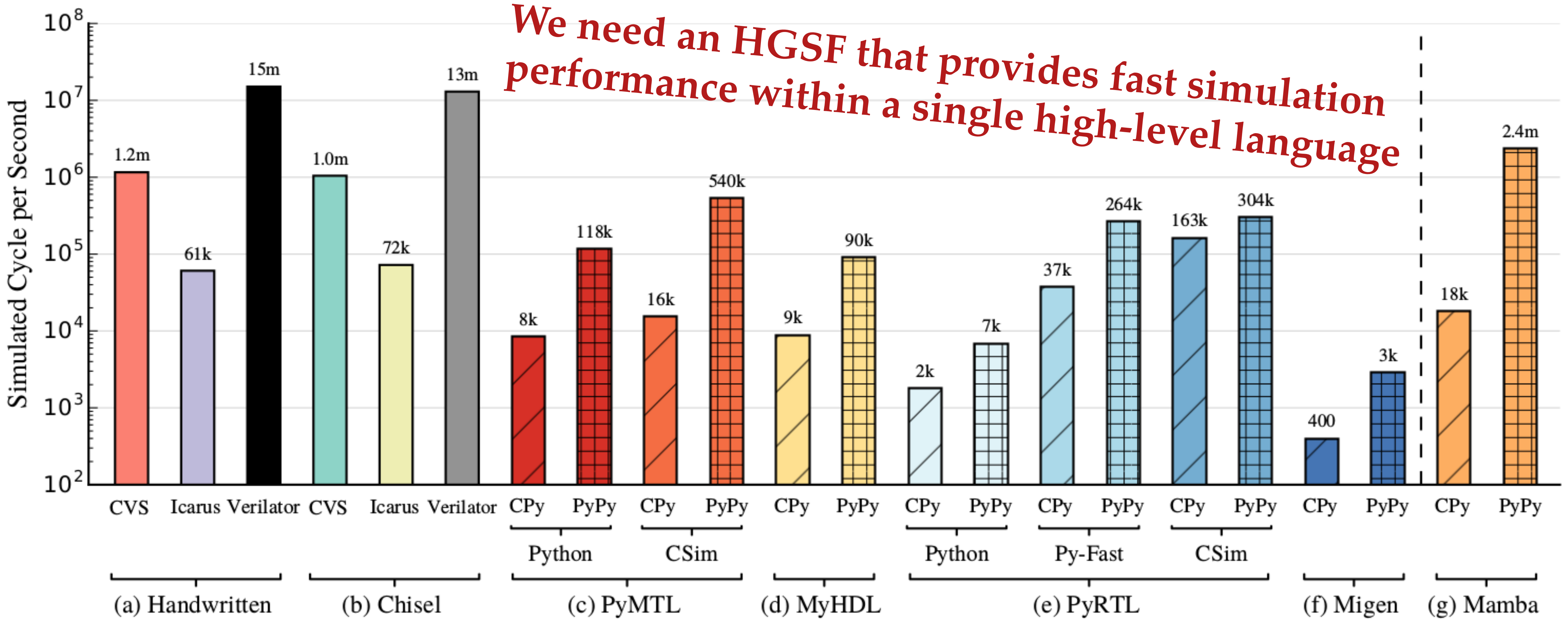
- Hybrid C/C++ cosimulation improves the performance but:
  - Only works with a subset of code
  - May require the user to work with C/C++ and Python at the same time

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER

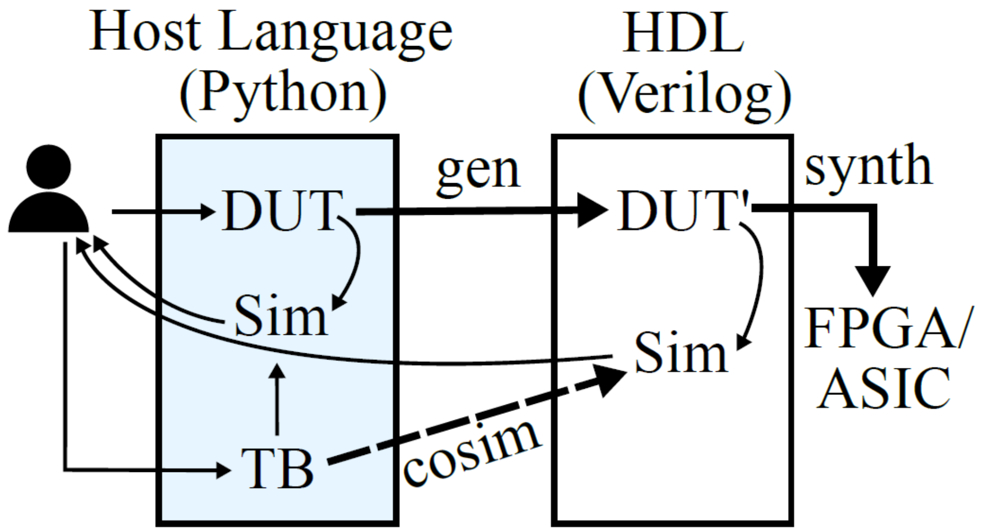


- Hybrid C/C++ cosimulation improves the performance but:
  - Only works with a subset of code
  - May require the user to work with C/C++ and Python at the same time.

SIMULATION PERFORMANCE OF 64-BIT ITERATIVE DIVIDER



CLOSING THE PERFORMANCE GAP IN HGSFS

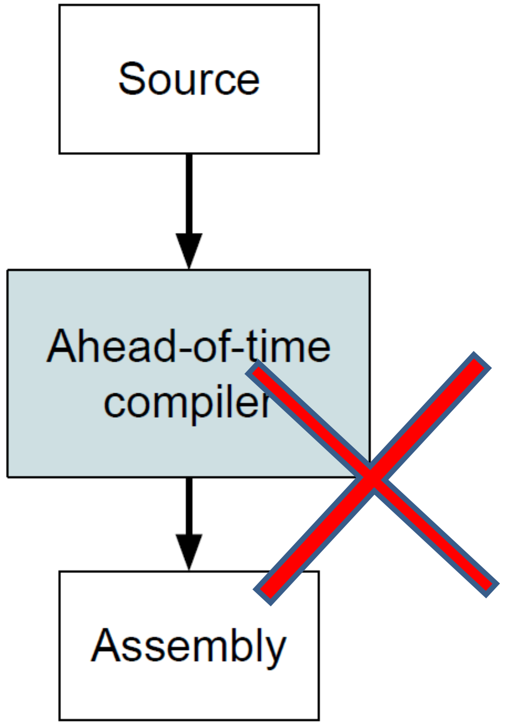


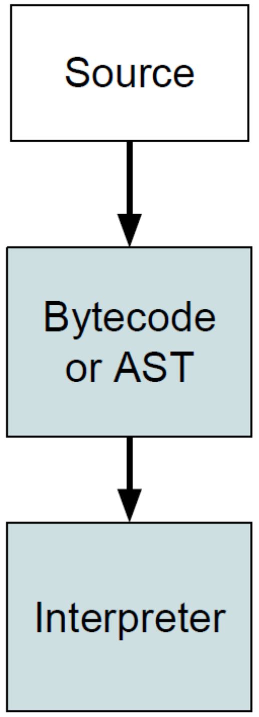
Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

