
Derek Lockhart, Gary Zibrat, and Christopher Batten
Outline

- The Computer Architecture Research Methodology Gap
  - PyMTL

- The Performance-Productivity Gap
  - SimJIT
Trends in Computing Systems

Energy & Power
Constrained

Credible
Energy and Power
Analysis
Trends in Computing Systems

Energy & Power Constrained

Extensive Specialization

Credible Energy and Power Analysis

Productive Design Space Exploration of Specialized Units
Trends in Computing Systems

Energy & Power Constrained

Extensive Specialization

Cross-Layer Optimization

Credible Energy and Power Analysis

Productive Design Space Exploration of Specialized Units

Effective Strategies for Vertically Integrated Design
Managing Increasing Design Complexity

- Abstractions
- Methodologies
- Patterns, Languages, Tools

Computer Architecture Research Abstractions

- Abstractions
- Methodologies
- Patterns, Languages, Tools
Computer Architecture Research Abstractions

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Sea of Transistors

Computer Architecture Research Abstractions

Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

VLSI

Sea of Transistors

Industry Development

Hundreds of Engineers

Academic Research

A Few Researchers

Computer Architecture Research Abstractions

Industry Development
- Hundreds of Engineers

Academic Research
- A Few Researchers

Sea of Transistors

Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI

Computer Architecture Research Methodologies

- Abstractions
- Methodologies
- Patterns, Languages, Tools
Computer Architecture Research Methodologies

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Sea of Transistors

Cycle Level
- Behavior
- Timing
Computer Architecture Research Methodologies

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Sea of Transistors

Cycle Level
- Behavior
- Timing

Computer Architecture Research Methodologies

- Applications
  - Algorithms
  - Compilers
  - Instruction Set Architecture
  - Microarchitecture
  - VLSI
  - Sea of Transistors

Functional Level
- Behavior

Cycle Level
- Behavior
- Timing
Computer Architecture Research Methodologies

- Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI
- Sea of Transistors

Functional Level
- Behavior

Cycle Level
- Behavior
- Timing

Register Transfer Level
- Behavior
- Timing
- Physical Resources
Computer Architecture Research Methodologies

Functional Level
- Behavior

Cycle Level
- Behavior
- Timing

Register Transfer Level
- Behavior
- Timing
- Physical Resources

Applications
- Algorithms
- Compilers
- Instruction Set Architecture
- Microarchitecture
- VLSI

Sea of Transistors

Computer Architecture Research Methodologies

Modeling Towards Layout

- Functional Level
  - Behavior

- Cycle Level
  - Behavior
  - Timing

- Register Transfer Level
  - Behavior
  - Timing
  - Physical Resources
Computer Architecture Research Methodologies

Computer Architecture Research Frameworks

- Abstractions
- Methodologies
- Patterns, Languages, Tools
Computer Architecture Research Frameworks

MATLAB/Python Algorithm or C++ Instruction Set Simulator

C++ Computer Architecture Simulation Framework (Object-Oriented)

Verilog or VHDL Design with EDA Toolflow (Concurrent-Structural)

Functional Level

Algorithm and ISA Development

Cycle Level

Design Space Exploration

Register Transfer Level

Area/Energy/Timing Validation and Prototype Development
Computer Architecture Research Frameworks

The Computer Architecture Research Methodology Gap

Different languages, patterns, and tools!

