PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

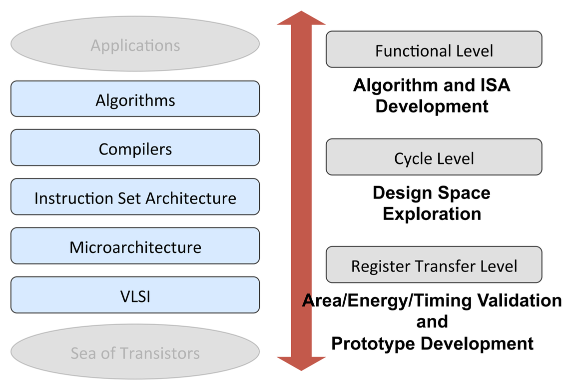
Derek Lockhart, Gary Zibrat, Christopher Batten
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

1 Abstract

Technology trends prompting architects to consider greater heterogeneity and hardware specialization have exposed an increasing need for vertically integrated research methodologies that can effectively assess performance, area, and energy metrics of future architectures. However, constructing such a methodology with existing tools is a significant challenge due to the unique languages, design patterns, and tools used in functional-level (FL), cycle-level (CL), and register-transfer-level (RTL) modeling. **We introduce a new framework called PyMTL that aims to close this computer architecture research methodology gap by providing a unified design environment for FL, CL, and RTL modeling.** PyMTL leverages the Python programming language to create a highly productive domain-specific embedded language for concurrent-structural modeling and hardware design. While the use of Python as a modeling and framework implementation language provides considerable benefits in terms of productivity, it comes at the cost of significantly longer simulation times. We address this performance-productivity gap with a hybrid JIT compilation and JIT specialization approach. We introduce **SimJIT, a custom JIT specialization engine that automatically generates optimized C++ for CL and RTL models.** To reduce the performance impact of the remaining unspecialized code, we combine SimJIT with an off-the-shelf Python interpreter with a meta-tracing JIT compiler (PyPy). SimJIT+PyPy provides **speedups of up to 72× for CL models and 200× for RTL models, bringing us within 4–6× of optimized C++ code** while providing significant benefits in terms of productivity and usability.

2 Motivation

Energy and power constraints in modern computing systems have driven architects to consider optimizations which reach across the entire computing stack. This has prompted a need for vertically integrated research approaches that use multiple modeling methodologies to effectively explore novel architectures at various levels of abstraction. For example, the incremental design of a specialized accelerator from algorithm to implementation may leverage the following modeling methodologies:



- **Functional-level** (FL) modeling to perform algorithmic exploration.
- **Cycle-level** (CL) modeling for rapid architectural design space exploration.
- **Register-transfer-level** (RTL) modeling for extraction of credible area, energy, and timing estimation.

We call such a vertically integrated approach to design space exploration a **modeling towards layout** methodology. Unfortunately, current tools for FL, CL, and RTL modeling typically use different programming languages, design patterns, and software tools that make it difficult for designers to quickly transition between abstraction levels. This **computer architecture research methodology gap** makes it a challenge for architecture researchers to rapidly iterate across the stack and create a productive, vertically integrated design flow.

	FL	CL	RTL
Modeling Languages	Productivity Level (PLL)	Efficiency Level (ELL)	Hardware Description (HDL)
	MATLAB/Python	*C/C++*	*SystemVerilog/VHDL*
Modeling Patterns	Functional: Data Structures, Algorithms	Object Oriented: Classes, Methods, Ticks and/or Events	Concurrent-Structural: Combinational Logic, Clocked Logic, Port Interfaces
Modeling Tools	3rd-party Algorithm Packages and Toolboxes	Computer Architecture Simulation Frameworks	Simulator Generators, Synthesis Tools, Verification Tools

Inspired by insights from prior work, we set out to create a new framework incorporating several key design features to improve the productivity of vertically integrated design space exploration. These features include:

- **Concurrent-structural** programming constructs for hardware-centric modeling.
- A **unified modeling language** for FL, CL, and RTL model descriptions.
- **Hardware generation language** capabilities to improve RTL design productivity.
- **HDL integration** to enable co-simulation with Verilog IP.
- **Latency-insensitive design** to promote component and testbench reuse.