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

INTERPRETER AND JUST-IN-TIME COMPILER FOR DYNAMIC LANGUAGES

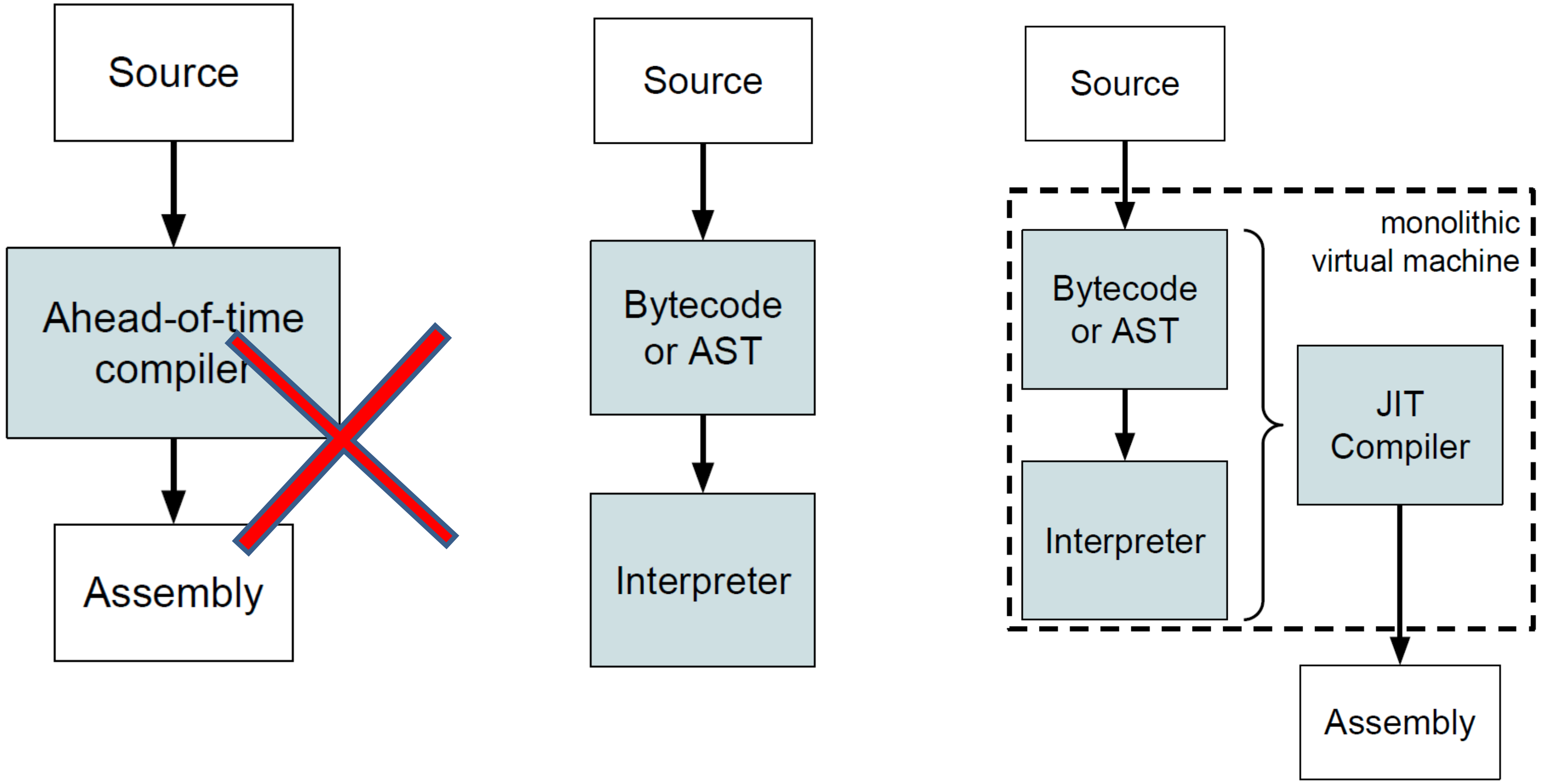
- **Dynamic languages provide vast productivity features. As a result, they require interpreter. (e.g. CPython)**





INTERPRETER AND JUST-IN-TIME COMPILER FOR DYNAMIC LANGUAGES

- 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



How tracing JIT works

```
# 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)
```

HOW TRACING JIT WORKS

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

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

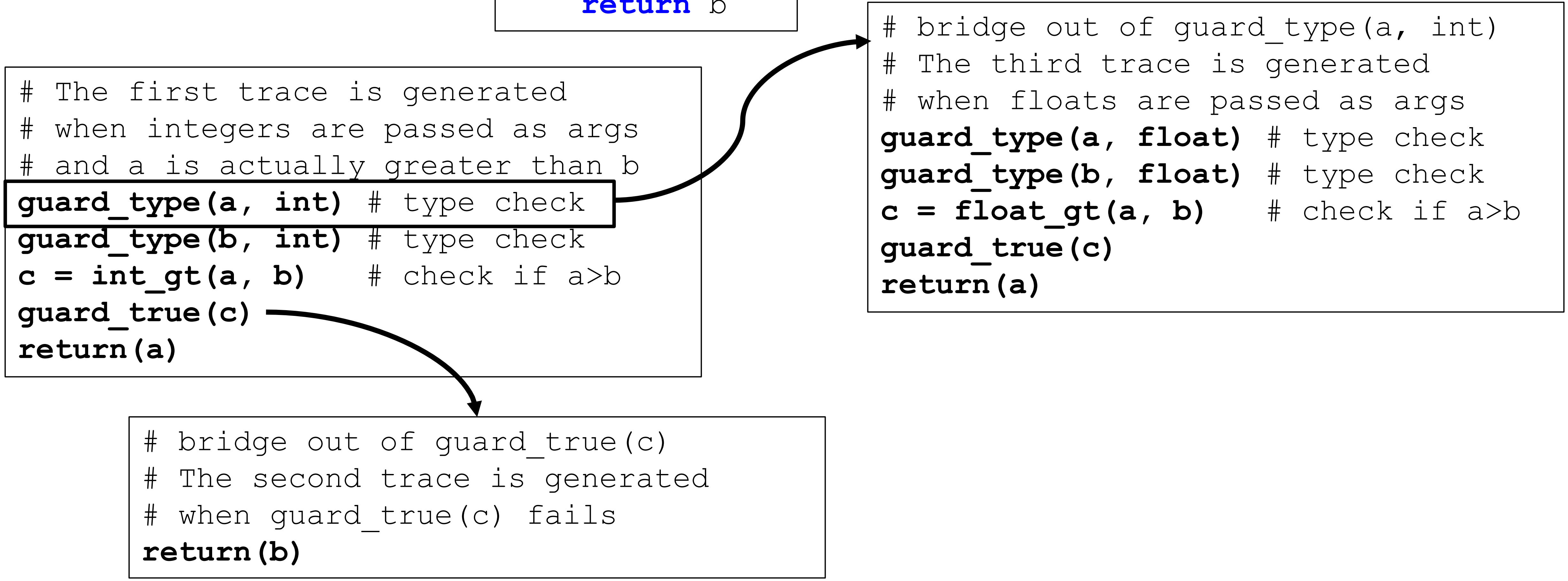
```
# The first trace is generated
# when integers are passed as args
# and a is actually greater than b
guard_type(a, int) # type check
guard_type(b, int) # type check
c = int_gt(a, b)   # check if a>b
guard_true(c)
return(a)
```

```
# bridge out of guard_true(c)
# The second trace is generated
# when guard_true(c) fails
return(b)
```

