Appears in the Proceedings of the 47th Int'l Symp. on Microarchitecture (MICRO-47), December 2014

PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{dml257,gdz4,cbatten}@cornell.edu

***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.**

I. Introduction

Limitations in technology scaling have led to a growing interest in non-traditional system architectures that incorporate heterogeneity and specialization as a means to improve performance under strict power and energy constraints. Unfortunately, computer architects exploring these more exotic architectures generally lack existing physical designs to validate their power and performance models. The lack of validated models makes it challenging to accurately evaluate the computational efficiency of these designs [7,8,15–17,19]. As specialized accelerators become more integral to achieving the performance and energy goals of future hardware, there is a crucial need for researchers to supplement cycle-level simulation with algorithmic exploration and RTL implementation.

Future computer architecture research will place an increased emphasis on a methodology we call *modeling towards layout* (MTL). While computer architects have long leveraged multiple modeling abstractions (functional level, cycle level, register-transfer level) to trade off simulation time and accuracy, an MTL methodology aims to vertically integrate these abstractions for iterative refinement of a design from algorithm, to exploration, to implementation. Although an MTL methodology is especially valuable for prototyping specialized accelerators and exploring more exotic architectures, it has general value as a methodology for more traditional architecture research as well.

Unfortunately, attempts to implement an MTL methodology using existing publicly-available research tools reveals numerous practical challenges we call the *computer architecture research methodology gap*. This gap is manifested as the distinct languages, design patterns, and tools commonly used by functional level (FL), cycle level (CL), and register-transfer level (RTL) modeling. We believe the computer architecture research methodology gap exposes a critical need for a new vertically integrated framework to facilitate rapid design-space exploration and hardware implementation. Ideally such a framework would use a single specification language for FL, CL, and RTL modeling, enable multi-level simulations that mix models at different abstraction levels, and provide a path to design automation toolflows for extraction of credible area, energy, and timing results.

This paper introduces PyMTL[1], our attempt to construct such a unified, highly productive framework for FL, CL, and RTL modeling. PyMTL leverages a common high-productivity language (Python2.7) for behavioral specification, structural elaboration, and verification, enabling a rapid code-test-debug cycle for hardware modeling. Concurrent-structural modeling combined with latency-insensitive design allows reuse of test benches and components across abstraction levels while also enabling mixed simulation of FL, CL, and RTL models. A model/tool split provides separation of concerns between model specification and simulator generation letting architects focus on implementing hardware, not simulators. PyMTL's modular construction encourages extensibility: using elaborated model instances as input, users can write custom tools (also in Python) such as simulators, translators, analyzers, and visualizers. Python's glue language facilities provide flexibility by allowing PyMTL models and tools to be extended with C/C++ components or embedded within existing C/C++ simulators [42]. Finally, PyMTL serves as a productive hardware generation language for building synthesizable hardware templates thanks to an HDL translation tool that converts PyMTL RTL models into Verilog-2001 source.

Leveraging Python as a modeling language improves model conciseness, clarity, and implementation time [11,33], but comes at a significant cost to simulation time. For example, a pure Python cycle-level mesh network simulation in PyMTL exhibits a 300x slowdown when compared to an identical simulation written in C++. To ad-

[1]PyMTL loosely stands for [Py]thon framework for [M]odeling [T]owards [L]ayout and is pronounced the same as "py-metal".

dress this *performance-productivity gap* we take inspiration from the scientific computing community which has increasingly adopted *productivity-level languages* (e.g., MATLAB, Python) for computationally intensive tasks by replacing hand-written *efficiency-level language* code (e.g., C, C++) with dynamic techniques such as just-in-time (JIT) compilation [12, 27, 38] and selective-embedded JIT specialization [5, 10].

We introduce SimJIT, a custom just-in-time specializer which takes CL and RTL models written in PyMTL and automatically generates, compiles, links, and executes fast, Python-wrapped C++ code seamlessly within the PyMTL framework. SimJIT is both *selective* and *embedded* providing much of the benefits described in previous work on domain-specific embedded specialization [10]. SimJIT delivers significant speedups over CPython (up to 34× for CL models and 63× for RTL models), but sees even greater benefits when combined with PyPy, an interpreter for Python with a meta-tracing JIT compiler [6]. PyPy is able to optimize unspecialized Python code as well as hot paths between Python and C++, boosting the performance of SimJIT simulations by over 2× and providing a net speedup of 72× for CL models and 200× for RTL models. These optimizations mitigate much of the performance loss incurred by using a productivity-level language, closing the performance gap between PyMTL and C++ simulations to within 4–6×.

The contributions of this paper are the following:

1. We introduce PyMTL, a novel, vertically-integrated framework for concurrent-structural modeling and hardware design (Section III). PyMTL provides a unified environment for FL, CL, and RTL modeling and enables a path to EDA toolflows via generation of Verilog HDL.

2. We introduce and evaluate SimJIT, a selective embedded just-in-time specializer for PyMTL CL and RTL models (Section IV). SimJIT significantly improves the execution time of PyMTL simulations, providing near efficiency-level-language performance from productivity-level-language models.

II. Background

We consider three modeling abstractions of primary importance to computer architects, each with a distinct design methodology consisting of preferred languages, design patterns, and tools. These methodologies are summarized in Table I and described in greater detail below.

Functional-Level – FL models implement the *functionality* but not the timing constraints of a target. FL models are useful for exploring algorithms, performing fast emulation of hardware targets, and creating golden models for validation of CL and RTL models. The FL methodology usually has a data structure and algorithm-centric view, leveraging productivity-level languages such as MATLAB or Python to enable rapid implementation and verification. FL models often make use of open source algorithmic packages or toolboxes to aid construction of golden models where correctness is of primary concern. Performance-oriented FL models may use efficiency-level languages such as C or C++ when simulation time is the priority (e.g., instruction-set simulators).

Table I. Modeling Methodologies

	FL	**CL**	**RTL**
Modeling Languages	Productivity Level (PLL)	Efficiency Level (ELL)	Hardware Description (HDL)
	MATLAB/Python	*C/C++*	*Verilog/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	Third-party Algorithm Packages and Toolboxes	Computer Architecture Simulation Frameworks	Simulator Generators, Synthesis Tools, Verification Tools

Cycle-Level – CL models capture the *cycle-approximate behavior* of a hardware target. CL models attempt to strike a balance between accuracy, performance, and flexibility while exploring the timing behavior of hypothetical hardware organizations. The CL methodology places an emphasis on simulation speed and flexibility, leveraging high-performance efficiency-level languages like C++. Encapsulation and reuse is typically achieved through classic object-oriented software engineering paradigms, while timing is most often modeled using the notion of ticks or events. Established computer architecture simulation frameworks (e.g., ESESC [1], gem5 [4]) are frequently used to increase productivity as they typically provide libraries, simulation kernels, and parameterizable baseline models that allow for rapid design-space exploration.

Register-Transfer-Level – RTL models are *cycle-accurate*, *resource-accurate*, and *bit-accurate* representations of hardware. RTL models are built for the purpose of verification and synthesis of specific hardware implementations. The RTL methodology uses dedicated hardware description languages (HDLs) such as SystemVerilog and VHDL to create bit-accurate, synthesizable hardware specifications. Language primitives provided by HDLs are designed specifically for describing hardware: encapsulation is provided using port-based interfaces, composition is performed via structural connectivity, and logic is described using combinational and synchronous concurrent blocks. These HDL specifications are passed to simulators for evaluation/verification and EDA toolflows for collection of area, energy, timing estimates and construction of physical FPGA/ASIC prototypes. Originally intended for the design and verification of individual hardware instances, traditional HDLs are not well suited for extensive design space exploration [2, 40, 41].

The distinct languages, design patterns, and tools utilized by each abstraction create a *computer architecture research methodology gap*. This methodology gap introduces intellectual and technical barriers that make it challenging to transition *between* modeling abstractions and even more challenging to create an integrated flow *across* modeling abstractions. Industry is able to bridge this gap by utilizing their considerable resources to build large design teams. For academic research groups with limited resources and manpower, bridging this gap can be exceedingly difficult often resulting in over emphasis on a single level of abstraction.

A. *Mechanisms for Closing the Methodology Gap*

The design of PyMTL has received considerable inspiration from mechanisms introduced in prior work. We list these mechanisms below and briefly discuss how each can address challenges contributing to the methodology gap.

Concurrent-Structural Frameworks – Concurrent-structural frameworks provide hardware-inspired constructs for modeling port-based interfaces, concurrent execution, and structural composition. Vachharajani et. al have shown these constructs address the *mapping problem* inherent to CL models written in sequential, object-oriented languages, greatly improving clarity, accuracy, and component reuse [46]. HDLs for RTL design generally provide these constructs natively, however, a few general-purpose language, cycle-level simulation frameworks have adopted similar features (e.g., Liberty [46,47], Cascade [18], and SystemC [29]).

