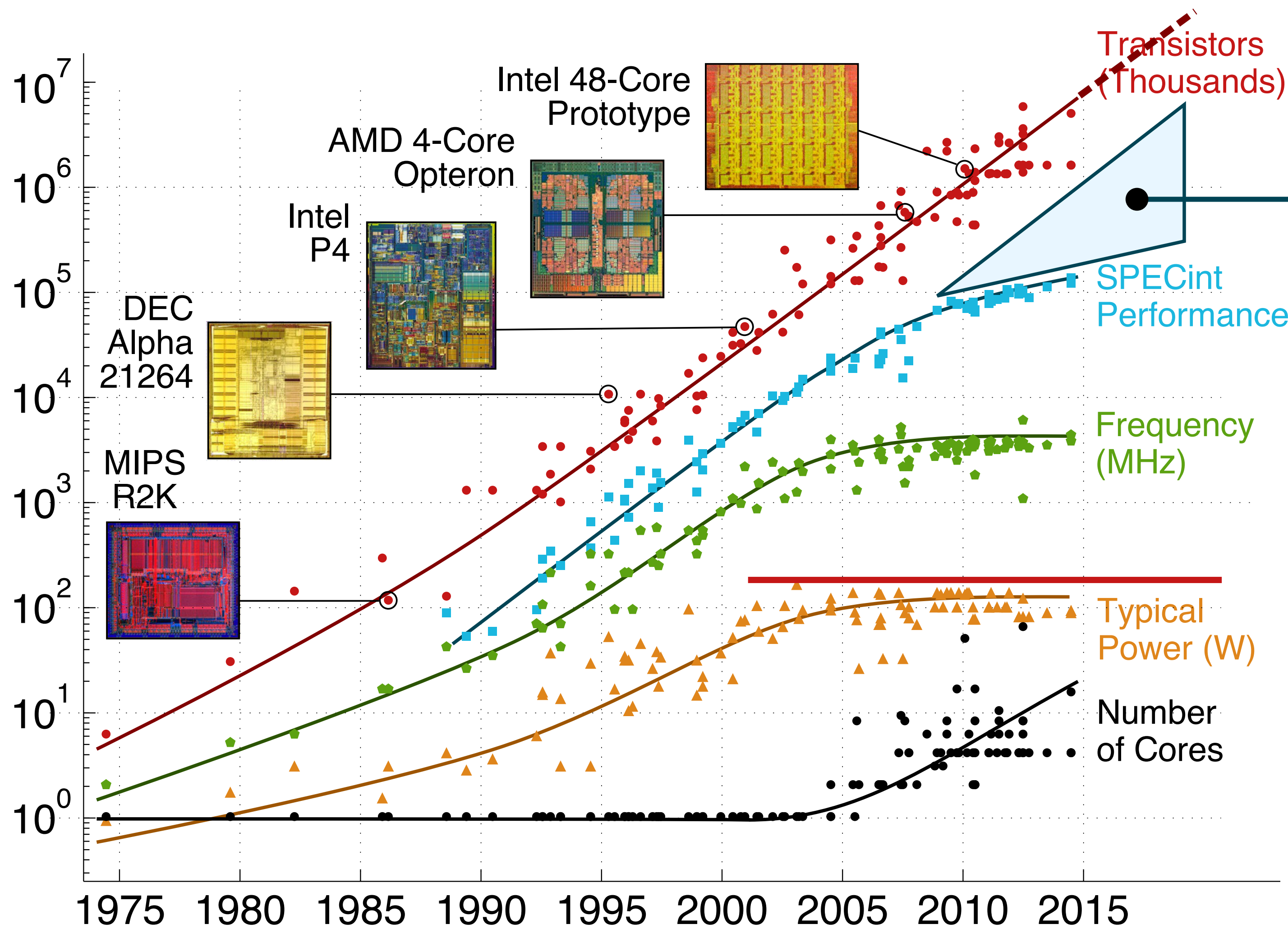
PyMTL and Pydgin: Python Frameworks for Highly Productive Computer Architecture Research

Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

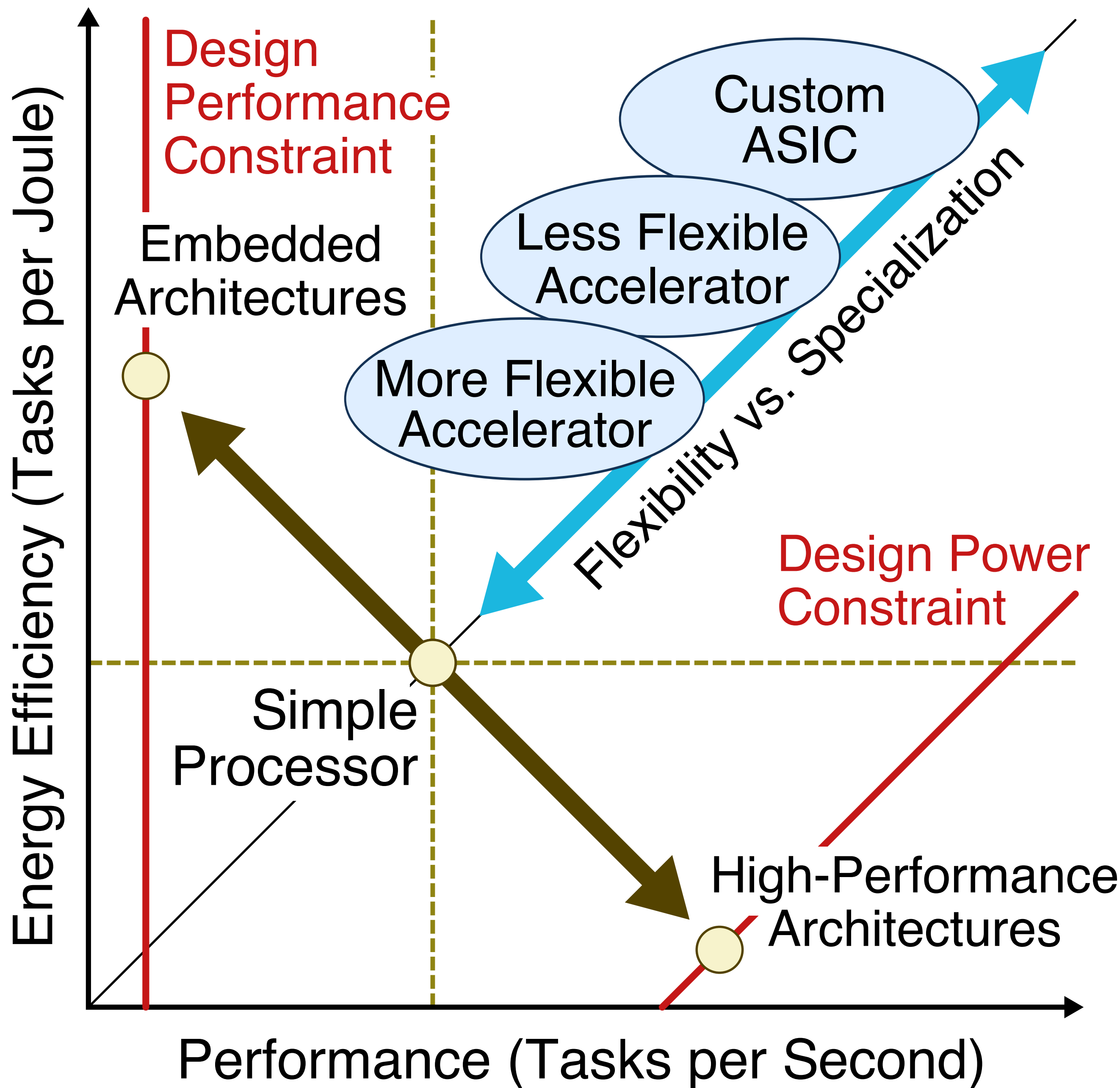
Spring 2016

Motivating Trends in Computer Architecture



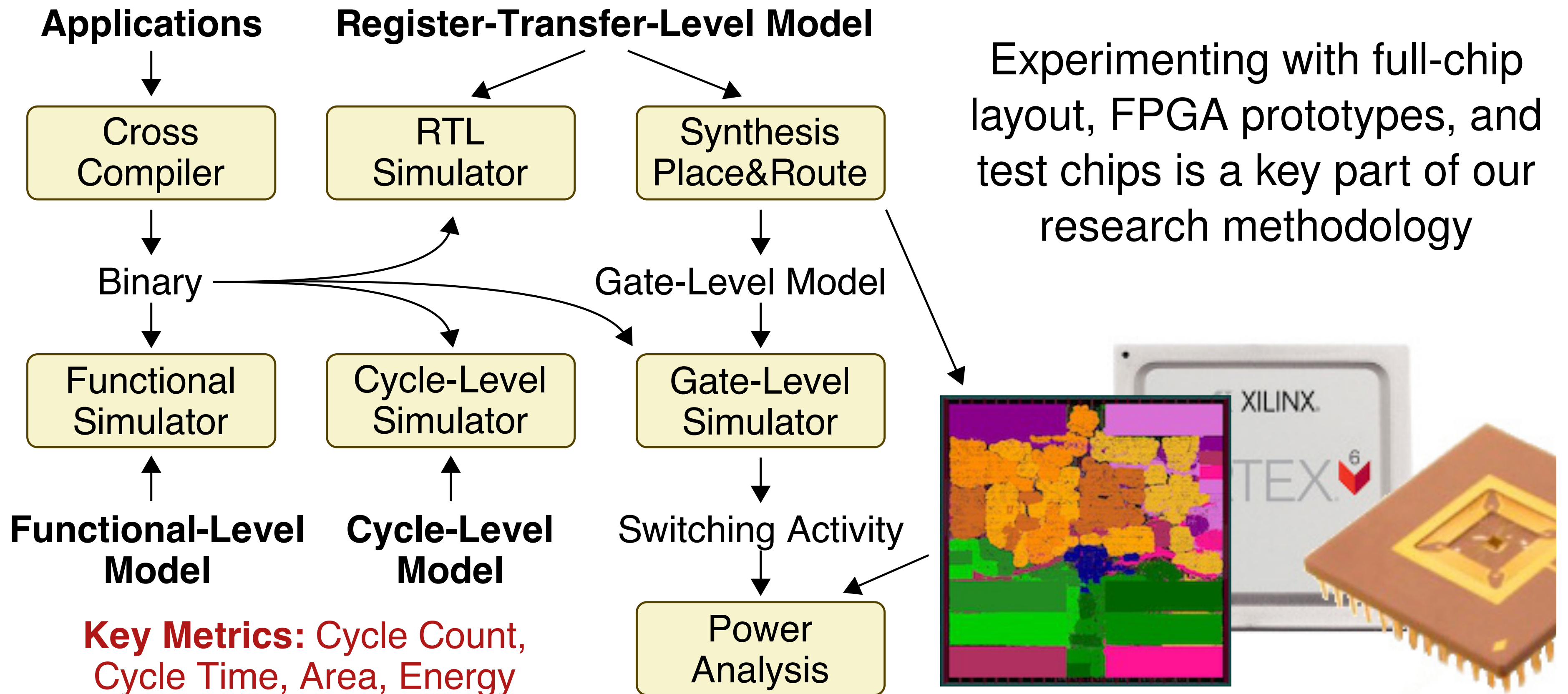
Hardware Specialization

- Data-Parallelism via GPGPUs and Vector
- Fine-Grain Task-Level Parallelism
- Instruction Set Specialization
- Subgraph Specialization
- Application-Specific Accelerators
- Domain-Specific Accelerators
- Coarse-Grain Reconfig Arrays
- Field-Programmable Gate Arrays