- Functional Level
- Algorithm and ISA Development
- Cycle Level
- Design Space Exploration
- Register Transfer Level
- Area/Energy/Timing Validation and Prototype Development

Great Ideas From Prior Work

- **Concurrent-Structural Modeling**
  (Liberty, Cascade, SystemC)
Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)
  - Consistent interfaces across abstractions

- **Unified Modeling Languages** (SystemC)
  - Unified design environment for FL, CL, RTL
Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)
  - Consistent interfaces across abstractions

- **Unified Modeling Languages** (SystemC)
  - Unified design environment for FL, CL, RTL

- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL)
  - Productive RTL design space exploration
Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)
  - Consistent interfaces across abstractions
- **Unified Modeling Languages** (SystemC)
  - Unified design environment for FL, CL, RTL
- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL)
  - Productive RTL design space exploration
- **HDL-Integrated Simulation Frameworks** (Cascade)
  - Productive RTL validation and cosimulation
### Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)
  - Consistent interfaces across abstractions

- **Unified Modeling Languages** (SystemC)
  - Unified design environment for FL, CL, RTL

- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL)
  - Productive RTL design space exploration

- **HDL-Integrated Simulation Frameworks** (Cascade)
  - Productive RTL validation and cosimulation

- **Latency-Insensitive Interfaces** (Liberty, BlueSpec)
  - Component and test bench reuse
Outline

The Computer Architecture Research Methodology Gap → PyMTL

The Performance-Productivity Gap → SimJIT
What is PyMTL?

• A Python DSEL for concurrent-structural hardware modeling
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
- A Python tool for simulating PyMTL FL, CL, and RTL models
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
- A Python tool for simulating PyMTL FL, CL, and RTL models
- A Python tool for translating PyMTL RTL models into Verilog
What is PyMTL?

• A Python DSEL for concurrent-structural hardware modeling
• A Python API for analyzing models described in the PyMTL DSEL
• A Python tool for simulating PyMTL FL, CL, and RTL models
• A Python tool for translating PyMTL RTL models into Verilog
• A Python testing framework for model validation
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
What Does PyMTL Enable?

• Incremental refinement from algorithm to accelerator implementation
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog models
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog models
  (see Srinath et. al. in MICRO-47, Session 6B!)

![Diagram showing relationships between gem5, PyMTL, C++ Model, PyMTL, and Verilog Model]
# The PyMTL Framework

<table>
<thead>
<tr>
<th>Specification</th>
<th>Tools</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The PyMTL Framework

**Specification**

- Model
- Config

**Tools**

- Elaborator

**Output**

- Model Instance
The PyMTL Framework

Specification

- Test & Sim Harness
- Model
- Config

Tools

- Elaborator
- Simulation Tool

Output

- Model Instance
- Traces & VCD

The PyMTL Framework

**Specification**
- Test & Sim Harness
- Model
- Config

**Tools**
- Elaborator
- Simulation Tool
- Translation Tool

**Output**
- Model Instance
- Traces & VCD
- Verilog
- EDA Toolflow

The PyMTL Framework

**Specification**
- Test & Sim Harness
- Model
- Config

**Tools**
- Simulation Tool
- Translation Tool
- User Tool

**Output**
- Traces & VCD
- Verilog
- User Tool Output

EDA Toolflow
The PyMTL Framework

Speciﬁcation

- Test & Sim Harness
- Model
- Config

Tools

- Simulation Tool
- Translation Tool
- User Tool

Output

- Traces & VCD
- Verilog
- User Tool Output

EDA Toolflow

- Visualization
- Static Analysis
- Dynamic Checking
- FPGA Simulation
- High Level Synthesis
The PyMTL DSEL

def sorter_network( input ):
    return sorted( input )

[ 3, 1, 2, 0 ] ----> \( f(x) \) ----> [ 0, 1, 2, 3 ]
def sorter_network( input ):  
    return sorted( input )

class SorterNetworkFL( Model )

[3, 1, 2, 0] ----> f(x) ----> [0, 1, 2, 3]
def sorter_network( input ):
    return sorted( input )

class SorterNetworkFL( Model )
    def __init__( s, nbits, nports ):
        type   = Bits( nbits )
        s.in_  = InPort[ nports ]( type )
        s.out  = OutPort[ nports ]( type )
def sorter_network( input ):
    return sorted( input )

class SorterNetworkFL( Model )
    def __init__( s, nbits, nports ):
        s.in_ = InPort[ nports ]( nbits )
        s.out = OutPort[ nports ]( nbits )
The PyMTL DSEL

def sorter_network( input ):
    return sorted( input )

class SorterNetworkFL( Model )
    def __init__( s, nbits, nports ):
        s.in_ = InPort [nports](nbits)
        s.out = OutPort[nports](nbits)