HOW TRACING JIT WORKS

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

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

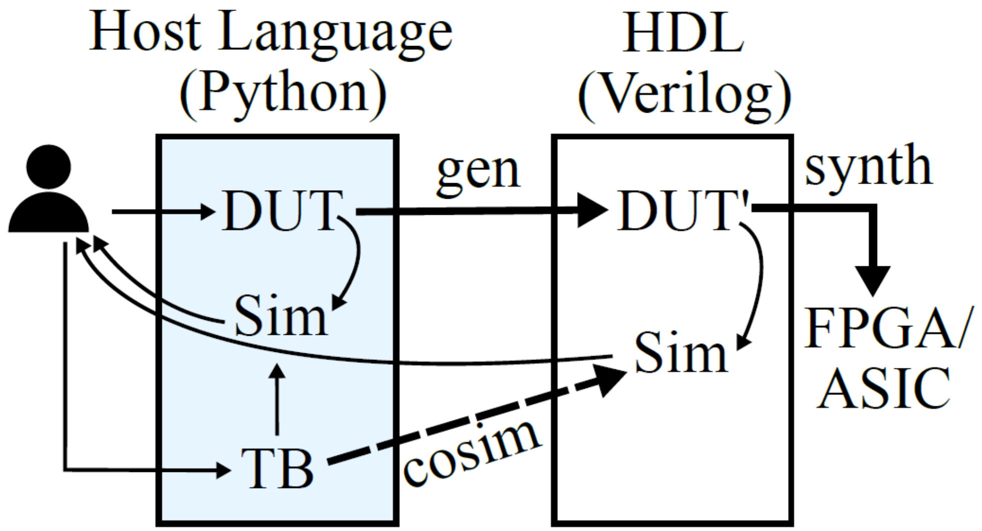


```
# The first trace is generated
# when integers are passed as args
# and a is actually greater than b
guard_type(a, int) # type check
guard_type(b, int) # type check
c = int_gt(a, b)   # check if a>b
guard_true(c)
return(a)
```

```
# bridge out of guard_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



Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

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

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**

Challenges: Event-Driven Simulation

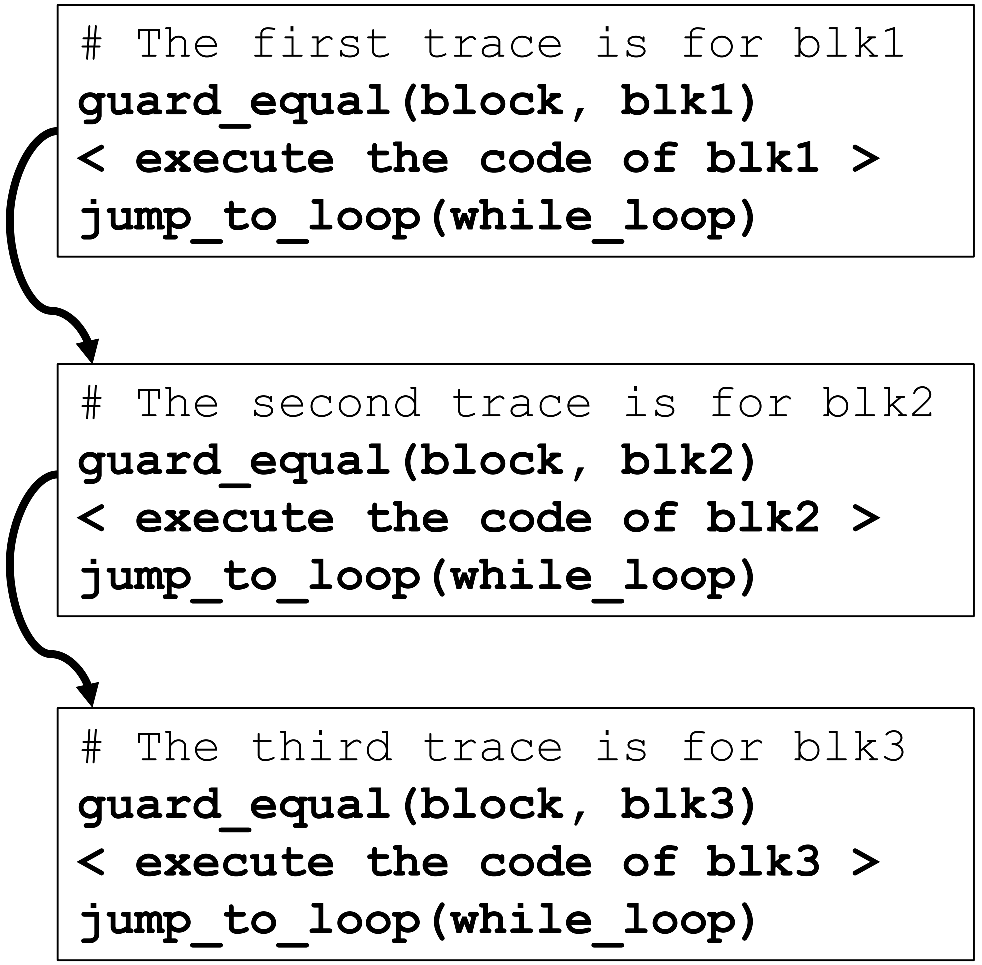
- **Event-driven simulation's inner loop is a bad pattern for tracing JIT**

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

CHALLENGES: EVENT-DRIVEN SIMULATION

- **Event-driven simulation's inner loop is a bad pattern for tracing JIT**

```
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
guard_equal(block, blk1)
< execute the code of blk1 >
jump_to_loop(while_loop)
```

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

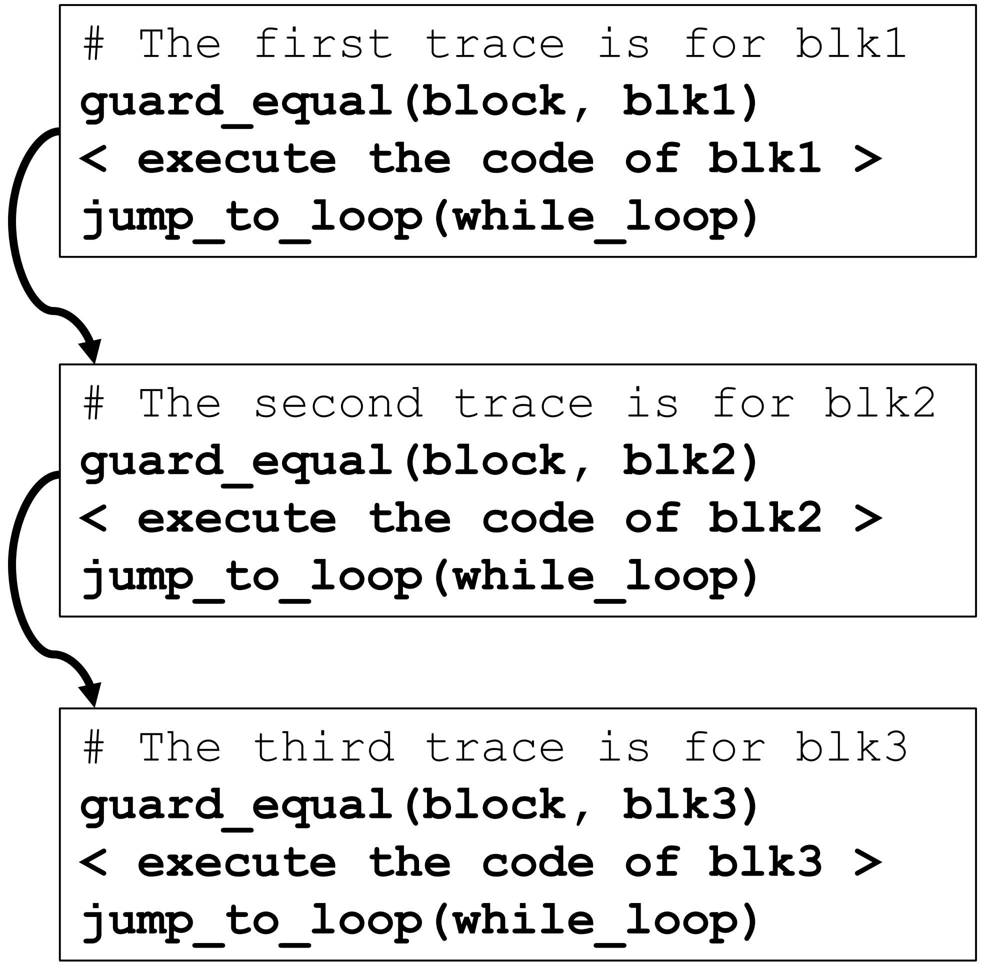
```
# The third trace is for blk3
guard_equal(block, blk3)
< execute the code of blk3 >
jump_to_loop(while_loop)
```

CHALLENGES: EVENT-DRIVEN SIMULATION

- **Event-driven simulation's inner loop is a bad pattern for tracing JIT**

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

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.



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

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