Unified Modeling Languages – Unified modeling languages enable specification of multiple modeling abstractions using a single description language. The use of a single specification language greatly reduces cognitive overhead for designers who would otherwise need expertise in multiple design languages. SystemC was proposed as a C++ language for multiple modeling tasks [29] including FL/CL/transaction-level modeling and RTL design (using a synthesizable subset), but has primarily seen wide adoption for virtual system prototyping and high-level synthesis.

Hardware Generation Languages – *Hardware generation languages* (HGLs) are hardware design languages that enable the construction of highly-parameterizable *hardware templates* [40]. HGLs facilitate design space exploration at the register-transfer level, and some HGLs additionally improve the productivity of RTL design through the introduction of higher-level design abstractions. Examples of HGLs include Genesis II [41], Chisel [2], and Bluespec [26]

HDL Integration – HDL integration provides mechanisms for native cosimulation of Verilog/VHDL RTL with FL/CL models written in more flexible general-purpose languages. Such integration accelerates RTL verification by supporting fast multilevel simulation of Verilog components with CL models, and enabling embedding of FL/CL golden models within Verilog for test bench validation. Grossman et al. noted that the unique HDL integration techniques in the Cascade simulator, such as interface binding, greatly assisted hardware validation of the Anton supercomputer [18].

SEJITS – Selective embedded just-in-time specialization (SEJITS) pairs domain-specific embedded languages (DSELs) [21] with DSEL-specific JIT compilers to provide runtime generation of optimized, platform-specific implementations from high-level descriptions. SEJITS enables efficiency-level language (ELL) performance from productivity-level language (PLL) code, significantly closing the *performance-productivity gap* for domain specific computations [10]. As an additional benefit, SEJITS greatly simplifies the construction of new domain-specific abstractions and high-performance JIT specializers by embedding specialization machinery within PLLs like Python.

```
 1 class MyModel( Model ):
 2   def __init__( s, constructor_params ):
 3
 4     # input  port  declarations
 5     # output port  declarations
 6     # other member declarations
 7
 8     # wire      declarations
 9     # submodule declarations
10     # connectivity statements
11     # concurrent logic specification
12
13     # more connectivity statements
14     # more concurrent logic specification
```

Figure 1. PyMTL Model Template – A basic skeleton of a PyMTL model. Elaboration logic allows mixing of wire declarations, submodule declarations, structural connectivity, and concurrent logic definitions.

Latency-Insensitive Interfaces – While more of a best-practice than explicit mechanism, consistent use of latency-insensitive interfaces at module boundaries is key to constructing libraries of interoperable FL, CL, and RTL models. Latency-insensitive protocols provide *control abstraction* through module-to-module stall communication, significantly improving component composability, design modularity, and facilitating greater test re-use [9,47].

III. PyMTL: A Unified Framework Enabling Modeling Towards Layout

PyMTL is a proof-of-concept framework designed to provide a unified environment for constructing FL, CL, and RTL models. The PyMTL framework consists of a collection of classes implementing a concurrent-structural DSEL within Python for hardware modeling, as well as a collection of tools for simulating and translating those models. The dynamic typing and reflection capabilities provided by Python enable succinct model descriptions, minimal boilerplate, and expression of flexible and highly parameterizable behavioral and structural components. The use of a popular, general-purpose programming language provides numerous benefits including access to mature numerical and algorithmic libraries, tools for test/development/debug, as well as access to the knowledge-base of a large, active development community.

A. *PyMTL Models*

PyMTL models are described in a concurrent-structural fashion: interfaces are port-based, logic is specified in concurrent logic blocks, and components are composed structurally. Users define model implementations as Python classes that inherit from `Model`. An example PyMTL class skeleton is shown in Figure 1. The `__init__` model constructor (lines 2–14) both executes *elaboration-time* configuration and declares *run-time* simulation logic. Elaboration-time configuration specializes model construction based on

```
1 class Register( Model ):
2   def __init__(s, nbits):
3     type  = Bits( nbits )
4     s.in_ = InPort ( type )
5     s.out = OutPort( type )
6
7     @s.tick_rtl
8     def seq_logic():
9       s.out.next = s.in_
```

```
1 class Mux( Model ):
2   def __init__(s, nbits, nports):
3     s.in_ = InPort[nports](nbits)
4     s.sel = InPort (bw(nports))
5     s.out = OutPort(nbits)
6
7     @s.combinational
8     def comb_logic():
9       s.out.value = s.in_[s.sel]
```

```
1 class MuxReg( Model ):
2   def __init__( s, nbits=8, nports=4 ):
3     s.in_ = [ InPort( nbits ) for x in range( nports ) ]
4     s.sel = InPort ( bw( nports ) )
5     s.out = OutPort( nbits )
6
7     s.reg_ = Register( nbits )
8     s.mux  = Mux     ( nbits )
9
10    s.connect( s.sel, s.mux.sel )
11    for i in range( nports ):
12      s.connect( s.in_[i], s.mux.in_[i] )
13    s.connect( s.mux.out,  s.reg_.in_ )
14    s.connect( s.reg_.out, s.out      )
```

Figure 2. PyMTL Example Models – Basic RTL models demonstrating sequential, combinational, and structural components in PyMTL. Powerful construction and elaboration logic enables design of highly-parameterizable components, while remaining fully Verilog translatable.

user-provided parameters. This includes the model interface (number, direction, message type of ports), internal constants, and structural hierarchy (wires, submodels, connectivity). Run-time simulation logic is defined using nested functions decorated with annotations that indicate their simulation-time execution behavior. Provided decorators include `@s.combinational` for combinational logic and `@s.tick_fl`, `@s.tick_cl`, and `@s.tick_rtl` for FL, CL, and RTL sequential logic, respectively. The semantics of signals (ports and wires) differ depending on whether they are updated in a combinational or sequential logic block. Signals updated in combinational blocks behave like wires; they are updated by writing their `.value` attributes and the concurrent block enclosing them only executes when its sensitivity list changes. Signals updated in sequential blocks behave like registers; they are updated by writing their `.next` attributes and the concurrent block enclosing them executes once every simulator cycle. Much like Verilog, submodel instantiation, structural connectivity, and behavioral logic definitions can be intermixed throughout the constructor.

A few simple PyMTL model definitions are shown in Figure 2. The `Register` model consists of a constructor that declares a single input and output port (lines 4–5) as well as a sequential logic block using a `s.tick_rtl` decorated nested function (lines 7–9). Ports are parameterizable by message type, in this case a `Bits` fixed-bitwidth message of size `nbits` (line 3). Due to the pervasive use of the `Bits` message type in PyMTL RTL modeling, syntactic sugar has been added such that `InPort(4)` may be used in place of the more explicit `InPort(Bits(4))`. We use this shorthand for the remainder of our examples. The `Mux` model is parameterizable by bitwidth and number of ports: the input is declared as a list of ports using a custom shorthand provided by the PyMTL framework (line 3), while the select port bitwidth is calculated using a user-defined function `bw` (line 4). A single combinational logic block is defined during elaboration (lines 8–9). No explicit sensitivity list is necessary as this is automatically inferred during simulator construction. The `MuxReg` model structurally composes `Register` and `Mux` models by instantiating them like normal Python objects (lines 7–9) and connecting their ports via the `s.connect()` method (lines 10–14). Note that it is not necessary to declare temporary wires in order to connect submodules as ports can simply be directly connected. A *list comprehension* is used to instantiate the input port list of `MuxReg` (line 3). This Python idiom is commonly used in PyMTL design to flexibly construct parameterizable lists of ports, wires, and submodules.

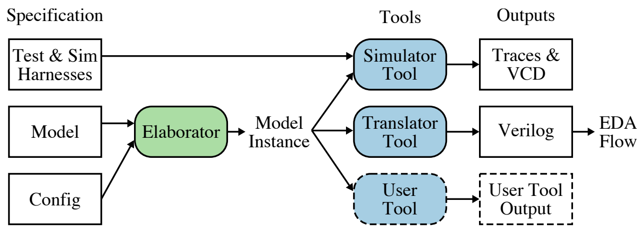


Figure 3. PyMTL Methodology – A model and configuration are elaborated into a model instance; tools manipulate the model instance to simulate or translate the design.

The examples in Figure 2 also provide a sample of PyMTL models that are fully translatable to synthesizable Verilog HDL. Translatable models must (1) describe all behavioral logic within within `s.tick_rtl` and `s.combinational` blocks, (2) use only a restricted, translatable subset of Python for logic within these blocks, and (3) pass all data using ports or wires with fixed-bitwidth message types (like `Bits`). While these restrictions limit some of the expressive power of Python, PyMTL provides mechanisms such as `BitStructs`, `PortBundles`, and type inference of local temporaries to improve the succinctness and productivity of translatable RTL modeling in PyMTL. Purely structural models like `MuxReg` are always translatable if all child models are translatable. This enables the full power of Python to be used during elaboration. Even greater flexibility is provided to non-translatable FL and CL models as they may contain arbitrary Python code within their `@s.tick_fl` and `@s.tick_cl` behavioral blocks. However, SimJIT can only specialize CL models that utilize a limited subset of the Python language (discussed in Section IV). Examples of FL and CL models, shown in Figures 7, 8, and 10, will be discussed in further detail in Sections III-C and III-D.