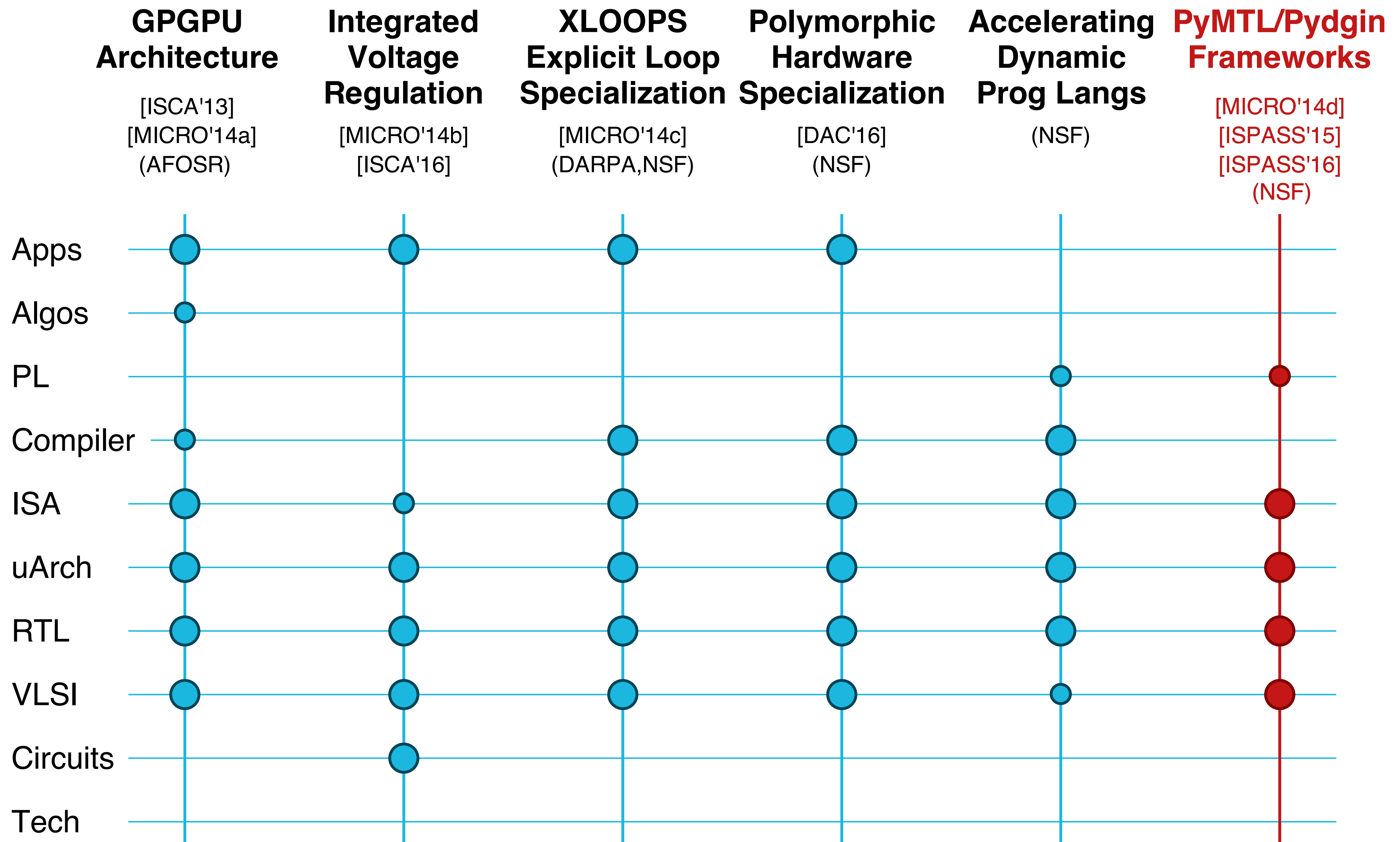

Design
Performance
Constraint
Embedded
Architectures
Custom
ASIC
Less Flexible
Accelerator
More Flexible
Accelerator
Flexibility vs. Specialization
Design Power
Constraint
Simple
Processor
High-Performance
Architectures
Energy Efficiency (Tasks per Joule)
Performance (Tasks per Second)

Vertically Integrated Research Methodology

Our research involves reconsidering all aspects of the computing stack including applications, programming frameworks, compiler optimizations, runtime systems, instruction set design, microarchitecture design, VLSI implementation, and hardware design methodologies



Projects Within the Batten Research Group



PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, Christopher Batten

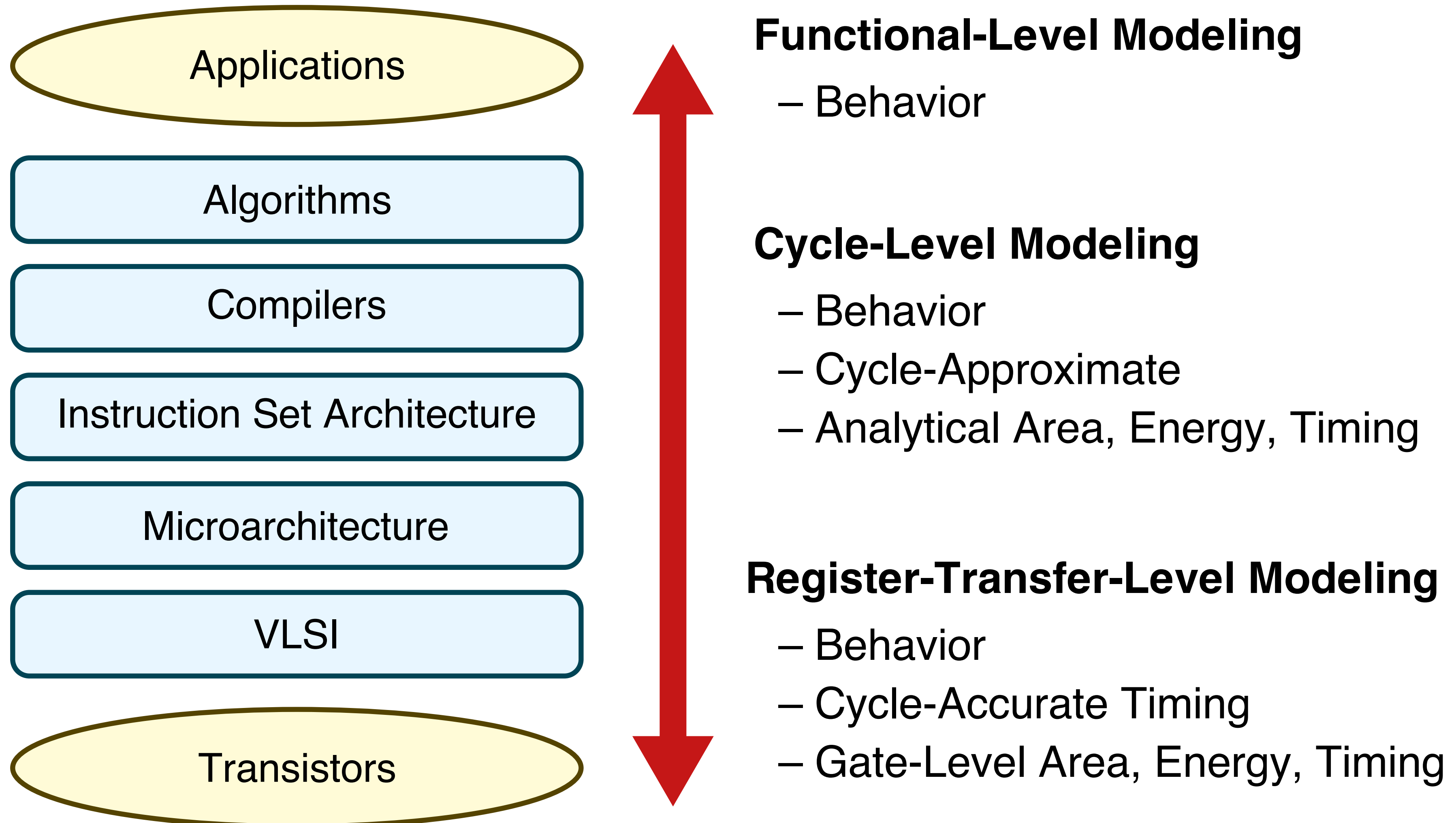
47th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)
Cambridge, UK, Dec. 2014