3 PyMTL

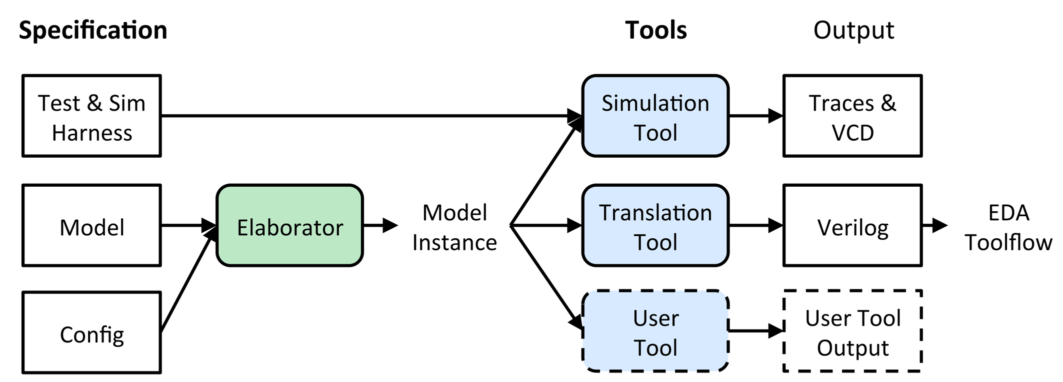
PyMTL is a Python-based proof-of-concept framework designed to provide a unified environment for constructing FL, CL, and RTL models, enabling productive vertically integrated computer architecture research. PyMTL has been released as **open-source software available via GitHub.**

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

The PyMTL framework consists of the following core components:

- 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 verification

PyMTL software architecture is designed with **model/tool split**. The modular nature of the model/tool split encourages extensibility and provides a simple path for users to write their own custom tools such as linters, translators, and visualization tools. More importantly, it provides a clean boundary between hardware modeling logic and simulator implementation logic letting users focus on hardware design rather than simulator software engineering.



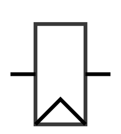
4 The PyMTL DSEL

The PyMTL domain-specific embedded language (DSEL) provides several constructs to enable concurrent-structural modeling within Python, including:

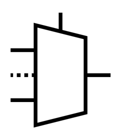
- `InPorts`, `OutPorts`, and `PortBundles` for specifying parameterizable interfaces
- `Wires` and `s.connect` for programmatic structural composition of models
- Several decorators for specifying concurrent block execution semantics (`s.tick_fl`, `s.tick_cl`, `s.tick_rtl`, `s.combinational`)
- `Bits` and `BitStruct` fixed-bitwidth message types
- The `Model` base class which provides helpers for inspecting elaborated designs

Below we show three very basic examples of sequential (`Register`), combinational (`Mux`), and structural (`MuxReg`) models described using the PyMTL DSEL.

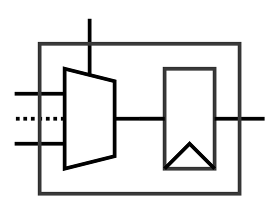
```
class Register( Model ):
  def __init__(s, nbits):
    type  = Bits( nbits )
    s.in_ = InPort ( type )
    s.out = OutPort( type )

    @s.tick_rtl
    def seq_logic():
      s.out.next = s.in_
```



```
class Mux( Model ):
  def __init__(s, nbits, nports):
    s.in_ = InPort[nports](nbits)
    s.sel = InPort (bw(nports))
    s.out = OutPort(nbits)

    @s.combinational
    def comb_logic():
      s.out.value = s.in_[s.sel]
```



```
class MuxReg( Model ):
  def __init__( s, nbits=8, nports=4 ):
    s.in_ = [ InPort( nbits ) for x in range( nports ) ]
    s.sel = InPort ( bw( nports ) )
    s.out = OutPort( nbits )

    s.reg_ = Register( nbits )
    s.mux  = Mux     ( nbits, nports )

    s.connect( s.sel, s.mux.sel )
    for i in range( nports ):
      s.connect( s.in_[i], s.mux.in_[i] )
    s.connect( s.mux.out,  s.reg_.in_ )
    s.connect( s.reg_.out, s.out      )
```