B. PyMTL Tools

The software architecture of the PyMTL framework is shown in Figure 3. User-defined models are combined with their configuration parameters to construct and elaborate model classes into model instances. Model instances act as in-memory representations of an elaborated design that can be accessed, inspected, and manipulated by various tools, just like a normal Python object. For example, the `TranslationTool` takes PyMTL RTL models, like those in Figure 2, inspects their structural hierarchy, connectivity,

```
1 @pytest.mark.parametrize(
2   'nbits,nports', [( 8, 2), ( 8, 3), ( 8, 4), ( 8, 8)
3                    (16, 2), (16, 3), (16, 4), (16, 8),
4                    (32, 2), (32, 3), (32, 4), (32, 8)]
5 )
6 def test_muxreg( nbits, nports, test_verilog ):
7   model = MuxReg( nbits, nports )
8   model.elaborate()
9   if test_verilog:
10    model = TranslationTool( model )
11
12  sim = SimulationTool( model )
13  for inputs, sel, output in gen_vectors(nbits,nports):
14    for i, val in enumerate( inputs ):
15      model.in_[i].value = val
16    model.sel.value = sel
17    sim.cycle()
18    assert model.out == output
```

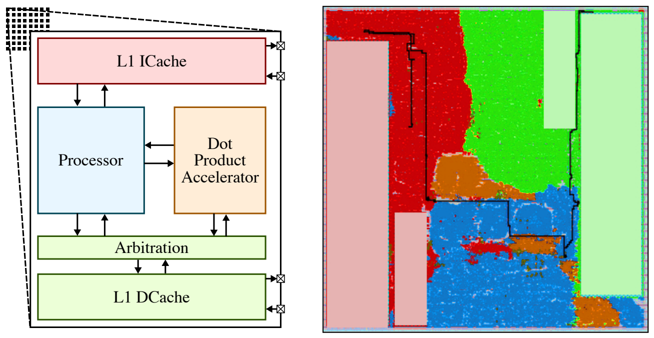
Figure 4. PyMTL Test Harness – The PyMTL `SimulationTool` and py.test package are used to simulate and verify the `MuxReg` module in Figure 2. A command-line flag uses the `TranslationTool` to automatically convert the `MuxReg` model into Verilog and test it within the same harness.

and concurrent logic, then uses this information to generate synthesizable Verilog that can be passed to an EDA toolflow. Similarly, the `SimulationTool` inspects elaborated models to automatically register concurrent logic blocks, detect sensitivity lists, and analyze the structure of connected ports to generate optimized Python simulators. The modular nature of this model/tool split encourages extensibility making it easy 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.

The PyMTL framework uses the open-source testing package py.test [34] along with the provided `SimulationTool` to easily create extensive unit-test suites for each model. One such unit-test can be seen in Figure 4. After instantiating and elaborating a PyMTL model (lines 7–8), the test bench constructs a simulator using the `SimulationTool` (line 12) and tests the design by setting input vectors, cycling the simulator, and asserting outputs (lines 14–18). A number of powerful features are demonstrated in this example: the py.test `@parametrize` decorator instantiates a large number of test configurations from a single test definition (lines 1–5), user functions are used to generate configuration-specific test vectors (line 13), and the model can be automatically translated into Verilog and verified within the same test bench by simply passing the `--test-verilog` flag at the command line (lines 9–10). In addition, py.test can provide test coverage statistics and parallel test execution on multiple cores or multiple machines by importing additional py.test plugins [35, 36].

C. *PyMTL by Example: Accelerator Coprocessor*

In this section, we demonstrate how PyMTL can be used to model, evaluate, and implement a simple accelerator for dot product computations. A *modeling towards layout* methodology is used to refine the accelerator from algorithm to implementation. Computer architects are rarely concerned only with the performance of a single component, rather we aim to



(a) Block Diagram (b) Post-Place-and-Route Layout

Figure 5. Hypothetical Heterogeneous Architecture – (a) Accelerator-augmented compute tiles interconnected by an on-chip network; (b) Synthesized, placed, and routed layout of compute tile shown in (a). Processor, cache, and accelerator RTL for this design were implemented and tested entirely in PyMTL, automatically translated into Verilog HDL, then passed to a Synopsys toolflow. Processor shown in blue, accelerator in orange, L1 caches in red and green, and critical path in black.

determine how a given mechanism may impact system performance as a whole. With this in mind, we implement our accelerator in the context of the hypothetical heterogeneous system shown in Figure 5(a). This system consists of numerous compute tiles interconnected by an on-chip network. This section will focus on modeling a dot product accelerator within the context of a tile containing a simple RISC processor and L1 caches. The accelerator is implemented as a coprocessor which shares a port to the L1 data cache with the processor. Section III-D will investigate a simple mesh network that might interconnect such tiles.

Functional Level – Architects build an FL model as a first step in the design process to familiarize themselves with an algorithm and create a golden model for validating more detailed implementations. Figure 6 demonstrates two basic approaches to constructing a simple FL model. The first approach (lines 1–2) manually implements the dot product algorithm in Python. This approach provides an opportunity for the designer to rapidly experiment with alternative algorithm implementations. The second approach (lines 4–5) simply calls the `dot` library function provided by the numerical package NumPy [28]. This approach provides immediate access to a verified, optimized, high-performance golden reference.

Unfortunately, integrating such FL implementations into an architecture framework can be a challenge. Our accelerator is designed as a coprocessor that interacts with both a processor and memory, so the FL model must implement communication protocols to interact with the rest of the system. This is a classic example of the methodology gap.

Figure 7 demonstrates a PyMTL FL model for the dot-product accelerator capable of interacting with FL, CL, and RTL models of the processor and memory. While more verbose than the simple implementations in Figure 6, the `DotProductFL` model must additionally control interactions with the processor, memory, and accelerator state. This additional complexity is greatly simplified by several PyMTL provided components: `ReqRespBundles` encap-

```
def dot_product_manual( src0, src1 ):
  return sum( [x*y for x,y in zip(src0, src1)] )

def dot_product_library( src0, src1 ):
  return numpy.dot( src0, src1 )
```

Figure 6. Functional Dot Product Implementation – A functional implementation of the dot product operator. We show both a manual implementation in Python, as well as a higher-performance library implementation.

```
class DotProductFL( Model ):
  def __init__( s, mem_ifc_types, cpu_ifc_types ):
    s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
    s.mem_ifc = ParentReqRespBundle( mem_ifc_types )

    s.cpu  = ChildReqRespQueueAdapter( s.cpu_ifc )
    s.src0 = ListMemPortAdapter      ( s.mem_ifc )
    s.src1 = ListMemPortAdapter      ( s.mem_ifc )

    @s.tick_fl
    def logic():
      s.cpu.xtick()
      if not s.cpu.req_q.empty() and not s.cpu.resp_q.full():
        req = s.cpu.get_req()
        if req.ctrl_msg == 1:
          s.src0.set_size( req.data )
          s.src1.set_size( req.data )
        elif req.ctrl_msg == 2: s.src0.set_base( req.data )
        elif req.ctrl_msg == 3: s.src1.set_base( req.data )
        elif req.ctrl_msg == 0:
          result = numpy.dot( s.src0, s.src1 )
          s.cpu.push_resp( result )
```

Figure 7. PyMTL DotProductFL Accelerator – Concurrent-structural modeling allows composition of FL models with CL and RTL models, but introduces the need to implement communication protocols. Proxies provide programmer-friendly interfaces that hide the complexities of these protocols.

```
class DotProductCL( Model ):
  def __init__( s, mem_ifc_types, cpu_ifc_types ):
    s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
    s.mem_ifc = ParentReqRespBundle( mem_ifc_types )

    s.cpu     = ChildReqRespQueueAdapter ( s.cpu_ifc )
    s.mem     = ParentReqRespQueueAdapter( s.mem_ifc )

    s.go    = False
    s.size  = 0
    s.src0  = 0
    s.src1  = 0
    s.data  = []
    s.addrs = []

    @s.tick_cl
    def logic():
      s.cpu.xtick()
      s.mem.xtick()

      if s.go:

        if s.addrs and not s.mem.req_q.full():
          s.mem.push_req( mreq( s.addrs.pop() ) )
        if not s.mem.resp_q.empty():
          s.data.append( s.mem.get_resp() )

        if len( s.data ) == s.size*2:
          result = numpy.dot( s.data[0::2], s.data[1::2] )
          s.cpu.push_resp( result )
          s.go = False

      elif not s.cpu.req_q.empty() and not s.cpu.resp_q.full():
        req = s.cpu.get_req()
        if   req.ctrl_msg == 1: s.size = req.data
        elif req.ctrl_msg == 2: s.src0 = req.data
        elif req.ctrl_msg == 3: s.src1 = req.data
        elif req.ctrl_msg == 0:
          s.addrs = gen_addresses( s.size, s.src0, s.src1 )
          s.go    = True
```