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

Derek Lockhart, Berkin Ilbeyi, Christopher Batten

IEEE Int'l Symp. on Perf Analysis of Systems and Software (ISPASS)
Philadelphia, PA, Mar. 2015

Computer Architecture Research Methodologies



Computer Architecture Research Methodologies

Computer Architecture Research Methodology Gap

FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies

Our Approch: Modeling Towards Layout

Unified Python-based framework for FL, CL, and RTL modeling

Functional-Level Modeling

- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

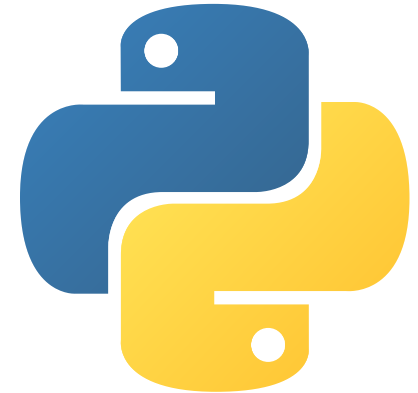
Cycle-Level Modeling

- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

Register-Transfer-Level Modeling

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow

Why Python?



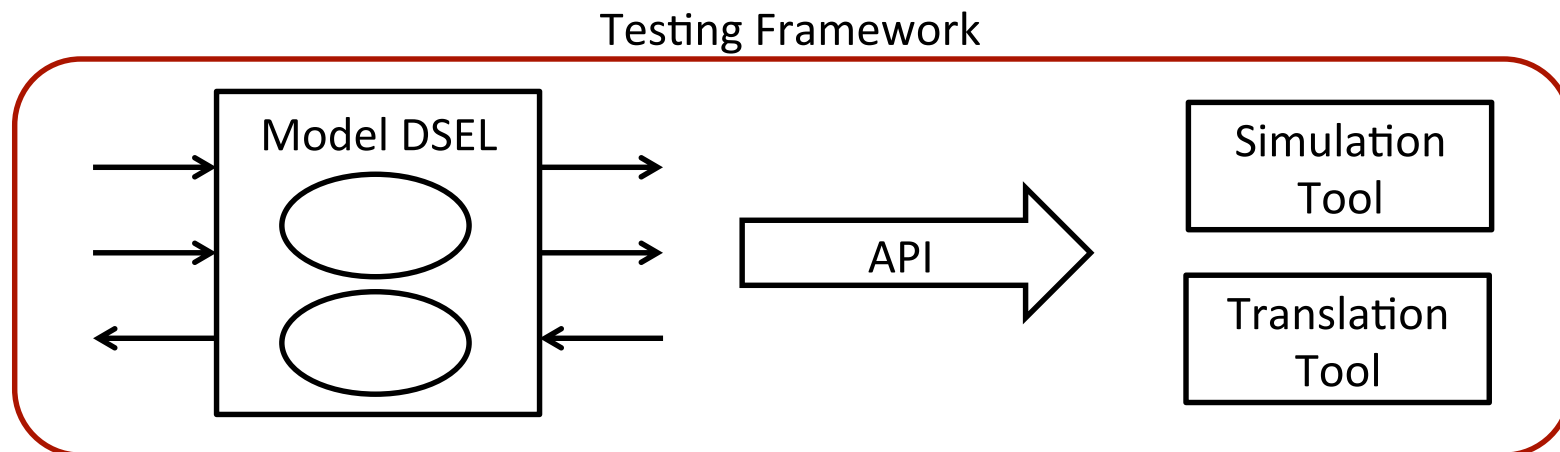
- Python is well regarded as a highly productive language with lightweight, pseudocode-like syntax
- Python supports modern language features to enable rapid, agile development (dynamic typing, reflection, metaprogramming)
- Python has a large and active developer and support community
- Python includes extensive standard and third-party libraries
- Python enables embedded domain-specific languages
- Python facilitates engaging application-level researchers
- Python includes built-in support for integrating with C/C++
- Python performance is improving with advanced JIT compilation

Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC) — Consistent interfaces across abstractions
- **Unified Modeling Languages** (SystemC) — Unified design environment for FL, CL, RTL
- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL) — Productive RTL design space exploration
- **HDL-Integrated Simulation Frameworks** (Cascade) — Productive RTL validation and cosimulation
- **Latency-Insensitive Interfaces** (Liberty, BlueSpec) — Component and test bench reuse

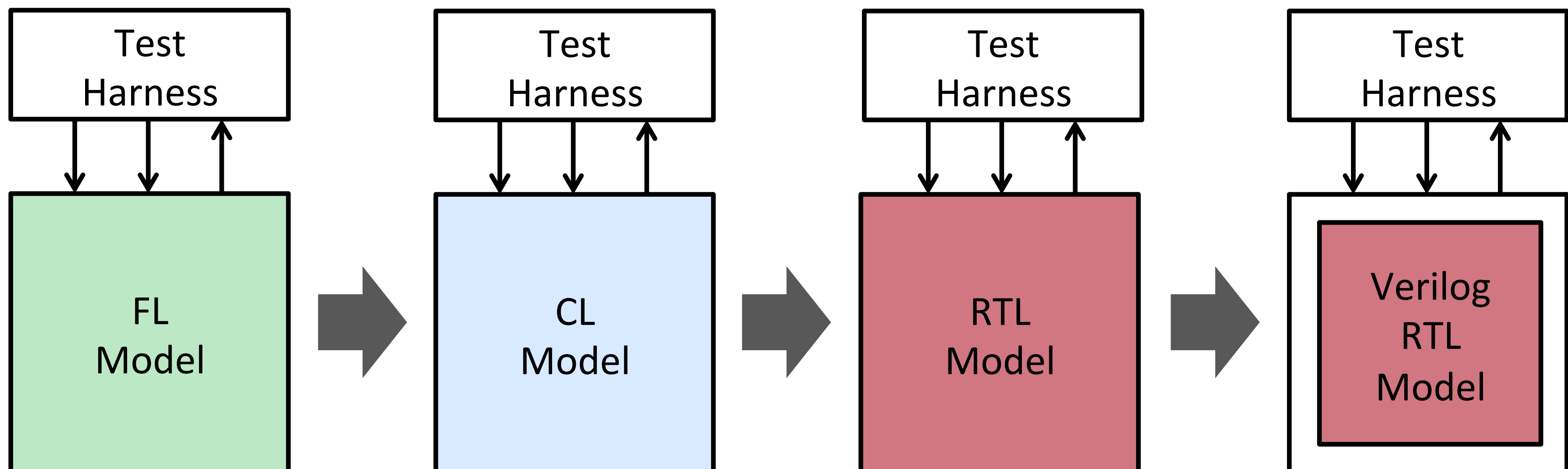
What is PyMTL?