5 Testing PyMTL Models

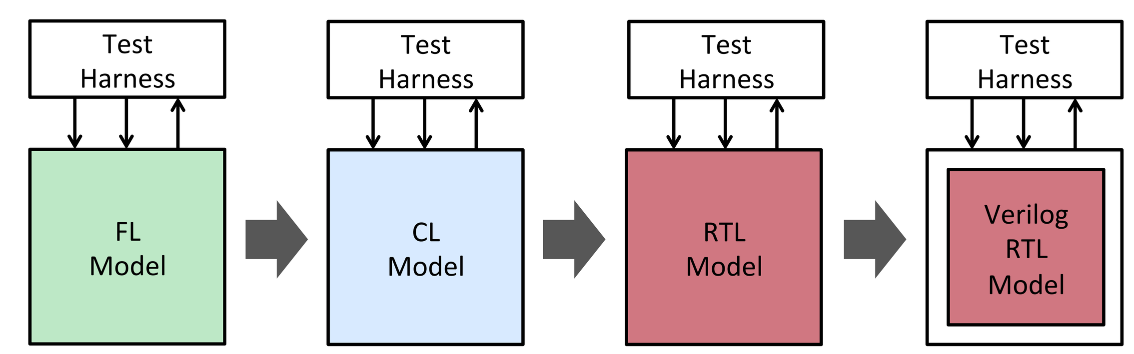
Testing of PyMTL tools and libraries is performed in Python using the open-source **py.test** library. Below is an example test which is parameterized to verify behavior for a variety of bitwidths and port numbers.

```
@pytest.mark.parametrize(
  'nbits,nports', [( 8, 2), ( 8, 3), ( 8, 4), ( 8, 8)
                   (32, 2), (32, 3), (32, 4), (32, 8)]
)
def test_muxreg( nbits, nports, test_verilog ):

  # instantiate the MuxReg model with th provided parameters
  model = MuxReg( nbits, nports )
  model.elaborate()

  # create a Python-wrapped Verilog translation of MuxReg model
  # if --test-verilog is passsed at the commandline
  if test_verilog:
    model = TranslationTool( model )

  # construct a simulator for the MuxReg (or Verilog MuxReg) model
  # set input vectors and verify output vectors
  sim = SimulationTool( model )
  for inputs, sel, output in gen_vectors(nbits,nports):
    for i, val in enumerate( inputs ):
      model.in_[i].value = val
    model.sel.value = sel
    sim.cycle()
    assert model.out == output
```

The `SimulationTool` is used to construct a Python simulator for the `MuxReg` model and verify its behavior. When combined with the `TranslationTool`, this same test can be used to validate the PyMTL-generated Verilog translation of the model.

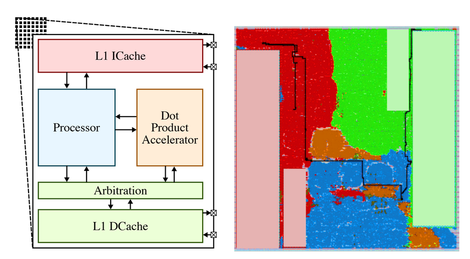
6 Modeling Towards Layout in PyMTL

PyMTL was designed to enable the incremental refinement of a component from high-level model to bit-precise RTL implementation. The use of **port-based** and **latency-insensitive** interfaces enables the designer to construct a test harness once and reuse it across abstraction levels to verify FL, CL, and RTL PyMTL models, as well as to validate Verilog models generated using the `TranslationTool`.