Figure 8. PyMTL DotProductCL Accelerator – Python's high-level language features are used to rapid-prototype a cycle-approximate model with pipelined memory requests. Queue-based, latency insensitive interfaces provide backpressure which naturally implements stall logic.

```
class DotProductRTL( Model ):
  def __init__( s, mem_ifc_types, cpu_ifc_types ):
    s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
    s.mem_ifc = ParentReqRespBundle( mem_ifc_types )

    s.dpath = DotProductDpath( mem_ifc_types, cpu_ifc_types )
    s.ctrl  = DotProductCtrl ( mem_ifc_types, cpu_ifc_types )
    s.connect_auto( s.dpath, s.ctrl )

class DotProductDpath( Model ):
  def __init__( s, mem_ifc_types, cpu_ifc_types ):
    s.cpu_ifc = ChildReqRespBundle ( cpu_ifc_types )
    s.mem_ifc = ParentReqRespBundle( mem_ifc_types )
    s.cs      = InPort ( CtrlSignals()   )
    s.ss      = OutPort( StatusSignals() )

    #--- Stage M: Memory Request ------------------------------
    s.count       = Wire( cpu_ifc_types.req .data.nbits )
    s.size        = Wire( cpu_ifc_types.req .data.nbits )
    s.src0_addr_M = Wire( mem_ifc_types.req .addr.nbits )
    s.src1_addr_M = Wire( mem_ifc_types.req .addr.nbits )

    @s.tick_rtl
    def stage_seq_M():
      ctrl_msg = s.cpu_ifc.req_msg .ctrl_msg
      cpu_data = s.cpu_ifc.req_msg .data

      if s.cs.update_M:
        if   ctrl_msg == 1: s.size       .next = cpu_data
        elif ctrl_msg == 2: s.src0_addr_M.next = cpu_data
        elif ctrl_msg == 3: s.src1_addr_M.next = cpu_data
        elif ctrl_msg == 0: s.ss.go      .next = True

      if   s.cs.count_clear_M: s.count.next = 0
      elif s.cs.count_en_M:    s.count.next = s.count + 1

    @s.combinational
    def stage_comb_M():
      if s.cs.baddr_sel_M == src0: base_addr_M = s.src0_addr_M
      else:                        base_addr_M = s.src1_addr_M

      s.mem_ifc.req_msg.type.value = 0
      s.mem_ifc.req_msg.addr.value = base_addr_M + (s.count<<2)

      s.ss.last_item_M.value = s.count == (s.size - 1)

    #--- Stage R: Memory Response -----------------------------
    s.src0_data_R = Wire( mem_ifc_types.resp.data.nbits )
    s.src1_data_R = Wire( mem_ifc_types.resp.data.nbits )

    @s.tick_rtl
    def stage_seq_R():
      mem_data = s.mem_ifc.resp_msg.data
      if s.cs.src0_en_R: s.src0_data_R.next = mem_data
      if s.cs.src1_en_R: s.src1_data_R.next = mem_data

    #--- Stage X: Execute Multiply ----------------------------
    s.result_X = Wire( cpu_ifc_types.req.data.nbits )
    s.mul      = IntPipelinedMultiplier(
                   nbits   = cpu_ifc_types.req.data.nbits,
                   nstages = 4,
                 )
    s.connect_dict( { s.mul.op_a    : s.src0_data_R,
                      s.mul.op_b    : s.src1_data_R,
                      s.mul.product : s.result_X     } )

    #--- Stage A: Accumulate ----------------------------------
    s.accum_A   = Wire( cpu_ifc_types.resp.data.nbits )
    s.accum_out = Wire( cpu_ifc_types.resp.data.nbits )

    @s.tick_rtl
    def stage_seq_A():
      if   s.reset or s.cs.accum_clear_A:
        s.accum_A.next = 0
      elif s.cs.accum_en_A:
        s.accum_A.next = s.accum_out

    @s.combinational
    def stage_comb_A():
      s.accum_out.value = s.result_X + s.accum_A
      s.cpu_ifc.resp_msg.value = s.accum_A
```

Figure 9. PyMTL DotProductRTL Accelerator – RTL implementation of a dot product accelerator in PyMTL, control logic is not shown for brevity. Python DSEL combines familiar HDL syntax with powerful elaboration.

sulate collections of signals needed for latency-insensitive communication with the processor and memory (lines 3–4), the ChildReqRespQueueAdapter provides a simple queue-based interface to the ChildReqRespBundle and automatically manages latency-insensitive communication to the processor (lines 6, 12–14, 22), and the ListMemPortAdapter provides a list-like interface to the ParentReqRespBundle and automatically manages the latency-insensitive communication to the memory (lines 7–8).

Of particular note is the ListMemPortAdapter which allows us to reuse numpy.dot from Figure 6 without modification. This is made possible by the greenlets concurrency package [37] that enables proxying array index accesses into memory request and response transactions over the latency-insensitive, port-based model interfaces. The exact mechanism of this functionality is beyond the scope of this paper, but the effect is an ability to compose existing, library-provided utility functions with port-based processors and memories to quickly create a target for validation and software co-development.

Cycle Level – Construction of a cycle-level model provides a sense of the timing behavior of a component, enabling architects to estimate system-level performance and make first-order design decisions prior to building a detailed RTL implementation. Figure 8 shows an implementation of the DotProductCL model in PyMTL. Rather than faithfully emulating detailed pipeline behavior, this model simply aims to issue and receive memory requests in a cycle-approximate manner by implementing a simple pipelining scheme. Like the FL model, the CL model takes advantage of higher-level PyMTL library constructs such as the ReqResponseBundles and QueueAdapters to simplify the design, particularly with regards to interfacing with external communication protocols (lines 3–7). Logic is simplified by pre-generating all memory requests and storing them in a list once the go signal is set (line 39), this list is used to issue requests to memory as backpressure allows (lines 23–24). Data is received from the memory in a pipelined manner and stored in another list (lines 25–26). Once all data is received it is separated, passed into numpy.dot, and returned to the processor (lines 28–31).

Because DotProductCL exposes an identical port-based interface to DotProductFL, construction of the larger tile can be created using an incremental approach. These steps include writing unit-tests based on golden FL model behavior, structurally composing the FL model with the processor and memory to validate correct system behavior, verifying the CL model in isolation by reusing the FL unit tests, and finally swapping the FL model and the CL model for final system-level integration testing. This pervasive testing gives us confidence in our model, and the final composition of the CL accelerator with CL or RTL memory and processor models allow us to evaluate system-level behavior.

To estimate the performance impact of our accelerator, a more detailed version of DotProductCL is combined with CL processor and cache components to create a CL tile. This accelerator-augmented tile is used to execute a 1024x1024 matrix-vector multiplication kernel (a computation consisting of 1024 dot products). The resulting CL simulation estimates our accelerator will provide a 2.9× speedup over a traditional scalar implementation with loop-unrolling optimizations.

Register-Transfer Level – The CL model allowed us to quickly obtain a cycle-approximate performance estimate for our accelerator-enhanced tile in terms of *simulated cycles*, however, *area*, *energy*, and *cycle time* are equally important metrics that must also be considered. Unfortunately, accurately predicting these metrics from high-level models is notoriously difficult. An alternative approach is to use an industrial EDA toolflow to extract estimates from a detailed RTL implementation. Building RTL is often the most appropriate approach for obtaining credible metrics, particularly when constructing exotic accelerator architectures.

PyMTL attempts to address many of the challenges associated with RTL design by providing a productive environment for constructing highly parameterizable RTL implementations. Figure 9 shows the top-level and datapath code for the DotProductRTL model. The PyMTL DSEL provides a familiar Verilog-inspired syntax for traditional combinational and sequential bit-level design using the Bits datatype, but also layers more advanced constructs and powerful elaboration-time capabilities to improve code clarity. A concise top-level module definition is made possible by the use of PortBundles and the connect_auto method, which automatically connects parent and child signals based on signal name (lines 1-8). BitStructs are used as message types to connect control and status signals (lines 14–15), improving code clarity by providing named access to bitfields (lines 30, 36–37, 41). Mixing of wire declarations, sequential logic definitions, combinational logic definitions, and parameterizable submodule instantiations (lines 58–61) enable code arrangements that clearly demarcate pipeline stages. In addition, DotProductRTL shares the same parameterizable interface as the FL and CL models enabling reuse of unmodified FL and CL test benches for RTL verification before automatic translation into synthesizable Verilog.