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



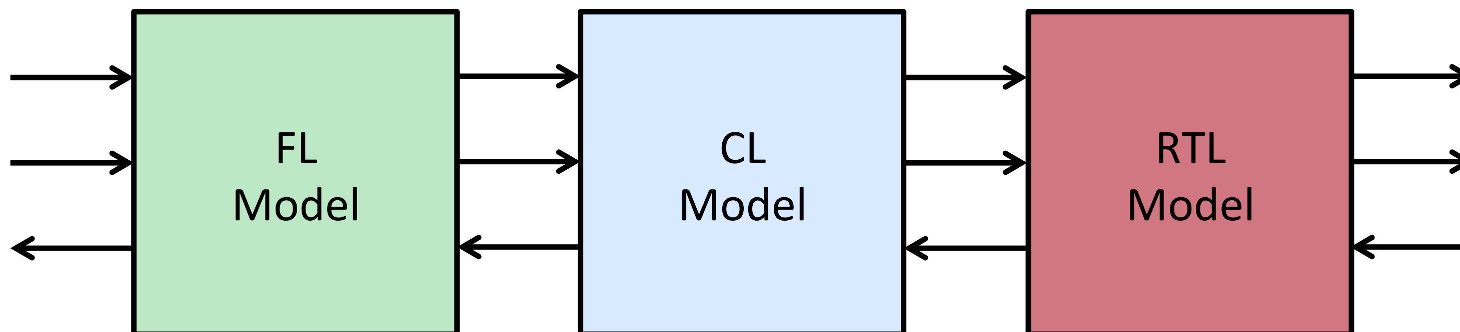
What Does PyMTL Enable?

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



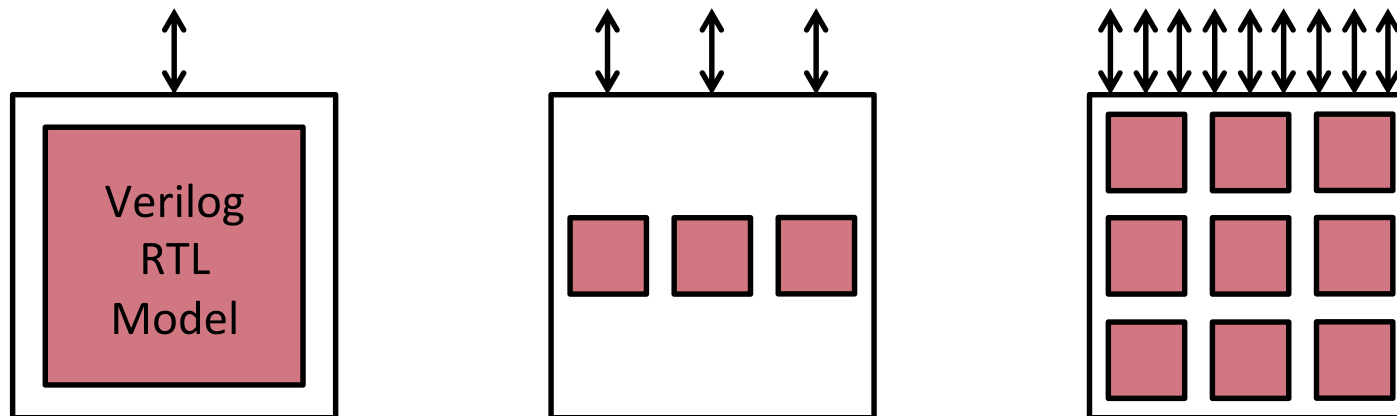
What Does PyMTL Enable?

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



What Does PyMTL Enable?

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



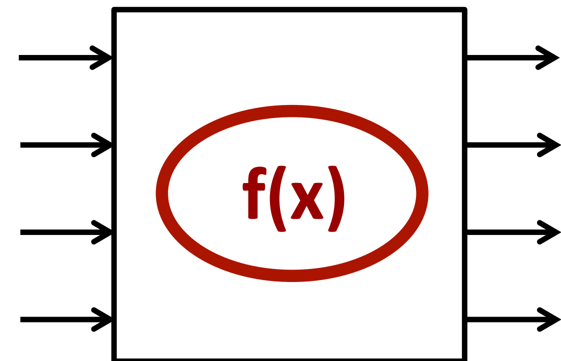
The PyMTL Framework

The PyMTL DSEL: FL Models

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

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

```
class SorterNetworkFL( Model )
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)


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

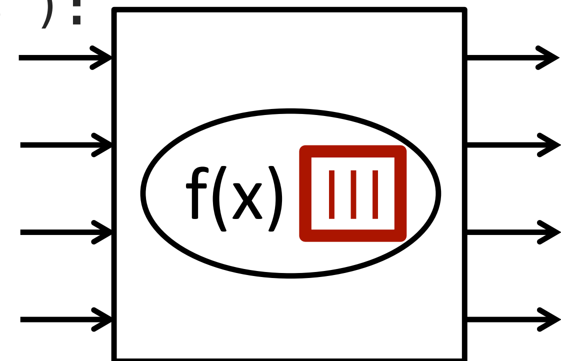


The PyMTL DSEL: CL Models

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

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

```
class SorterNetworkCL( Model )
  def __init__( s, nbits, nports, delay=3 ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)
    s.pipe = Pipeline( delay )