The FL model to the right shows some of the helpers PyMTL provides to facilitate creating port-based interfaces for FL and CL models. `PortBundles` are used to concisely describe interfaces with data, valid, and ready ports. `QueueAdapters` provide a simple, queue-like abstraction to these ports. This approach allows the user to use a traditional software implementation of CRC32 to perform the computation.

```
class CRC32FL( Model ):
  def __init__( s ):
    s.in_ = InValRdyBundle ( CRCMsgType() )
    s.out = OutValRdyBundle( 32           )

    s.q0 = InQueueAdapter ( s.in_ )
    s.q1 = OutQueueAdapter( s.out )

    @s.tick_fl
    def seq_logic():
      s.q0.xtick()
      s.q1.xtick()
      if not s.q0.empty() and not s.q1.full():
        t = s.q0.deq()
        s.q1.enq( crc32(t.data, t.start) )
```

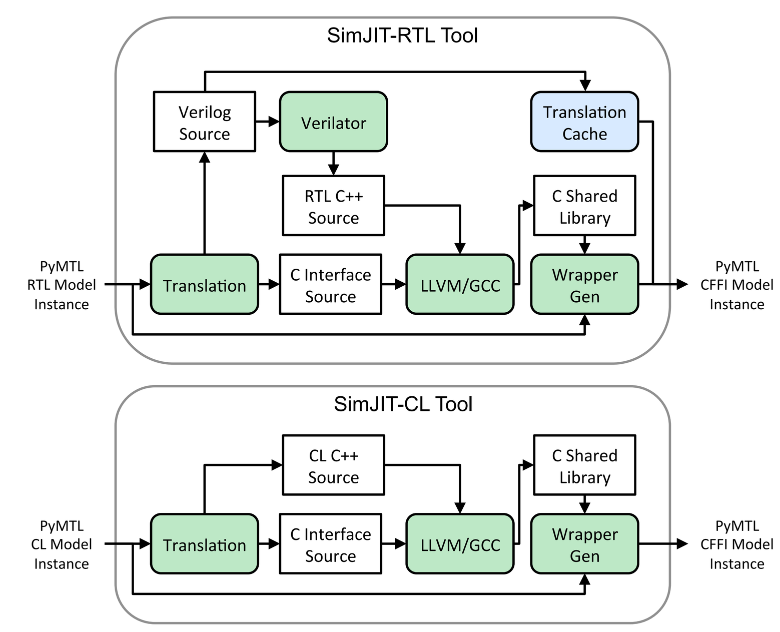


The above FL code can be refined into a cycle-approximate CL model by using timing information to delay the result, and then further refined to an RTL implementation using the low-level constructs provided by the PyMTL DSEL. Each of these models can reuse the same test harness due to the use of latency-insensitive interfaces.

Please see the paper for an example of using PyMTL to iteratively refine a dot-product coprocessor design from FL model down to placed-and-routed layout (shown in diagram above).

7 SimJIT

Python greatly improves the expressiveness, productivity, and flexibility of model code, but demonstrates poor simulation performance when compared to a statically compiled language like C++. We address this performance-productivity gap by using a hybrid just-in-time optimization approach that includes SimJIT, a custom just-in- time specializer for converting PyMTL models into optimized C++ code. SimJIT consists of two distinct specializers for cycle-level (SimJIT-CL) and RTL models (SimJIT-RTL).



SimJIT-RTL translates RTL models into Verilog HDL, then uses Verilator to generate C++ simulator source from this HDL. Generated C++ is wrapped in Python to create a PyMTL compatible interface.

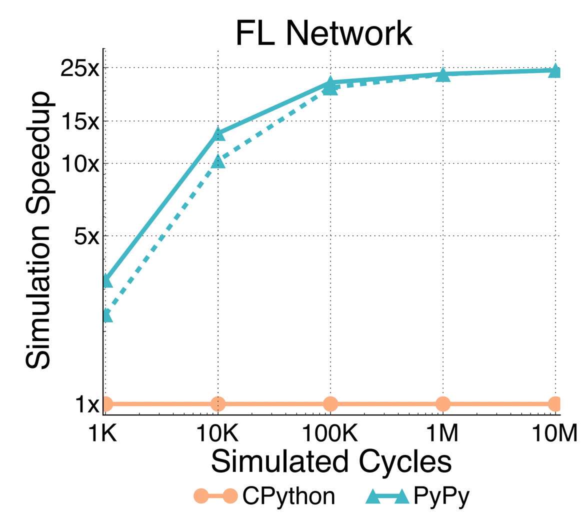
SimJIT-CL uses a custom code generator to convert CL models into Python wrapped C++ source. Caching is planned for future release.