Figure 5(b) shows a synthesized, placed, and routed implementation of the tile in Figure 5(a), including a 5-stage RISC processor, dot-product accelerator, instruction cache, and data cache. The entire tile was implemented, simulated, and verified in PyMTL before being translated into Verilog and passed to a Synopsys EDA toolflow. Using this placed-and-routed design we were able to extract area, energy, and timing metrics for the tile. The dot-product accelerator added an area overhead of 4% ($0.02\,\text{mm}^2$) and increased the cycle time of the tile by approximately 5%. Fortunately, the improvement in simulated cycles resulted in a net execution time speedup of 2.74×. This performance improvement must be weighed against the overheads and the fact that the accelerator is only useful for dot product computations.

D. PyMTL by Example: Mesh Network

The previous section evaluated adding an accelerator to a single tile from Figure 5(a) in isolation, however, this tile is just one component in a much larger multi-tile system. In this

```
class NetworkFL( Model ):
  def __init__( s, nrouters, nmsgs,
                   data_nbits, nentries ):

    # ensure nrouters is a perfect square
    assert sqrt( nrouters ) % 1 == 0

    net_msg = NetMsg( nrouters, nmsgs, data_nbits )
    s.in_   = InValRdyBundle [ nrouters ]( net_msg )
    s.out   = OutValRdyBundle[ nrouters ]( net_msg )

    s.nentries     = nentries
    s.output_fifos = [deque() for x in range(nrouters)]

    @s.tick_fl
    def network_logic():

      # dequeue logic
      for i, outport in enumerate( s.out ):
        if outport.val and outport.rdy:
          s.output_fifos[ i ].popleft()

      # enqueue logic
      for inport in s.in_:
        if inport.val and inport.rdy:
          dest = inport.msg.dest
          msg  = inport.msg[:]
          s.output_fifos[ dest ].append( msg )

      # set output signals
      for i, fifo in enumerate( s.output_fifos ):

        is_full  = len( fifo ) == s.nentries
        is_empty = len( fifo ) == 0

        s.out[ i ].val.next = not is_empty
        s.in_[ i ].rdy.next = not is_full
        if not is_empty:
          s.out[ i ].msg.next = fifo[ 0 ]
```

Figure 10. FL Network – Functional-level model emulates the functionality but not the timing of a mesh network. this is behaviorally equivalent to an ideal crossbar. Resource constraints exist only on the model interface: multiple packets can enter the same queue in a single cycle, but only one packet may leave per cycle.

```
 1 class MeshNetworkStructural( Model ):
 2   def __init__( s, RouterType, nrouters, nmsgs,
 3                    data_nbits, nentries ):
 4
 5     # ensure nrouters is a perfect square
 6     assert sqrt( nrouters ) % 1 == 0
 7
 8     s.RouterType = RouterType
 9     s.nrouters   = nrouters
10     s.params     = [nrouters, nmsgs, data_nbits, nentries]
11
12     net_msg = NetMsg( nrouters, nmsgs, data_nbits )
13     s.in_   = InValRdyBundle [ nrouters ]( net_msg )
14     s.out   = OutValRdyBundle[ nrouters ]( net_msg )
15
16     # instantiate routers
17     R = s.RouterType
18     s.routers = \
19       [ R(x, *s.params) for x in range(s.nrouters) ]
20
21     # connect injection terminals
22     for i in xrange( s.nrouters ):
23       s.connect( s.in_[i],  s.routers[i].in_[R.TERM] )
24       s.connect( s.out[i],  s.routers[i].out[R.TERM] )
25
26     # connect mesh routers
27     nrouters_1D = int( sqrt( s.nrouters ) )
28     for j in range( nrouters_1D ):
29       for i in range( nrouters_1D ):
30         idx = i + j * nrouters_1D
31         cur = s.routers[idx]
32         if i + 1 < nrouters_1D:
33           east = s.routers[ idx + 1 ]
34           s.connect(cur.out[R.EAST], east.in_[R.WEST])
35           s.connect(cur.in_[R.EAST], east.out[R.WEST])
36         if j + 1 < nrouters_1D:
37           south = s.routers[ idx + nrouters_1D ]
38           s.connect(cur.out[R.SOUTH], south.in_[R.NORTH])
39           s.connect(cur.in_[R.SOUTH], south.out[R.NORTH])
```

Figure 11. Structural Mesh Network – Structurally composed network parameterized by network message type, network size, router buffering, and router type (a PyMTL model). `ValRdyBundles` significantly reduce structural connectivity complexity. Elaboration can use arbitrary Python while still remaining Verilog translatable as long as `RouterType` is translatable.

section, we will briefly explore the design and performance of as simple mesh network that might interconnect these tiles.

Functional Level – Verifying tile behavior in the context of a multi-tile system can be greatly simplified by starting with an FL network implementation. PyMTL's concurrent-structural modeling approach allows us to quickly write a port-based FL model of our mesh network (behaviorally equivalent to a "magic" single-cycle crossbar) and connect it with FL, CL, or RTL tiles in order to verify our tiles and to provide a platform for multi-tile software development. Figure 10 shows a full PyMTL implementation of an FL network.

Cycle Level – A CL mesh network emulating realistic network behavior is implemented to investigate network performance characteristics. Figure 11 contains PyMTL code for a structural mesh network. This model is designed to take a PyMTL router implementation as a parameter (line 2) and structurally compose instances of this router into a complete network (lines 16–39). This approach is particularly powerful as it allows us to easily instantiate the network with either FL, CL, or RTL router models to trade-off accuracy and simulation speed, or quickly swap out routers with different microarchitectures for either verification or evaluation purposes. To construct a simple CL network model, we use `MeshNetworkStructural` to construct an 8x8 mesh network composed of routers using XY-dimension ordered routing and elastic-buffer flow control. Simulations of this model allow us to quickly estimate this network has a zero-load latency of 13 cycles and saturates at an injection rate of 32%.

Register-Transfer Level – Depending on our design goals, we may want to estimate area, energy, and timing for a single router, the entire network in isolation, or the network with the tiles attached. An RTL network can be created using the same top-level structural code as in Figure 11 by simply passing in an RTL router implementation as a parameter. Structural code in PyMTL is always Verilog translatable as long as all leaf modules are also Verilog translatable.

IV. SimJIT: Closing the Performance-Productivity Gap

While the dynamic nature of Python greatly improves the expressiveness, productivity, and flexibility of model code, it significantly degrades simulation performance when compared to a statically compiled language like C++. We address this performance limitation by using a hybrid just-in-time optimization approach. We combine SimJIT, a custom just-in-time specializer for converting PyMTL models into optimized C++ code, with the PyPy meta-tracing JIT interpreter. Below we discuss the design of SimJIT and evaluate its performance on CL and RTL models.

A. *SimJIT Design*

SimJIT consists of two distinct specializers: SimJIT-CL for specializing cycle-level PyMTL models and SimJIT-RTL for specializing register-transfer-level PyMTL models. Figure 12 shows the software architecture of the SimJIT-CL and SimJIT-RTL specializers. Currently, the designer must manually invoke these specializers on their models, although future work could consider adding support to automatically traverse the model hierarchy to find and specialize appropriate CL and RTL models.

SimJIT-CL begins with an elaborated PyMTL model instance and uses Python's reflection capabilities to inspect the model's structural connectivity and concurrent logic blocks. We are able to reuse several model optimization utilities from the previously described `SimulationTool` to help in generating optimized C++ components. We also leverage the `ast` package provided by the Python Standard Library to implement translation of concurrent logic blocks into C++ functions. The translator produces both C++ source implementing the optimized model as well as a C interface wrapper so that this C++ source may be accessed via CFFI, a fast foreign function interface library for calling C code from Python. Once code generation is complete, it is automatically compiled into a C shared library using LLVM, then imported into Python using an automatically generated PyMTL wrapper. This process gives the library a port-based interface so that it appears as a normal PyMTL model to the user.

Similar to SimJIT-CL, the SimJIT-RTL specializer takes an elaborated PyMTL model instance and inspects it to begin the translation process. Unlike SimJIT-CL, SimJIT-RTL does not attempt to perform any optimizations, rather it directly translates the design into equivalent synthesizable Verilog HDL. This translated Verilog is passed to Verilator, an open-source tool for generating optimized C++ simulators from Verilog source [48]. We combine the verilated C++ source with a generated C interface wrapper, compile it into a C shared library, and once again wrap this in a generated PyMTL model.

While both SimJIT-CL and SimJIT-RTL can generate fast C++ components that significantly improve simulation time, the Python interface still has a considerable impact on simulation performance. We leverage PyPy to optimize the Python simulation loop as well as the hot-paths between the Python and C++ call interface, significantly reducing the overhead of using Python component wrappers. Compilation time of the specializer can also take a considerable amount of time, especially for SimJIT-RTL. For this reason, PyMTL includes support for automatically caching the results from translation for SimJIT-RTL. While not currently implemented, caching the results from translation for SimJIT-CL should be relatively straight-forward. In the next two sections, we examine the performance benefits of SimJIT and PyPy in greater detail, using the PyMTL models discussed in Sections III-C and III-D as examples.