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

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

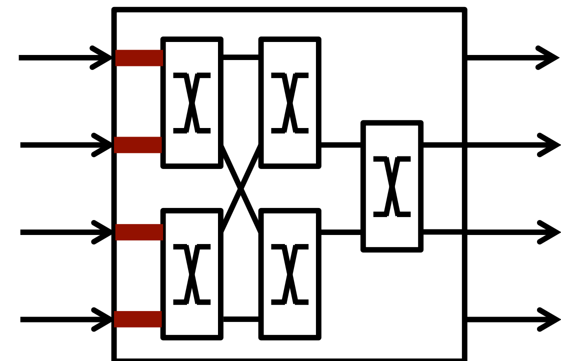


The PyMTL DSEL: RTL Models

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

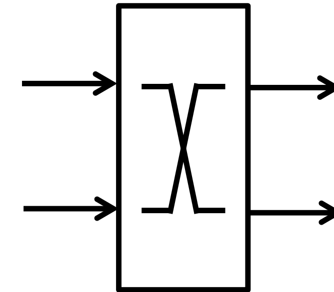
[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkRTL( Model )
  def __init__( s, nbits ):

    s.in_  = InPort [4](nbits)
    s.out  = OutPort[4](nbits)

    s.m = m = MinMaxRTL[5](nbits)