```
# The third trace is for blk3
guard_equal(block, blk3)
< execute the code of blk3 >
jump_to_loop(while_loop)
```

CHALLENGES: EMULATING FIX-WIDTH DATA TYPES

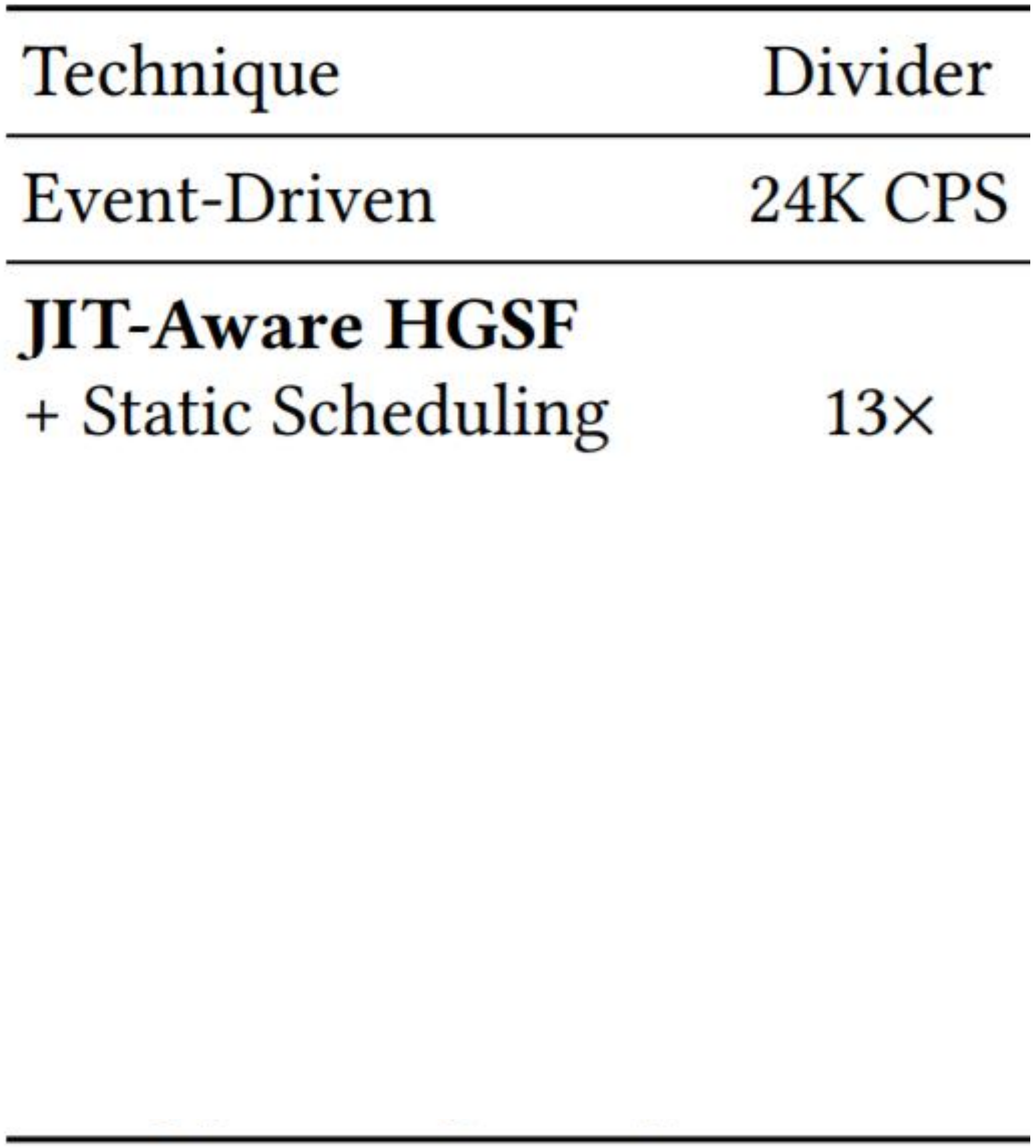
- **Emulating fix-width data types using Python integer is not the most efficient**
  - Python seamlessly promote integer to BigInt when overflowing 63-bit
  - However, each overflow is a guard failure
  - A 100-bit signal can either be BigInt or integer

  - 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

Mamba Techniques/Performance (All with PyPy)



Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF + Static Scheduling	13×

```
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)

Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF	
+ Static Scheduling	13×
+ Schedule Unrolling	16×

```
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();
```

Mamba Techniques/Performance (All with PyPy)

Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF	
+ Static Scheduling	13×
+ Schedule Unrolling	16×
+ Heuristic Toposort	18×

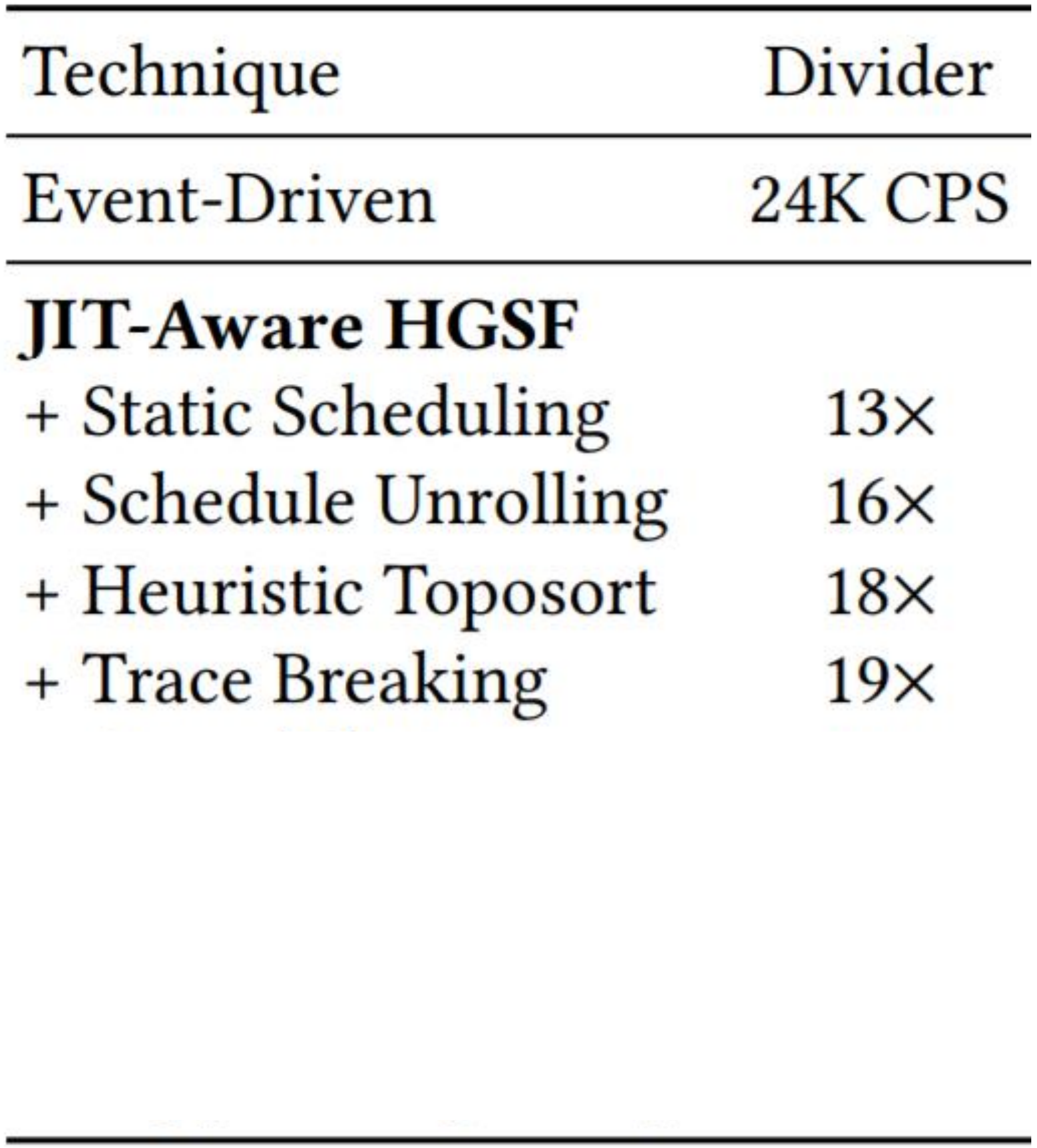
```
num_cycles = 1000000
for i in xrange(num_cycles):
  while not event_queue.empty():
    block = event_queue.pop()
    block()
```

```
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(); ...
```