B. *SimJIT Performance: Accelerator Tile*

We construct 27 different tile models at varying levels of detail by composing FL, CL, and RTL implementations of the processor (P), caches (C), and accelerator (A) for the compute tile in Figure 5(a). Each configuration is described as a tuple $\langle P, C, A \rangle$ where each entry is FL, CL, or RTL. Each configuration is simulated in CPython with no optimizations and also simulated again using both SimJIT and PyPy. For this experiment, a slightly more complicated dot product accelerator was used than the one described in Section III-C. SimJIT+PyPy runs applied SimJIT-RTL specialization to all RTL components in a model, whereas SimJIT-CL optimizations were only applied to the caches due to limitations of our current proof-of-concept SimJIT-CL specializer. Figure 13 shows the simulation performance of each run plotted against a "level of detail" (LOD) score assigned to each configuration. LOD is calculated such that LOD $= p + c + a$ where p, c, and a have a value corresponding to the model complexity: FL $= 1$, CL $= 2$, RTL $= 3$. Note that the LOD metric is not meant to be an exact measure of model accuracy but rather a high-level approximation of overall model complexity. Performance is calculated as the execution time of a configuration normalized against the execution time of a sim-

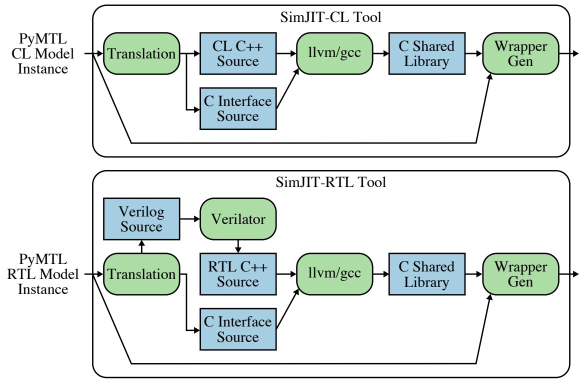


Figure 12. SimJIT Software Architecture – SimJIT consists of two specializers: one for CL models and one for RTL models. Each specializer can automatically translate PyMTL models into C++ and generate the appropriate wrappers to enable these C++ implementations to appear as standard PyMTL models.

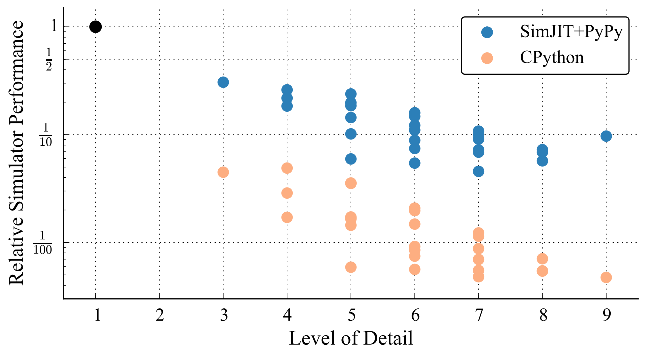


Figure 13. Simulator Performance vs. LOD – Simulator performance using CPython and SimJIT+PyPy with the processor, memory, and accelerator modeled at various levels of abstraction. Results are normalized against the pure ISA simulator using PyPy. Level of detail (LOD) is measured by allocating a score of one for FL, two for CL, and three for RTL and then summing across the three models. For example, a FL processor composed with a CL memory system and RTL accelerator would have an LOD of 1+2+3 = 6.

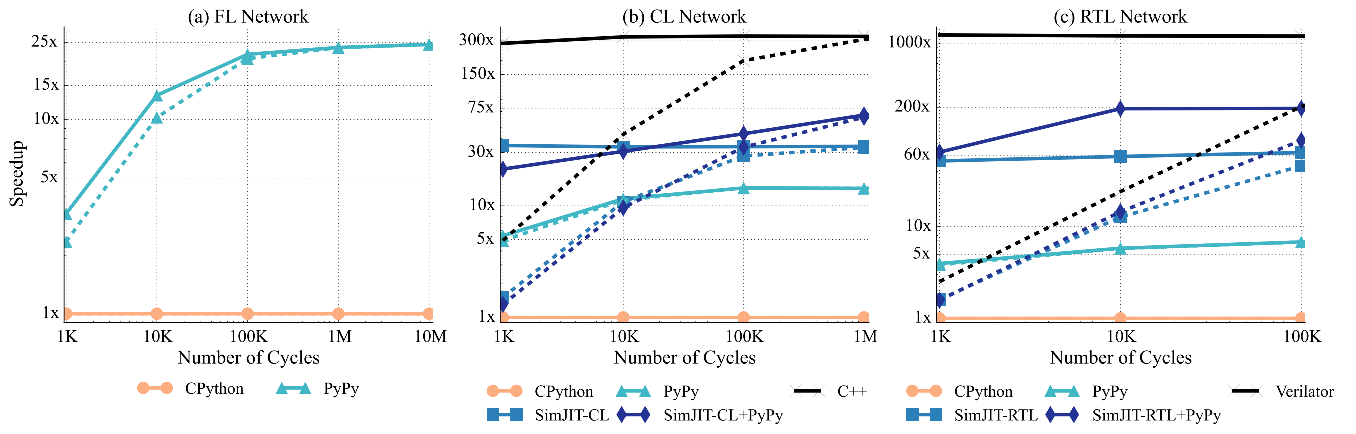


Figure 14. SimJIT Mesh Network Performance – Simulation of 64-node FL, CL, and RTL mesh network models operating near saturation. Dotted lines indicate total simulation-time speedup including all compilation and specialization overheads. Solid lines indicate speedup ignoring alloverheads shown in Figure 16. For CL and RTL models, the solid lines closely approximate the speedup seen with caching enabled. No SimJIT optimization exists for FL models, but PyPy is able to provide good speedups. SimJIT+PyPy brings CL/RTL execution time within 4×/6× of hand-written C++/optimized Verilator simulation, respectively.

ple object-oriented instruction-set simulator implemented in Python and executed using PyPy. This instruction-set simulator is given an LOD score of 1 since it consists of only a single FL component, and it is plotted at coordinate coordinate (1,1) in Figure 13.

A general downward trend is observed in relative simulation performance as LOD increases. This is due to the greater computational effort required to simulate increasingly detailed models, resulting in a corresponding increase in execution time. In particular, a significant drop in performance can be seen between the simple instruction-set simulator (LOD $=$ 1) and the ⟨FL,FL,FL⟩ configuration (LOD $= 3$). This gap demonstrates the costs associated with modular modeling of components, structural composition, and communication overheads incurred versus a monolithic implementation with a tightly integrated memory and accelerator implementation. Occasionally, a model with a high LOD will take less execution time than a model with low LOD. For CPython data points this is largely due to more detailed models taking advantage of pipelining or parallelism to reduce target execution cycles. For example, the FL model of the accelerator does not pipeline memory operations and therefore executes many more target cycles than the CL implementation. For SimJIT+PyPy data points the effectiveness of each specialization strategy and the complexity of each component being specialized plays an equally significant role. FL components only benefit from the optimizations provided by PyPy and in some cases may perform worse than CL or RTL models which benefit from both SimJIT and PyPy, despite their greater LOD. Section IV-C explores the performance characteristics of each specialization strategy in more detail.

Comparing the SimJIT+PyPy and CPython data points we can see that just-in-time specialization is able to significantly improve the execution time of each configuration, resulting in a vertical shift that makes even the most detailed models competitive with the CPython versions of simple models. Even better results could be expected if SimJIT-CL optimizations were applied to CL processor and CL accelerator models as well. Of particular interest is the ⟨RTL,RTL,RTL⟩ configuration (LOD $= 9$) which demonstrates better simulation performance than many less detailed configurations. This is because all subcomponents of the model can be optimized together as a monolithic unit, further reducing the overhead of Python wrapping. More generally, Figure 13 demonstrates the impact of two distinct approaches to improving PyMTL performance: (1) improvements that can be obtained *automatically* through specialization using SimJIT+PyPy, and (2) improvements that can be obtained *manually* by tailoring simulation detail via multi-level modeling.

C. SimJIT Performance: Mesh Network

We use the mesh network discussed in Section III-D to explore in greater detail the performance impact and overheads associated with SimJIT and PyPy. A network makes a good model for this purpose, since it allows us to flexibly configure size, injection rate, and simulation time to examine SimJIT's performance on models of varying complexity and under various loads. Figure 14 shows the impact of just-in-time specialization on 64-node FL, CL, and RTL mesh networks near saturation. All results are normalized to the performance of CPython. Dotted lines show speedup of *total* simulation time while solid lines indicate speedup after *subtracting the simulation overheads shown in Figure 16*. These overheads are discussed in detail later in this section. Note that the dotted lines in Figure 14 are the real speedup observed when running a single experiment, while the solid line is an approximation of the speedup observed when caching is available. Our SimJIT-RTL caching implementation is able to remove the *compilation* and *verilation* overheads (shown in Figure 16) so the solid line closely approximates the speedups seen when doing multiple simulations of the same model instance.