@s.tick_fl
    def logic():

[3, 1, 2, 0] ----> f(x) ----> [0, 1, 2, 3]
def sorter_network( input ):
    return sorted( input )

class SorterNetworkFL( Model )
    def __init__( s, nbits, nports ):
        s.in_ = InPort [nports](nbits)
        s.out = OutPort[nports](nbits)

@s.tick_fl
def logic():
    for i, v in enumerate( sorted( s.in_ ) ) :
        s.out[i].next = v

[ 3, 1, 2, 0 ] ----> f(x) ----> [ 0, 1, 2, 3 ]
def sorter_network( input ):  
    return sorted( input )

class SorterNetworkCL( Model )
    def __init__( s, nbits, nports, delay=3 ):
        s.in_ = InPort[ nports ]( nbits )
        s.out = OutPort[ nports ]( nbits )

@s.tick_cl
def logic():

[ 3, 1, 2, 0 ] ----> f(x) ----> [ 0, 1, 2, 3 ]
The PyMTL DSEL

```python
def sorter_network( input ): return sorted( input )

class SorterNetworkCL( Model ):
    def __init__( s, nbits, nports, delay=3 ):
        s.in_ = InPort[ nports ]( nbits )
        s.out = OutPort[ nports ]( nbits )
        s.pipe = Pipeline( delay )

    @s.tick_cl
    def logic():
        s.pipe.xtick()
        s.pipe.push( sorted( s.in_ ) )

        if s.pipe.ready():
            for i, v in enumerate( s.pipe.pop() ):
                s.out[i].next = v
```

$[3, 1, 2, 0] \rightarrow f(x) \rightarrow [0, 1, 2, 3]$
def sorter_network( input ):
    return sorted( input )

class SorterNetworkRTL( Model )
def __init__( s, nbits ):
    s.in_ = InPort [4](nbits)
    s.out = OutPort[4](nbits)

[3, 1, 2, 0] ----> f(x) ----> [0, 1, 2, 3]
def sorter_network( input ):  
    return sorted( input )

class SorterNetworkRTL( Model )
    def __init__( s, nbits ):
        s.in_ = InPort [4](nbits)
        s.out = OutPort[4](nbits)
        s.m = m = MinMaxRTL[5](nbits)

[3, 1, 2, 0] ----> f(x) ----> [0, 1, 2, 3]
The PyMTL DSEL

def sorter_network( input ):  
    return sorted( input )

class SorterNetworkRTL( Model )  
def __init__( s, nbits ):  
    s.in_ = InPort[4](nbits)  
    s.out = OutPort[4](nbits)  
    s.m = m = MinMaxRTL[5](nbits)  

    s.connect( s.in_[0], m[0].in_[0] )  
    s.connect( s.in_[1], m[0].in_[1] )  
    s.connect( s.in_[2], m[1].in_[0] )  
    s.connect( s.in_[3], m[2].in_[1] )  

    ...

[3, 1, 2, 0] \rightarrow f(x) \rightarrow [0, 1, 2, 3]
class MinMaxRTL( Model )
    def __init__( s, nbits ):
        s.in_ = InPort [2](nbits)
        s.out = OutPort[2](nbits)
    @s.combinational
    def logic():
        swap = s.in_[0] > s.in_[1]
        s.out[0].value = s.in[1] if swap else s.in[0]
        s.out[1].value = s.in[0] if swap else s.in[1]
class MinMaxRTL( Model )
    def __init__( s, nbits ):
        s.in_ = InPort [2](nbits)
        s.out = OutPort[2](nbits)
    @s.combinational
    def logic():
        swap = s.in_[0] > s.in_[1]
        s.out[0].value = s.in[1] if swap else s.in[0]
        s.out[1].value = s.in[0] if swap else s.in[1]

class RegRTL( Model )
    def __init__( s, nbits ):
        s.in_ = InPort (nbits)
        s.out = OutPort(nbits)
    @s.tick_rtl
    def logic():
        s.out.next = s.in_
The PyMTL DSEL

Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.
The PyMTL DSEL

Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.