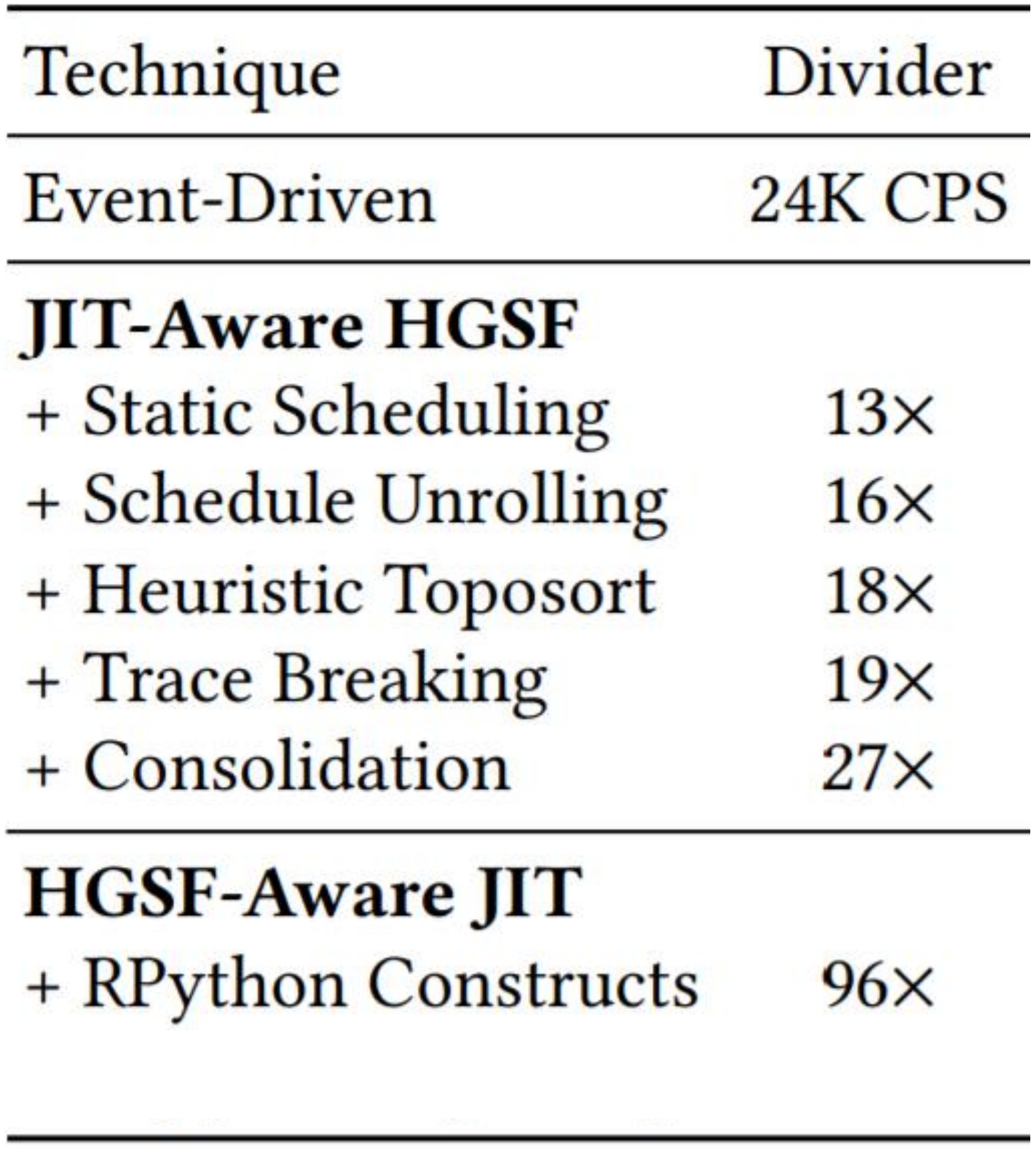
MAMBA TECHNIQUES/PERFORMANCE (ALL WITH PYPY)



Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF	
+ Static Scheduling	13×
+ Schedule Unrolling	16×
+ Heuristic Toposort	18×
+ Trace Breaking	19×

```
for i in xrange(num_cycles):
  block3();
  block1();
  jit_break_trace()
  block4();
  block2(); ...
```

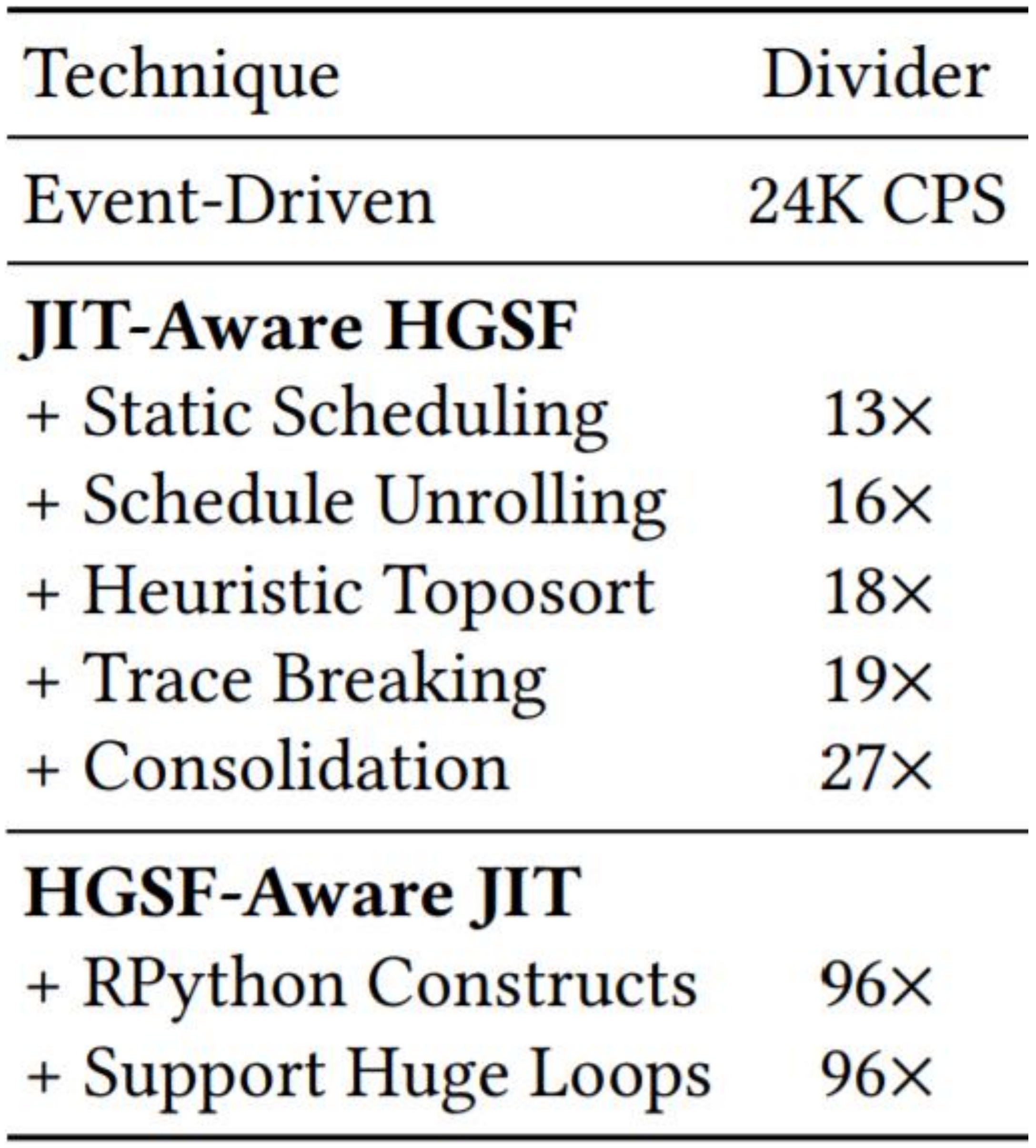
MAMBA TECHNIQUES/PERFORMANCE (ALL WITH PYPY)



Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF	
+ Static Scheduling	13×
+ Schedule Unrolling	16×
+ Heuristic Toposort	18×
+ Trace Breaking	19×
+ Consolidation	27×
HGSF-Aware JIT	
+ RPython Constructs	96×

"Letting the generate-purpose JIT recognize RTL simulation constructs" – As a proof of concept, we implement fix-bitwidth data types in RPython framework.

MAMBA TECHNIQUES/PERFORMANCE (ALL WITH PYPY)

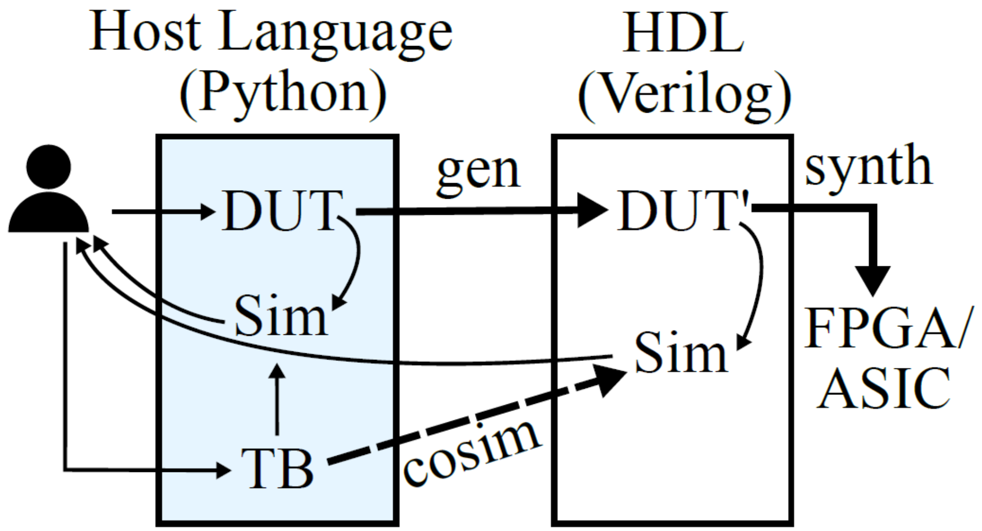


Technique	Divider
Event-Driven	24K CPS
JIT-Aware HGSF	
+ Static Scheduling	13×
+ Schedule Unrolling	16×
+ Heuristic Toposort	18×
+ Trace Breaking	19×
+ Consolidation	27×
HGSF-Aware JIT	
+ RPython Constructs	96×
+ Support Huge Loops	96×

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



Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

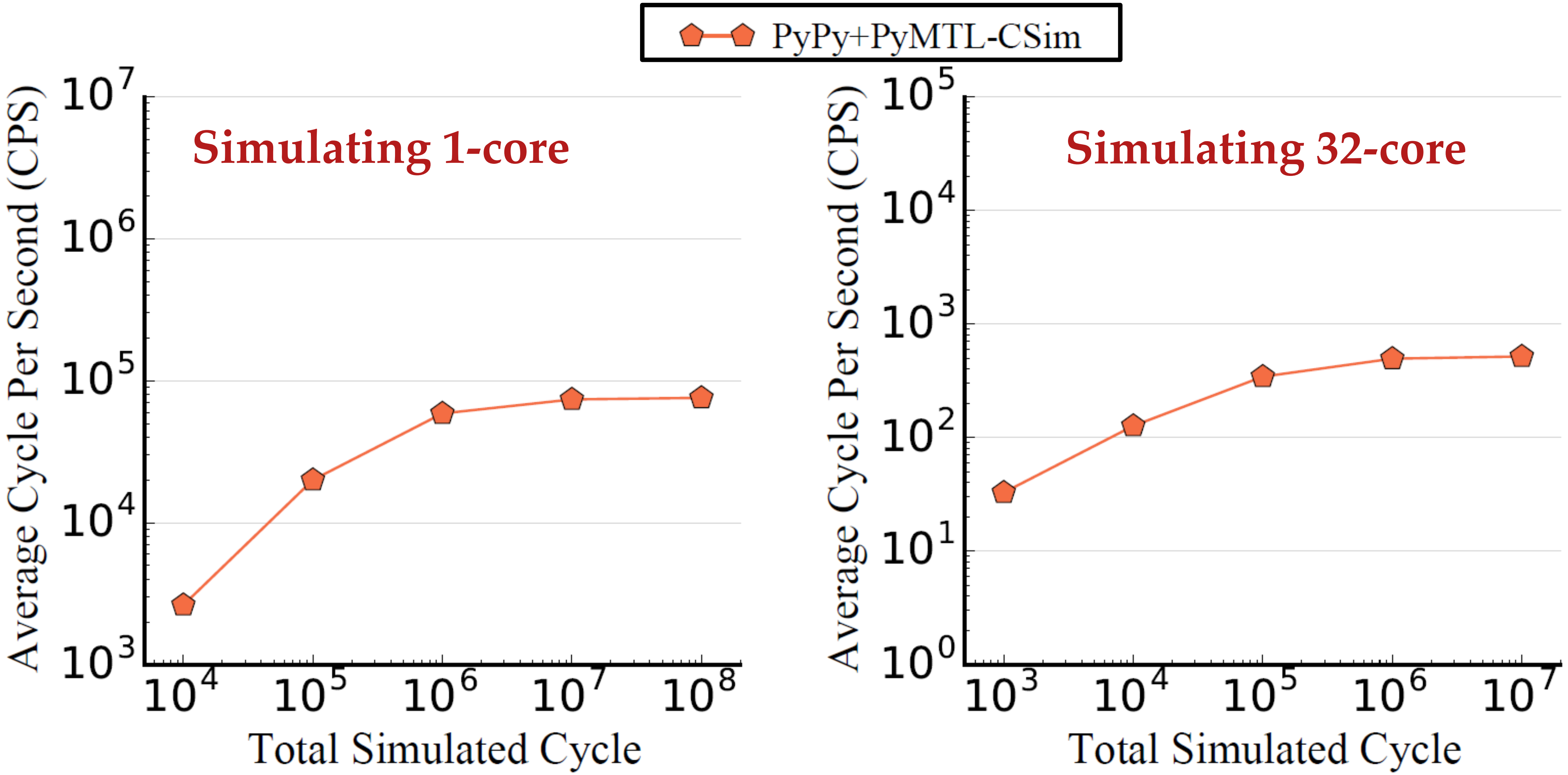
- Understanding the performance gap
- Background on tracing JIT compiler
- Co-optimizing the JIT and the HGSF
- **Mamba performance**