The FL network plot in Figure 14(a) compares only PyPy versus CPython execution since no embedded-specializer exists for FL models. PyPy demonstrates a speedup between 2–25× depending on the length of the simulation. The bend in the solid line represents the warm-up time associated with

PyPy's tracing JIT. After 10M target cycles the JIT has completely warmed-up and almost entirely amortizes all JIT overheads. The only overhead included in the dotted line is elaboration, which has a performance impact of less than a second.

The CL network plot in Figure 14(b) compares PyPy, SimJIT-CL, SimJIT-CL+PyPy, and a hand-coded C++ implementation against CPython. The C++ implementation is implemented using an in-house concurrent-structural modeling framework in the same spirit as Liberty [47] and Cascade [18]. It is designed to have cycle-exact simulation behavior with respect to the PyMTL model and is driven with an identical traffic pattern. The pure C++ implementation sees a speedup over CPython of up to 300× for a 10M-cycle simulation, but incurs a significant overhead from compilation time (dotted line). While this overhead is less important when model design has completed and long simulations are being performed for evaluation, this time significantly impacts the code-test-debug loop of the programmer, particularly when changing a module that forces a rebuild of many dependent components. An interpreted design language provides a significant productivity boost in this respect as simulations of less than 1K target cycles (often used for debugging) offer quicker turn around than a compiled language. For long runs of 10M target cycles, PyPy is able to provide a 12× speedup over CPython, SimJIT a speedup of 30×, and the combination of SimJIT and PyPy provides a speedup of 75× bringing us within 4× of hand-coded C++.

The RTL network plot in Figure 14(c) compares PyPy, SimJIT-RTL, SimJIT-RTL+PyPy, and a hand-coded Verilog implementation against CPython. For the Verilog network we use Verilator to generate a C++ simulator, manually write a C++ test harness, and compile them together to create a simulator binary. Again, the Verilog implementation has been verified to be cycle-exact with our PyMTL implementation and is driven using an identical traffic pattern. Due to the detailed nature of RTL simulation, Python sees an even greater performance degredation when compared to C++. For the longest running configuration of 10M target cycles, C++ observes a 1200× speedup over CPython. While this performance difference makes Python a non-starter for long running simulations, achieving this performance comes at a significant compilation overhead: compiling Verilator-generated C++ for the 64-node mesh network takes over 5 minutes using the relatively fast -O1 optimization level of GCC. PyPy has trouble providing significant speedups over more complicated designs, and in this case only achieves a 6× improvement over CPython. SimJIT-RTL provides a 63× speedup and combining SimJIT-RTL with PyPy provides a speedup of 200×, bringing us within 6× of verilated hand-coded Verilog.

To explore how simulator activity impacts our SimJIT speedups, we vary the injection rate of the 64-node mesh network simulations for both the CL and RTL models (see Figure 15). In comparison to CPython, PyPy performance is relatively consistent across loads, while SimJIT-CL and SimJIT-RTL see increased performance under greater load. SimJIT speedup ranges between 23–49× for SimJIT-CL+PyPy and 77–192× for SimJIT-RTL+PyPy. The curves of both plots

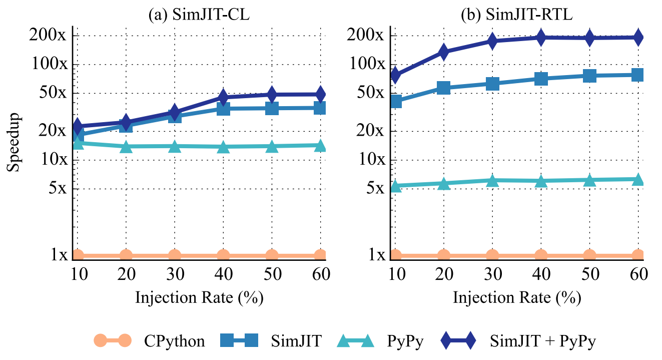
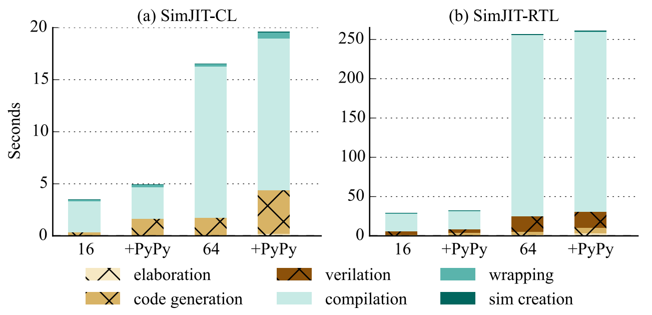


Figure 15. SimJIT Performance vs. Load – Impact of injection rate on a 64-node network simulation executing for 100K cycles. Heavier load results in longer execution times, enabling overheads to be amortized more rapidly for a given number of simulated cycles as more time is spent in optimized code.



config			elab	cgen	veri	comp	wrap	simc	total
CL	16	CPython	.02	.35	-	2.98	.11	.01	3.9
		PyPy	.04	1.64	-	2.99	.25	.01	4.9
	64	CPython	.08	1.69	-	14.51	.22	.02	16.5
		PyPy	.21	4.17	-	14.58	.60	.06	19.6
RTL	16	CPython	.41	.87	4.75	22.57	.13	.05	28.8
		PyPy	1.10	2.95	4.78	22.88	.28	.11	32.1
	64	CPython	1.84	3.48	20.09	230.42	.25	.67	256.8
		PyPy	3.58	7.12	20.20	228.57	.80	.66	260.9

Figure 16. SimJIT Overheads – Elaboration (elab), code generation (cgen), verilation (veri), compilation (comp), Python wrapping (wrap) and sim creation (simc) all contribute overhead to run-time creation of specializers. Compile time has the largest impact for both SimJIT-RTL and SimJIT-CL. Verilation also has a significant impact for SimJIT-RTL, especially for larger models. This step is not present in SimJIT-CL.

begin to flatten out at the network's saturation point near an injection rate of 30%. This is due to the increased amount of execution time being spent inside the network model during each simulation tick meaning more time is spent in optimized C++ code for the SimJIT configurations.

The overheads incurred by SimJIT-RTL and SimJIT-CL increase with larger model sizes due to the increased quantity of code that must be generated and compiled. Figure 16 shows these overheads for 4×4 and 8×8 mesh networks. These overheads are relatively modest for SimJIT-RTL at under 5 and 20 seconds for the 16- and 64-node meshes, respectively. The use of PyPy slightly increases the overhead of SimJIT. This is because SimJIT's elaboration, code generation, wrapping, and simulator creation phases are all too short to amortize PyPy's tracing JIT overhead. However, this slowdown is negligible compared to the significant speedups PyPy provides during simulation. SimJIT-RTL has an additional verilation phase, as well as significantly higher compilation times: 22 seconds for a 16-node mesh and 230 seconds for a 64-node mesh. Fortunately, the overheads for verilation, compilation, and wrapping can be converted into a one-time cost using SimJIT-RTL's simple caching scheme.

V. Related Work

A number of previous projects have proposed using Python for hardware design. Stratus, PHDL, and PyHDL generate HDL from parameterized structural descriptions in Python by using provided library blocks, but do not provide simulation capabilities or support for FL or CL modeling [3,20,23]. MyHDL uses Python as a hardware description language that can be simulated in a Python interpreter or translated to Verilog and VHDL [14,49]. SysPy is a tool intended to aid processor-centric SoC designs targeting FPGAs that integrates with existing IP and user-provided C source source [22]. PDSDL, enables behavioral and structural description of RTL models that can be simulated within a Python-based kernel, as well as translated into HDL. PDSDL was used in the construction of Trilobyte, a framework for refining behavioral processor descriptions into HDL [52,53]. Other than PDSDL, the above frameworks focus primarily on structural or RTL hardware descriptions and do not address higher level modeling. In addition, none attempt to address the performance limitations inherent to using Python for simulation.

Hardware generation languages help address the need for rapid design-space exploration and collection of area, energy, and timing metrics by making RTL design more productive. Genesis2 combined SystemVerilog with Perl scripts to create highly parameterizable hardware designs for the creation of chip generators [40,41]. Chisel is an HDL implemented as an DSEL within Scala. Hardware descriptions in Chisel are translated into to either Verilog HDL or C++ simulations. There is no Scala simulation of hardware descriptions [2]. BlueSpec is an HGL built on SystemVerilog that describes hardware using guarded atomic actions [26].