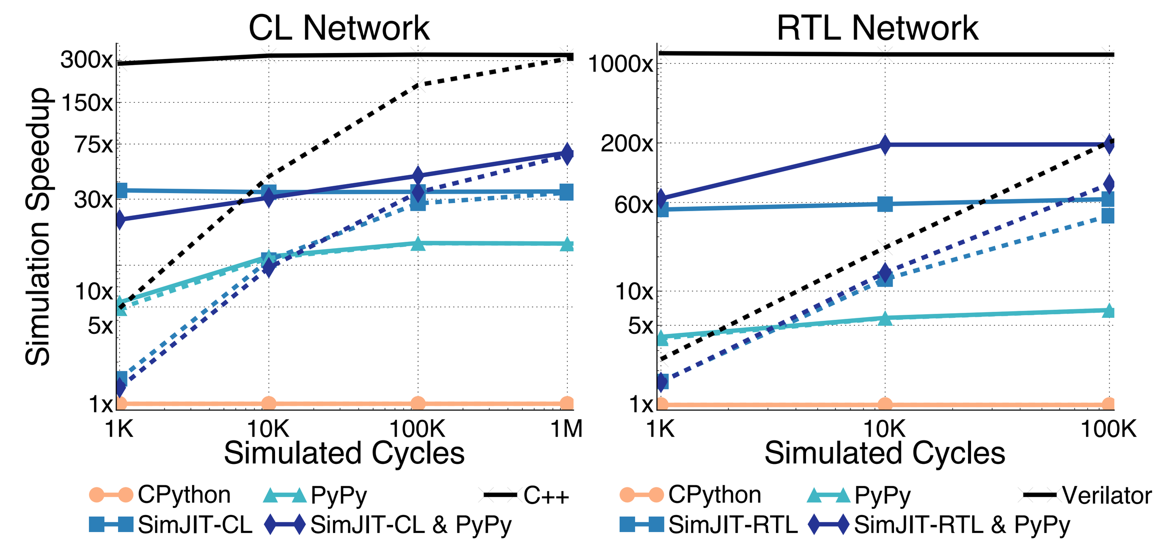
Note that while SimJIT-RTL is fairly robust and ready for use in most research flows, SimJIT-CL is considered "alpha" software that only works for a limited set of models.

Below we show the performance benefits of our SimJIT specializers for a simple 8x8 mesh network simulated near saturation. Pure Python simulation using the default CPython interpreter observes a slowdown of 300x/1200x when compared to a hand-written C++ CL model/Verilog RTL model. The use of PyPy, an alternative JIT-optimizing Python interpreter, can automatically improve performance of FL/CL/RTL simulation by 25x/15x/5x, but a large gap remains compared to C++.

SimJIT significantly improves CL/RTL simulation speed over CPython, but sees even greater benefit when combined with PyPy to optimize the unspecialized Python simulation loop and wrapper code. SimJIT+PyPy is able to close the gap between C++ and Python to 4.5 × for CL models and 6 × for RTL models.

Solid lines show speedup of the simulation loop compared to CPython, dotted lines additionally include all codegen and compilation overheads. Solid lines approximate simulation with caching.





Future work aims to reduce this performance gap even further by generating more optimized Python-to-C interfaces and possibly even creating parallel simulators.

8 Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, a DARPA Young Faculty Award, and donations from Intel Corporation and Synopsys, Inc. The authors acknowledge and thank Shreesha Srinath and Berkin Ilbeyi for their valuable PyMTL models and thoughtful feedback, Edgar Munoz for his help writing PyMTL models, and Sean Clark and Matheus Ogleari for their help developing the C++ and Verilog mesh network models.

Publication: Appears in the Proceedings of the 47th Int'l Symp. on Microarchitecture (MICRO-47), Dec. 2014. **URL:** `http://www.csl.cornell.edu/~cbatten/pdfs/lockhart-pymtl-micro2014.pdf` **Contact Author:** Derek Lockhart, 364 Upson Hall, Ithaca, NY 14853, `dml257@cornell.edu`