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

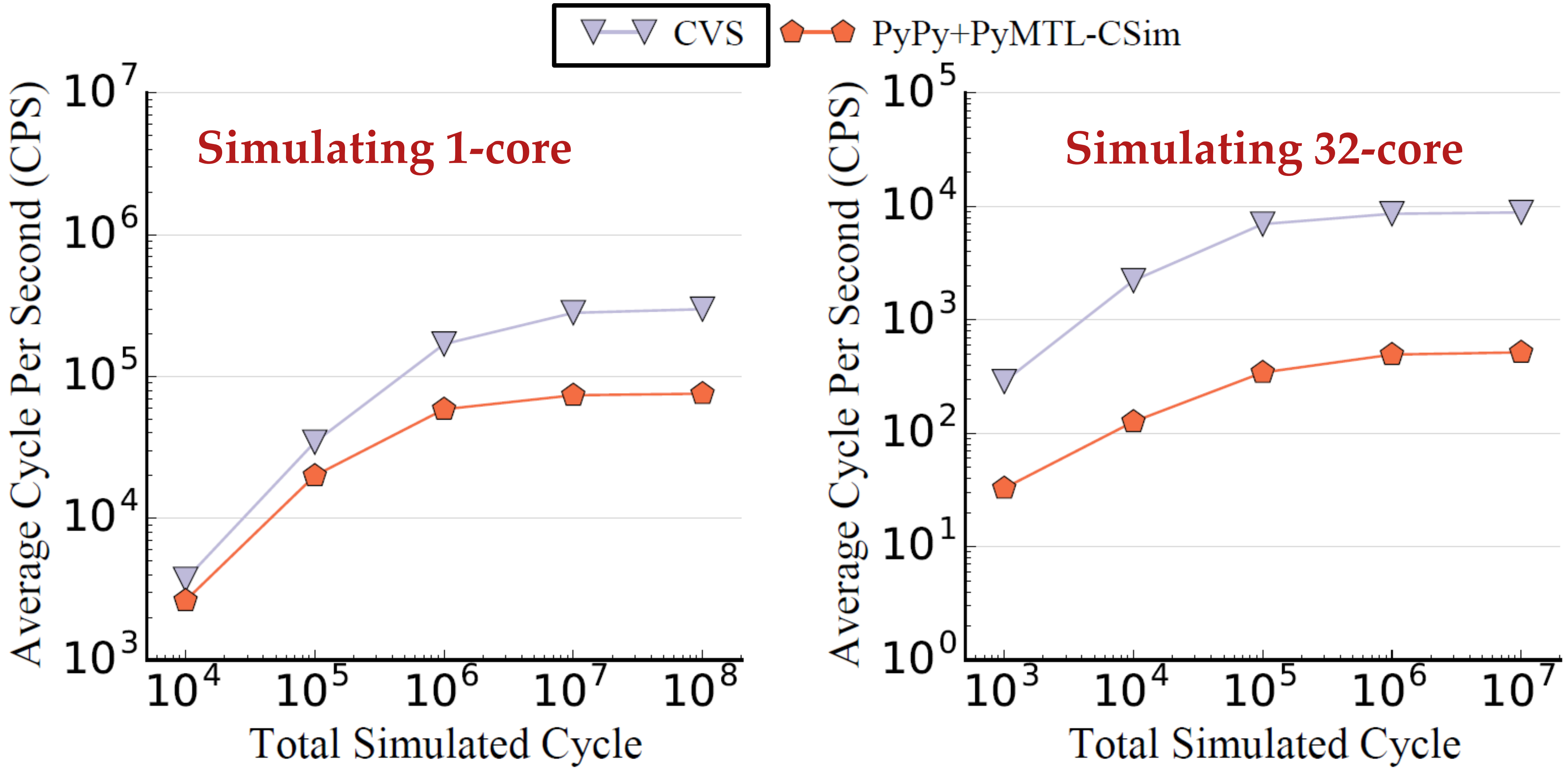
- **Competitors:**
  - Mamba
  - Verilator, Icarus Verilog, CVS
  - PyMTL, PyMTL-CSim

PERFORMANCE (W/ COMPILATION AND STARTUP OVERHEADS)



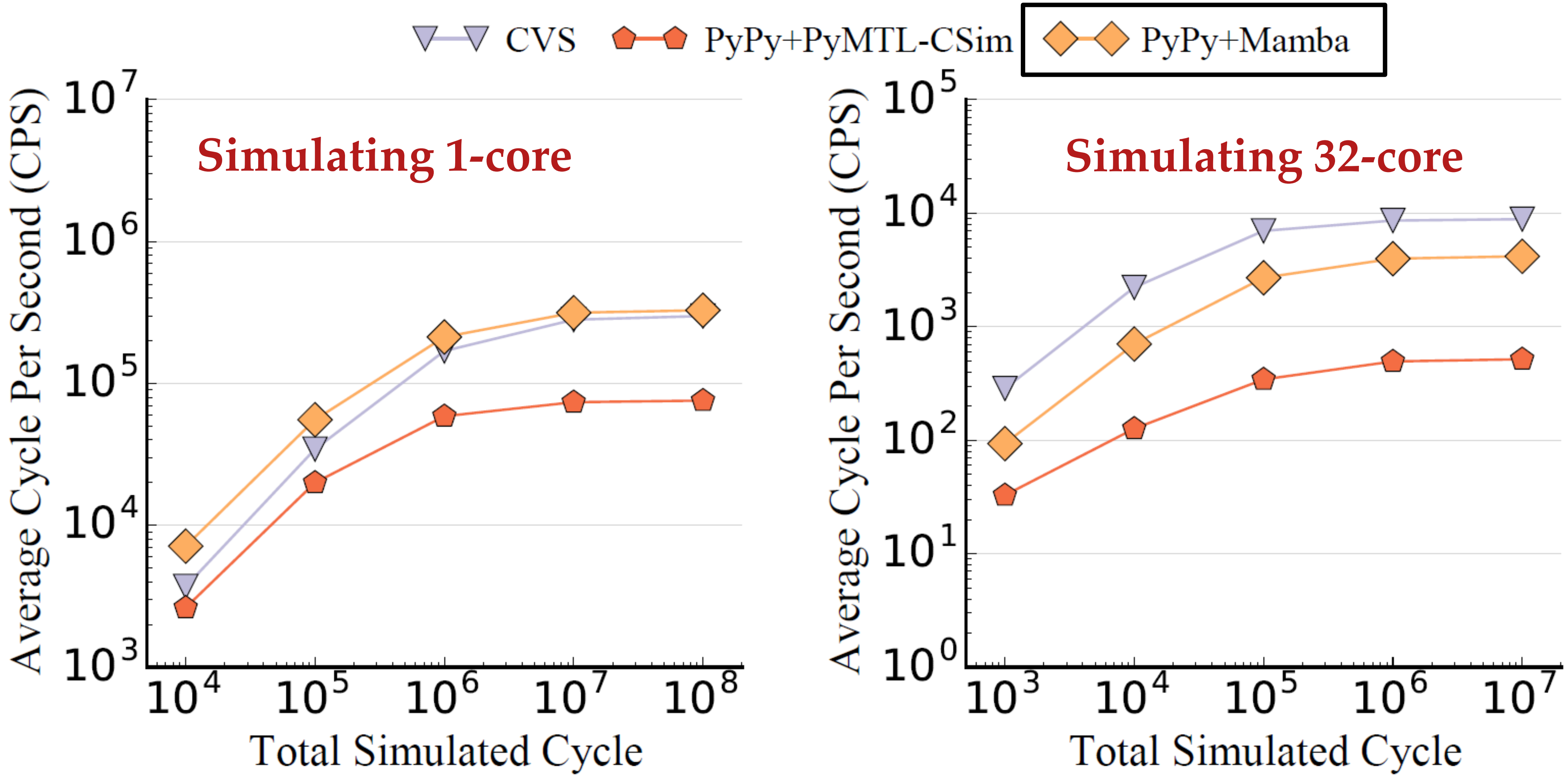
$$\text{Average Cycle Per Second} = \frac{\text{Simulated cycle}}{\textbf{Compilation time} + \textbf{Startup Overhead} + \text{Simulation time}}$$