A number of other simulation frameworks have applied a concurrent-structural modeling approach to cycle-level simulation. The Liberty Simulation Environment argued that concurrent-structural modeling greatly improved understanding and reuse of components, but provided no HDL integration or generation [45–47]. Cascade is a concurrent-structural simulation framework used in the design and verification of the Anton supercomputers. Cascade provides tight integration with an RTL flow by enabling embedding of Cascade models within Verilog test harnesses as well as Verilog components within Cascade models [18]. SystemC also leverages a concurrent-structural design methodology that was originally intended to provide an integrated framework for multiple levels of modeling and refinement to implementation, including a synthesizable language subset. Unfortunately, most of these thrusts did not see wide adoption and SystemC is currently used primarily for the purposes of virtual system prototyping and high level synthesis [29,43].

While significant prior work has explored generation of optimized simulators including work by Penry et al. [30–32], to our knowledge there has been no previous work on using just-in-time compilation to speed up CL and RTL simulations using dynamically-typed languages. SEJITS proposed just-in-time specialization of high-level algorithm descriptions written in dynamic languages into optimized, platform-specific multicore or CUDA source [10]. JIT techniques have also been previously leveraged to accelerate instruction-set simulators (ISS) [13,24,25,44,50,51]. The GEZEL environment combines a custom interpreted DSL for coprocessor design with existing ISS, supporting both translation into synthesizable VHDL and simulation-time conversion into C++ [39]. Unlike PyMTL, GEZEL is not a general-purpose language and only supports C++ translation of RTL models; PyMTL supports JIT specialization of CL and RTL models.

VI. Conclusion

This paper has presented PyMTL, a unified, vertically-integrated framework for FL, CL, and RTL modeling. Small case studies were used to illustrate how PyMTL can close the computer architecture methodology gap by enabling productive construction of composable FL, CL, and RTL models using concurrent-structural and latency-insensitive design. While these small examples demonstrated some of the power of PyMTL, we believe PyMTL is just a first step towards enabling rapid design space exploration and construction of flexible hardware templates to amortize design effort. Future work plans to explore extending PyMTL with higher-level design abstractions that further increase designer productivity.

In addition, a hybrid approach to just-in-time optimization was proposed to close the performance gap introduced by using Python for hardware modeling. SimJIT, a custom JIT specializer for CL and RTL models, was combined with the PyPy meta-tracing JIT interpreter to bring PyMTL simulation of a mesh network within 4×–6× of optimized C++ code. We hope to further develop SimJIT to support a wider variety of PyMTL constructs and explore more advanced specialization optimizations.

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.

References

[1] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2013.

[2] J. Bachrach et al. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf.*, Jun 2012.

[3] S. Belloeil et al. Stratus: A Procedural Circuit Description Language Based Upon Python. *Int'l Conf. on Microelectronics*, Dec 2007.

[4] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug 2011.

[5] P. Birsinger, R. Xia, and A. Fox. Scalable Bootstrapping for Python. *Int'l Conf. on Information and Knowledge Management*, Oct 2013.

[6] C. F. Bolz et al. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, Jul 2009.

[7] A. Butko et al. Accuracy Evaluation of GEM5 Simulator System. *Int'l Workshop on Reconfigurable Communication-Centric Systems-on-Chip*, Jul 2012.

[8] H. W. Cain et al. Precise and Accurate Processor Simulation. *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Feb 2002.

[9] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Sep 2001.

[10] B. Catanzaro et al. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. *Workshop on Programming Models for Emerging Architectures*, Sep 2009.

[11] J. C. Chaves et al. Octave and Python: High-level Scripting Languages Productivity and Performance Evaluation. *HPCMP Users Group Conference*, Jun 2006.

[12] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. *Int'l Conf. on Compiler Construction*, Mar 2010.

[13] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1994.

[14] J. Decaluwe. MyHDL: A Python-based Hardware Description Language. *Linux Journal*, Nov 2004.

[15] R. Desikan, D. Burger, and S. W. Keckler. Measuring Experimental Error in Microprocessor Simulation. *Int'l Symp. on Computer Architecture*, Jun 2001.

[16] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Dec 2000.

[17] M. Govindan, S. W. Keckler, and D. Burger. End-to-End Validation of Architectural Power Models. *Int'l Symp. on Low-Power Electronics and Design*, Aug 2009.

[18] J. P. Grossman et al. The Role of Cascade, a Cycle-Based Simulation Infrastructure, in Designing the Anton Special-Purpose Supercomputers. *Design Automation Conf.*, Jun 2013.

[19] A. Gutierrez et al. Sources of Error in Full-System Simulation. *Int'l Symp. on Performance Analysis of Systems and Software*, Mar 2014.

[20] P. Haglund et al. Hardware Design with a Scripting Language. *Int'l Conf. on Field Programmable Logic*, Sep 2003.

[21] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Comupting Surveys*, 28(4es), Dec 1996.

[22] E. Logaras and E. S. Manolakos. SysPy: Using Python For Processor-centric SoC Design. *Int'l Conf. on Electronics, Circuits, and Systems*, Dec 2010.

[23] A. Mashtizadeh. PHDL: A Python Hardware Design Framework. M.S. Thesis, EECS Department, MIT, May 2007.

[24] C. May. Mimic: A Fast System/370 Simulator. *Symp. on Interpreters and Interpretive Techniques*, Jun 1987.

[25] W. S. Mong and J. Zhu. DynamoSim: A Trace-based Dynamically Compiled Instruction Set Simulator. *Int'l Conf. on Computer-Aided Design*, Nov 2004.

[26] N. Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design*, Jun 2004.

[27] Numba. Online Webpage, accessed Oct 1, 2014. `http://numba.pydata.org`.

[28] T. E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[29] P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. *Int'l Symp. on Systems Synthesis*, Oct 2001.

[30] D. A. Penry. *The Acceleration of Structural Microarchitectural Simulation Via Scheduling*. Ph.D. Thesis, CS Department, Princeton University, Nov 2006.

[31] D. A. Penry and D. I. August. Optimizations for a Simulator Construction System Supporting Reusable Components. *Design Automation Conf.*, Jun 2003.

[32] D. A. Penry et al. Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2006.

[33] L. Prechelt. An Empirical Comparison of Seven Programming Languages. *IEEE Computer*, 33(10):23–29, Oct 2000.

[34] PyTest. Online Webpage, accessed Oct 1, 2014). `http://www.pytest.org`.

[35] PyTest Coverage Reporting Plugin. Online Webpage, accessed Oct 1, 2014. `https://pypi.python.org/pypi/pytest-cov`.

[36] PyTest Distributed Testing Plugin. Online Webpage, accessed Oct 1, 2014. `https://pypi.python.org/pypi/pytest-xdist`.

[37] Greenlet Concurrent Programming Package. Online Webpage, accessed Oct 1, 2014. `http://greenlet.readthedocs.org`.

[38] A. Rubinsteyn et al. Parakeet: A Just-In-Time Parallel Accelerator for Python. *USENIX Workshop on Hot Topics in Parallelism*, Jun 2012.

[39] P. Schaumont, D. Ching, and I. Verbauwhede. An Interactive Codesign Environment for Domain-Specific Coprocessors. *ACM Trans. on Design Automation of Electronic Systems*, 11(1):70–87, Jan 2006.

[40] O. Shacham et al. Rethinking Digital Design: Why Design Must Change. *IEEE Micro*, 30(6):9–24, Nov/Dec 2010.

[41] O. Shacham et al. Avoiding Game Over: Bringing Design to the Next Level. *Design Automation Conf.*, Jun 2012.

[42] S. Srinath et al. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture*, Dec 2014.

[43] SystemC TLM (Transaction-level Modeling). Online Webpage, accessed Oct 1, 2014. `http://www.accellera.org/downloads/standards/systemc/tlm`.

[44] N. Topham and D. Jones. High Speed CPU Simulation using JIT Binary Translation. *Workshop on Modeling, Benchmarking and Simulation*, Jun 2007.

[45] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse. *Conference on Programming Language Design and Implementation*, Jun 2004.

[46] M. Vachharajani et al. Microarchitectural Exploration with Liberty. *Int'l Symp. on Microarchitecture*, Dec 2002.

[47] M. Vachharajani et al. The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling. *ACM Trans. on Computer Systems*, 24(3):211–249, Aug 2006.

[48] Verilator. Online Webpage, accessed Oct 1, 2014. `http://www.veripool.org/wiki/verilator`.

[49] J. I. Villar et al. Python as a Hardware Description Language: A Case Study. *Southern Conf. on Programmable Logic*, Apr 2011.

[50] H. Wagstaff et al. Early Partial Evaluation in a JIT-compiled, Retargetable Instruction Set Simulator Generated from a High-level Architecture Description. *Design Automation Conf.*, Jun 2013.

[51] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1996.

[52] M. Zhang et al. Trilobite: A Natural Modeling Framework for Processor Design Automation System. *Int'l Conf. on ASIC*, Oct 2009.

[53] M. Zhang, S. Tu, and Z. Chai. PDSDL: A Dynamic System Description Language. *Int'l SoC Design Conf.*, Nov 2008.