Productivity helpers:

- MemoryProxies
- QueueAdapters
- PortBundles
- BitStructs
- TestSource
- TestSink
The PyMTL DSEL

Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.

Productivity helpers:

- MemoryProxies
- QueueAdapters
- PortBundles
- BitStructs
- TestSource
- TestSink

See the paper for more examples!
Why Python?

Benefits:

• Modern language features enable rapid prototyping (dynamic-typing, reflection, metaprogramming)
• Lightweight, pseudocode-like syntax
• Built-in support for integrating C/C++ code
• Large, active developer and support community
Why Python?

Benefits:

• Modern language features enable rapid prototyping (dynamic-typing, reflection, metaprogramming)
• Lightweight, pseudocode-like syntax
• Built-in support for integrating C/C++ code
• Large, active developer and support community

Drawbacks:

• Performance
Outline

The Computer Architecture Research Methodology Gap

PyMTL

The Performance-Productivity Gap

SimJIT
Performance-Productivity Gap

Experiment:

• Simple 8x8 Mesh Network Model

• Cycle-Precise CL Model:
  • PyMTL Model Simulated with the CPython Interpreter
  • Hand-Written C++ Model and Simulator
Performance-Productivity Gap

Experiment:

• Simple 8x8 Mesh Network Model

• Cycle-Precise CL Model:
  • PyMTL Model Simulated with the CPython Interpreter
  • Hand-Written C++ Model and Simulator

• Bit-Accurate RTL Model:
  • PyMTL Model Simulated with CPython Interpreter
  • Hand-Written Verilog RTL Simulated with Verilator
Performance-Productivity Gap

CL Network

RTL Network

CPython
Performance-Productivity Gap

**CL Network**
- Simulated Cycles: 1K, 10K, 100K, 1M, 10M
- Y-axis: 1x, 5x, 10x, 20x, 50x, 100x, 200x, 500x, 1000x, 300x
- Languages: CPython, C++

**RTL Network**
- Simulated Cycles: 1K, 10K, 100K, 1M, 10M
- Y-axis: 1x, 5x, 10x, 20x, 50x, 100x, 200x, 500x, 1000x
- Languages: CPython, Verilator

Performance-Productivity Gap

Performance degradation due to Compilation

CL Network

RTL Network

Simulated Cycles

1K 10K 100K 1M 10M

Simulated Cycles

1K 10K 100K 1M

CPython

C++

Verilator

Performance-Productivity Gap

Short Simulations: Large-Compilation Overhead

CL Network

RTL Network

CPython

C++

Verilator

Long Simulations: Compilation Overhead Amortized

- **CL Network**
  - CPython
  - C++

- **RTL Network**
  - CPython
  - Verilog
Performance-Productivity Gap

CL Network

RTL Network

CPython

C++

Verilator

Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**
Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**

- **Python-Wrapped C/C++ Libraries**
  
  *NumPy, CVXOPT, NLPy, pythonOCC, GEM5*

- **Numerical Just-In-Time Compilers**
  
  *Numba, Parakeet*

- **Just-In-Time Compiled Interpreters**
  
  *PyPy, Pyston*

- **Selective Embedded Just-In-Time Specialization**
  
  *SEJITS*
Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**

- **Python-Wrapped C/C++ Libraries**
  (NumPy, CVXOPT, NLPy, pythonOCC, GEM5)

- **Numerical Just-In-Time Compilers**
  (Numba, Parakeet)