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

    . . .
```

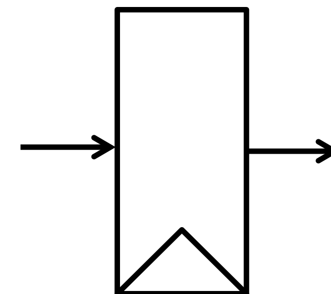


The PyMTL DSEL: RTL Models

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



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

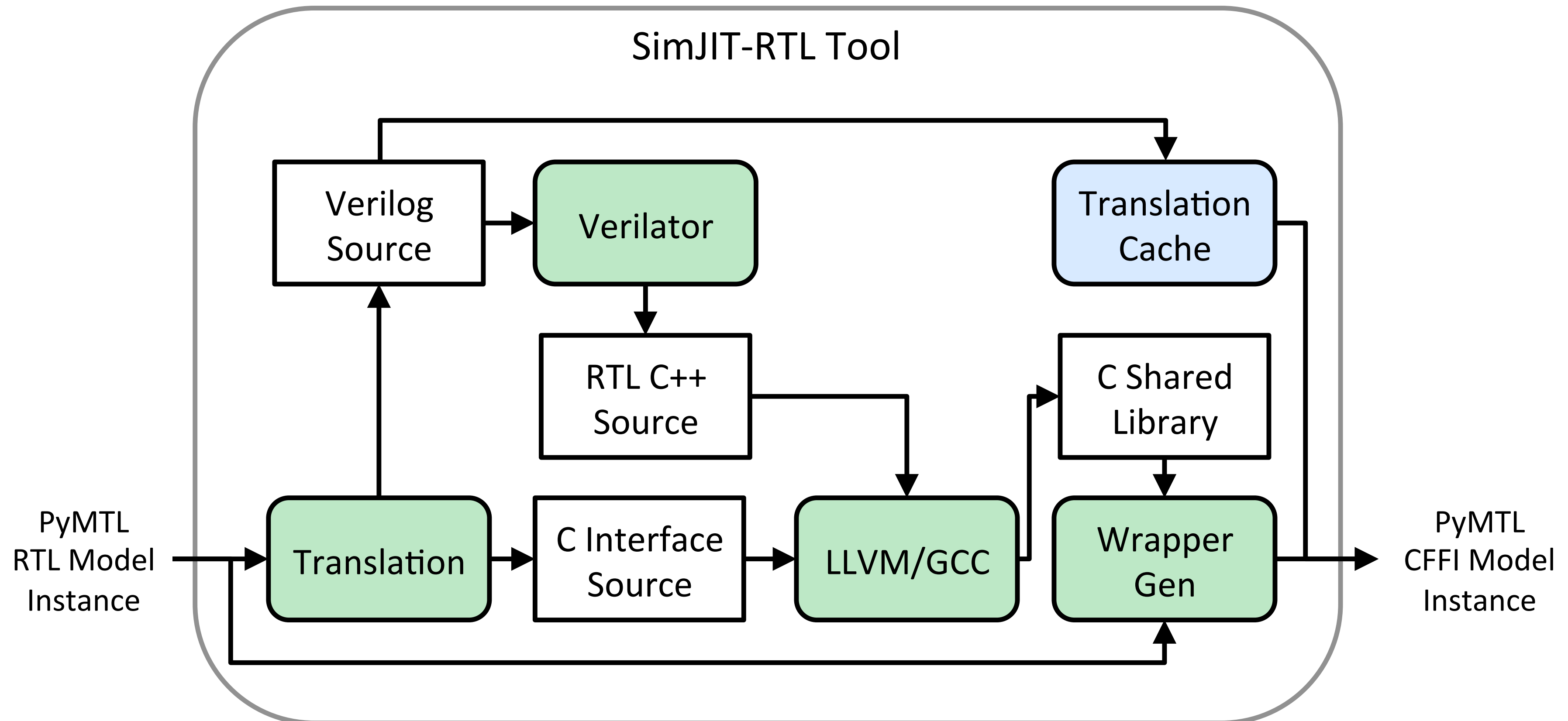


Performance/Productivity Gap

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

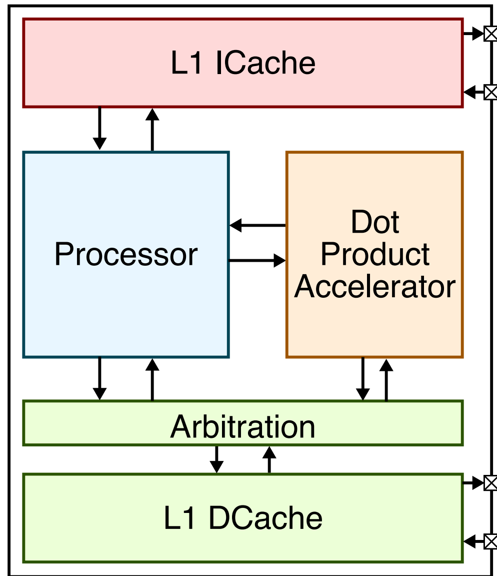
- Python-Wrapped C/C++ Libraries
  (NumPy, CVXOPT, NLPy, pythonoCC, gem5)
- Numerical Just-In-Time Compilers
  (Numba, Parakeet)
- Just-In-Time Compiled Interpreters
  (PyPy, Pyston)
- Selective Embedded Just-In-Time Specialization
  (SEJITS)

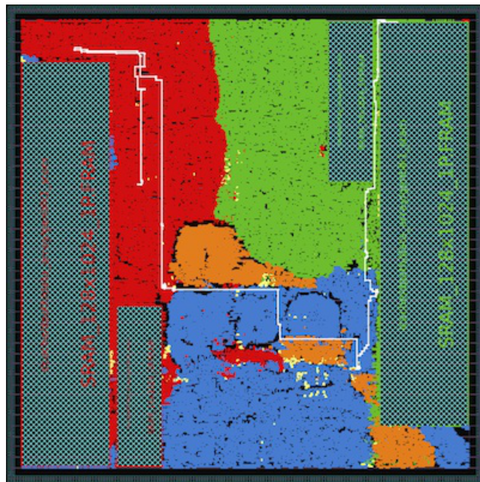
PyMTL SimJIT-RTL Architecture



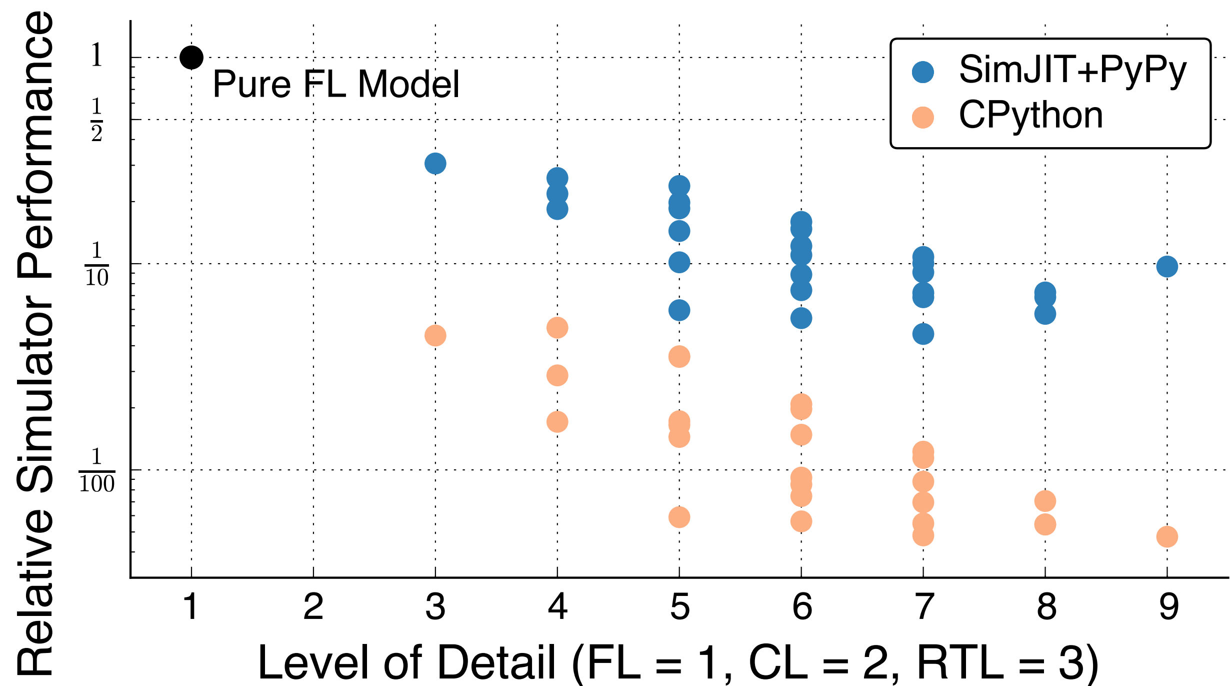
- **SimJIT-RTL:** Robust and actively used in research/teaching
- **SimJIT-CL:** Proof-of-concept implementation

PyMTL Accelerator Case Study

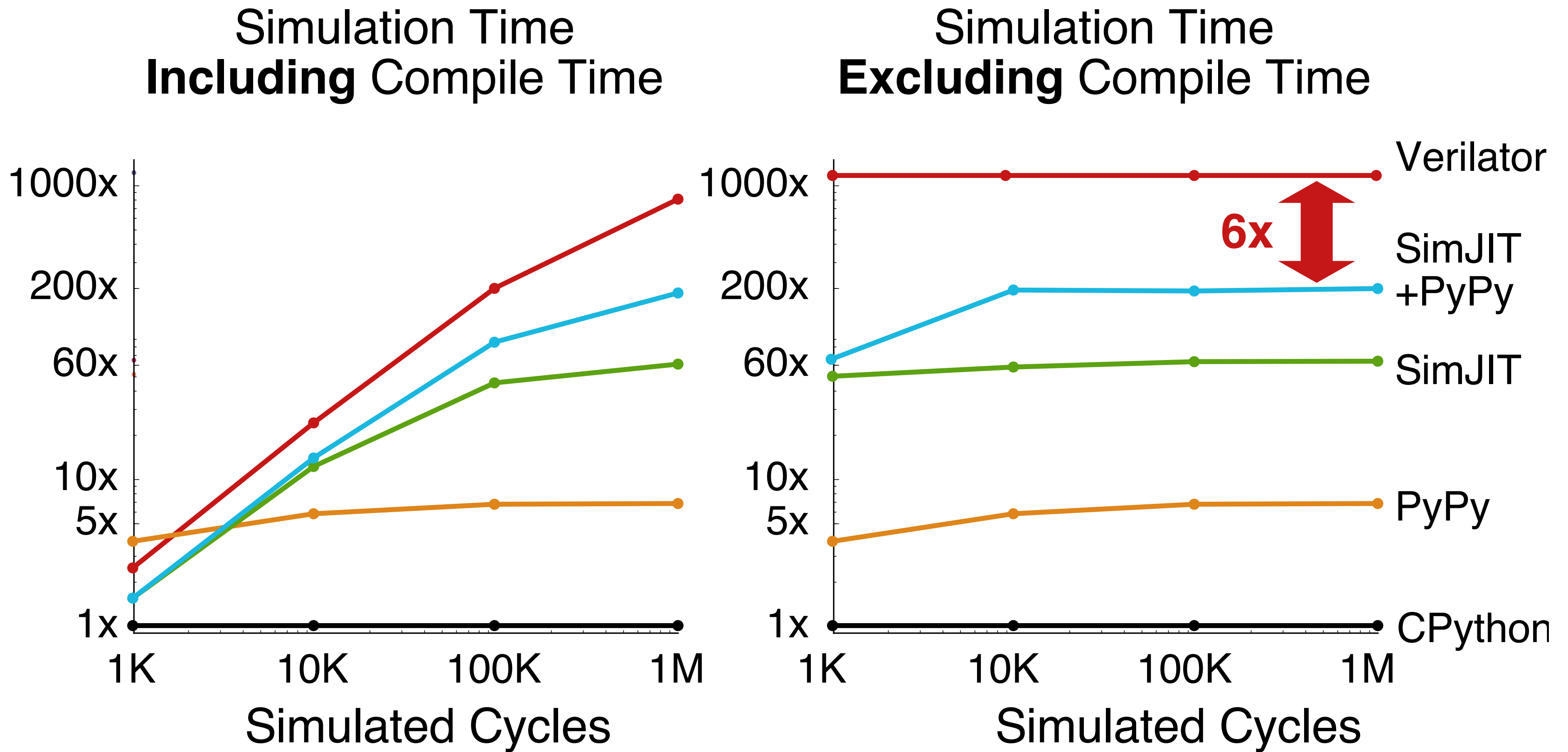




- Experimented with FL, CL, and RTL models of a pipelined processor, blocking cache, and dot-product accelerator
- 27 different compositions that trade-off simulator performance vs. accuracy



PyMTL Results: 64-Node Mesh Network



RTL model of 64-node mesh network with single-cycle routers, elastic buffer flow control, uniform random traffic, with an injection rate just before saturation

PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, Christopher Batten

47th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO) Cambridge, UK, Dec. 2014

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

Derek Lockhart, Berkin Ilbeyi, Christopher Batten

IEEE Int'l Symp. on Perf Analysis of Systems and Software (ISPASS) Philadelphia, PA, Mar. 2015