Performance (w/ compilation and startup overheads)



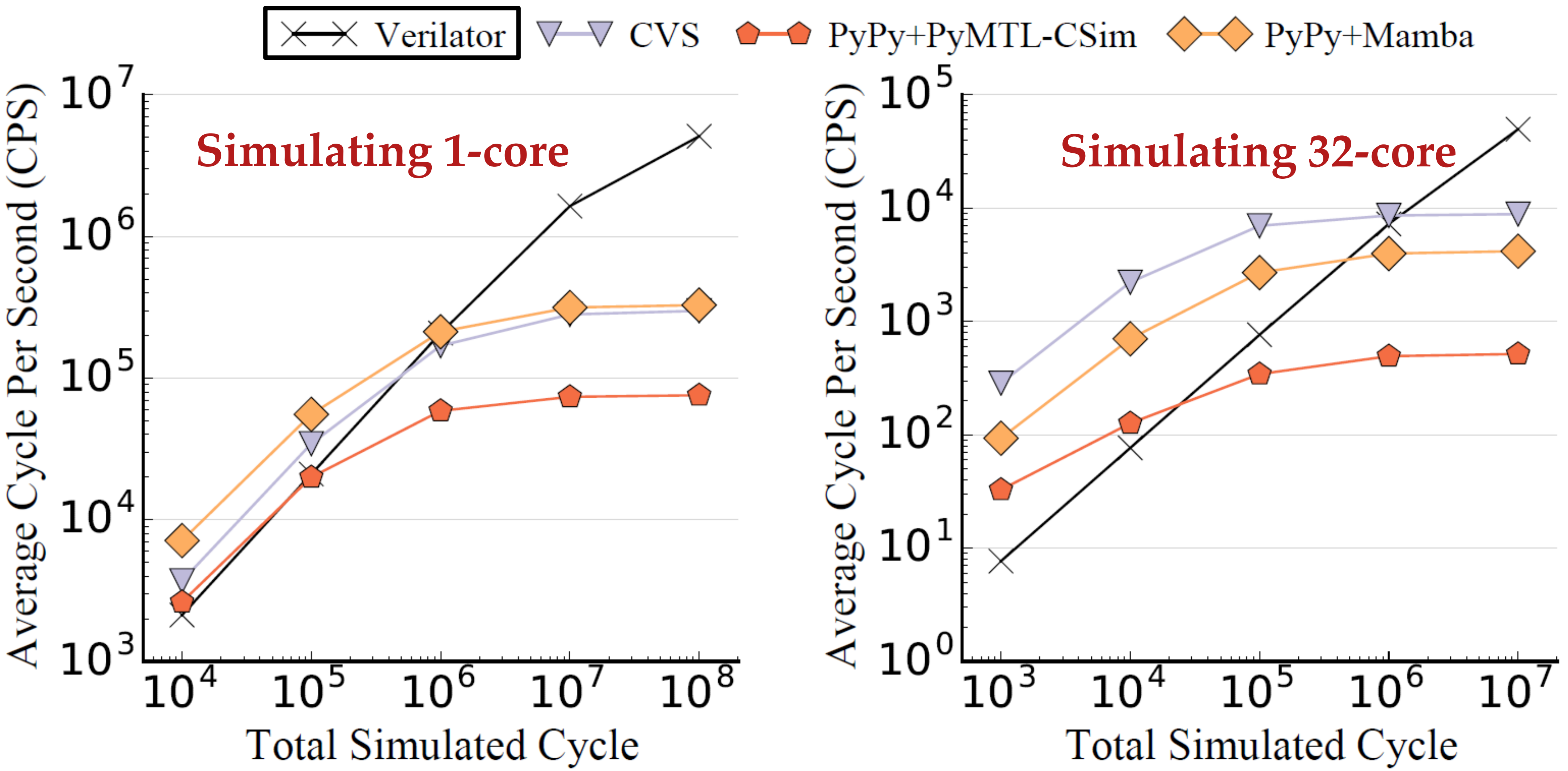
$$\text{Average Cycle Per Second} = \frac{\text{Simulated cycle}}{\textbf{Compilation time} + \textbf{Startup Overhead} + \text{Simulation time}}$$

PERFORMANCE (W/ COMPILATION AND STARTUP OVERHEADS)



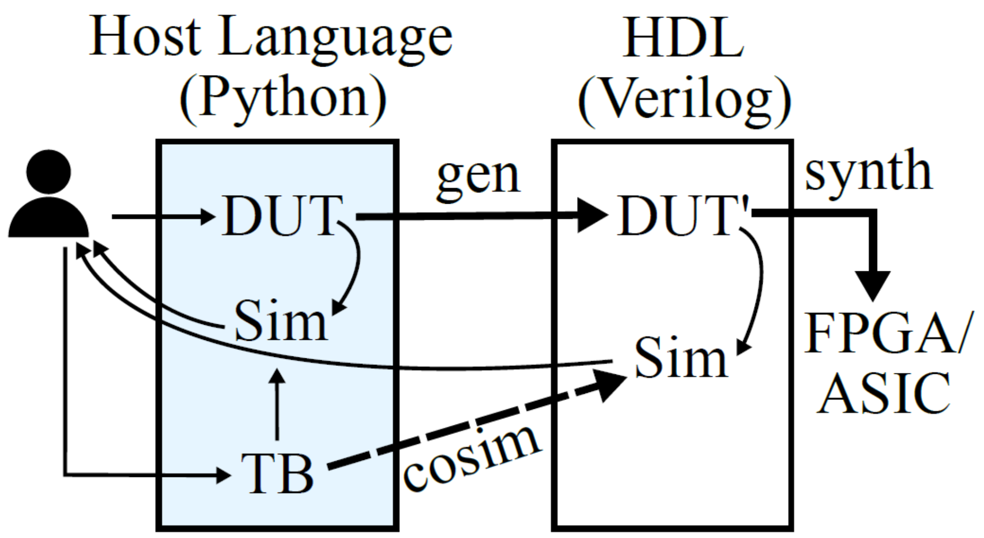
$$\text{Average Cycle Per Second} = \frac{\text{Simulated cycle}}{\textbf{Compilation time} + \textbf{Startup Overhead} + \text{Simulation time}}$$

Performance (w/ compilation and startup overheads)



$$\text{Average Cycle Per Second} = \frac{\text{Simulated cycle}}{\textbf{Compilation time} + \textbf{Startup Overhead} + \text{Simulation time}}$$

CONCLUSION



Hardware generation and *simulation* framework (HGSF)
- Example: PyMTL

- **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/mamba-dac2018
- 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