- **Just-In-Time Compiled Interpreters**
  (PyPy, Pyston)

- **Selective Embedded Just-In-Time Specialization**
  (SEJITS)
Performance-Productivity Gap

CL Network

- 300x performance gap

RTL Network

- 1200x performance gap

Simulated Cycles

CPython

C++

Verilator
Performance-Productivity Gap

CL Network

RTL Network

Simulated Cycles

CPython

PyPy

C++

Simulated Cycles

CPython

PyPy

Verilator

30x

240x

Outline

The Computer Architecture Research Methodology Gap → PyMTL

The Performance-Productivity Gap → SimJIT
PyMTL SimJIT Architecture

SimJIT-RTL Tool

Verilog Source

Translation

PyMTL RTL Model Instance
PyMTL SimJIT Architecture

- Verilog Source
- Verilator
- RTL C++ Source
- Translation

PyMTL RTL Model Instance

SimJIT-RTL Tool
PyMTL SimJIT Architecture

PyMTL SimJIT Architecture

PyMTL SimJIT Architecture
PyMTL SimJIT Architecture

SimJIT-RTL Tool

- Verilog Source
  - Verilator
  - RTL C++ Source
  - Translation Cache
  - C Shared Library
  - Wrapper Gen
  - LLVM/GCC
  - C Interface Source
  - Translation
  - PyMTL RTL Model Instance
  - PyMTL CFFI Model Instance
PyMTL SimJIT Architecture

SimJIT-RTL Tool

Verilog Source

Verilator

RTL C++ Source

Translation Cache

C Shared Library

C Interface Source

LLVM/GCC

Wrapper Gen

PyMTL CFFI Model Instance

PyMTL RTL Model Instance

Fairly robust, ready for use in research!
PyMTL SimJIT Architecture

SimJIT-CL Tool

- CL C++ Source
- C Interface Source
- LLVM/GCC
- C Shared Library
- Wrapper Gen

PyMTL CL Model Instance

Just a prototype!
Performance-Productivity Gap

CL Network

RTL Network

Simulated Cycles

1K 10K 100K 1M

Simulated Cycles

1K 10K 100K 1M

1x 5x 10x 30x

60x

200x

1000x

CPython PyPy C++

Verilator

PyMTL SimJIT Performance

![Graph showing comparison between CPython, PyPy, C++, and Verilator in CL and RTL network simulations.](image)

PyMTL SimJIT Performance

**CL Network**

- CPython
- PyPy
- C++
- SimJIT-CL
- SimJIT-CL+PyPy

**RTL Network**

- CPython
- PyPy
- Verilator
- SimJIT-RTL
- SimJIT-RTL+PyPy

4.5x improvement

6x improvement

Simulated Cycles:

1K 10K 100K 1M 10M

CPython
PyPy
C++
SimJIT-CL
SimJIT-CL+PyPy

Verilator
SimJIT-RTL
SimJIT-RTL+PyPy

PyMTL SimJIT Performance

Opportunities to further reduce the performance gap:

- **Reduce overhead of Python-to-C++ interfaces**
- Optimized (non-Python) event queue
- Better code generation
- Better event queue scheduling
- Removal of unnecessary double-buffering
- Parallel simulation
Contributions

PyMTL is a productive Python framework for FL, CL, and RTL modeling, enabling:

- Vertically Integrated Computer Architecture Research
- Accelerator Design Space Exploration
- Construction of Flexible RTL Chip Generators

SimJIT considerably closes the performance-productivity gap between Python and C++ simulations.

- 72x Speedup over CPython for SimJIT-CL (within 4.5x of C++)
- 200x Speedup over CPython for SimJIT-RTL (within 6x of Verilator)
PyMTL is a productive, open-source Python framework for FL/CL/RTL modeling and hardware design.

https://github.com/cornell-brg/pymtl

Thank you to our sponsors for their support: NSF, DARPA, and donations from Intel Corporation and Synopsys, Inc.