Computer Architecture Research Methodologies

While it is certainly possible to create stand-alone instruction set simulators in PyMTL, their performance is quite slow (~100 KIPS)

Can we achieve high-performance while maintaining productivity for instruction set simulators?

Functional-Level Modeling

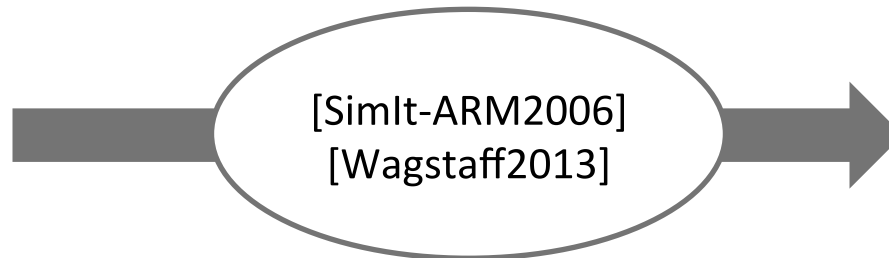
- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

Cycle-Level Modeling

- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

Register-Transfer-Level Modeling

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow



[Simit-ARM2006]

+ Page-based JIT
- Ad-hoc ADL with custom parser
- Unmaintained

[Wagstaff2013]

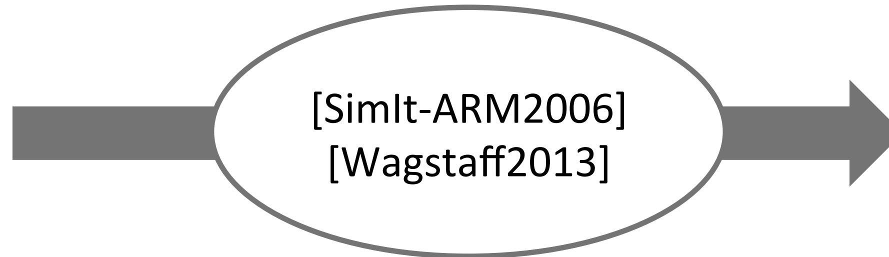
+ Region-based JIT
+ Industry-supported ADL (ArchC)
- C++-based ADL is verbose
- Not Public

[Simit-ARM2006] J.D'Errico and W.Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. DATE'06.
[Wagstaff2013] H. Wagstaff, M. Gould, B. Franke, and N.Topham. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. DAC'13.

Productivity

Performance

Architectural Description Language



Instruction Set Interpreter in C with DBT

Key Insight:

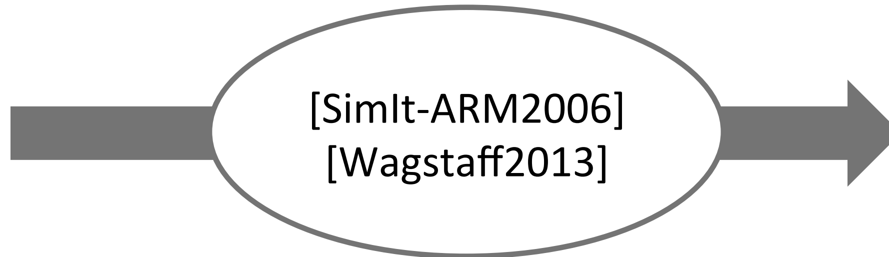
Similar productivity-performance challenges for building high-performance interpreters of dynamic languages.
(e.g. JavaScript, Python)

Dynamic Language Interpreter in C with JIT Compiler

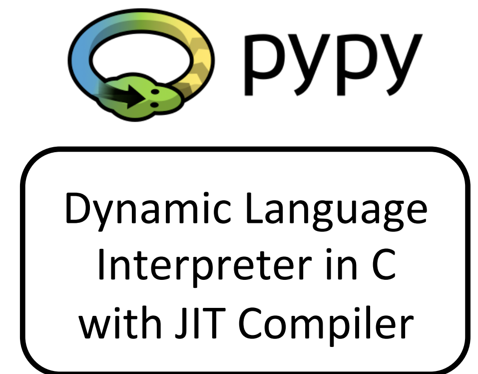
Productivity

Performance

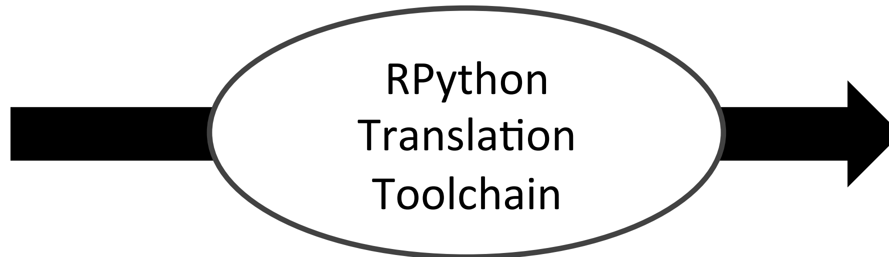
Architectural Description Language



Instruction Set Interpreter in C with DBT



Dynamic-Language Interpreter in RPython



Dynamic Language Interpreter in C with JIT Compiler

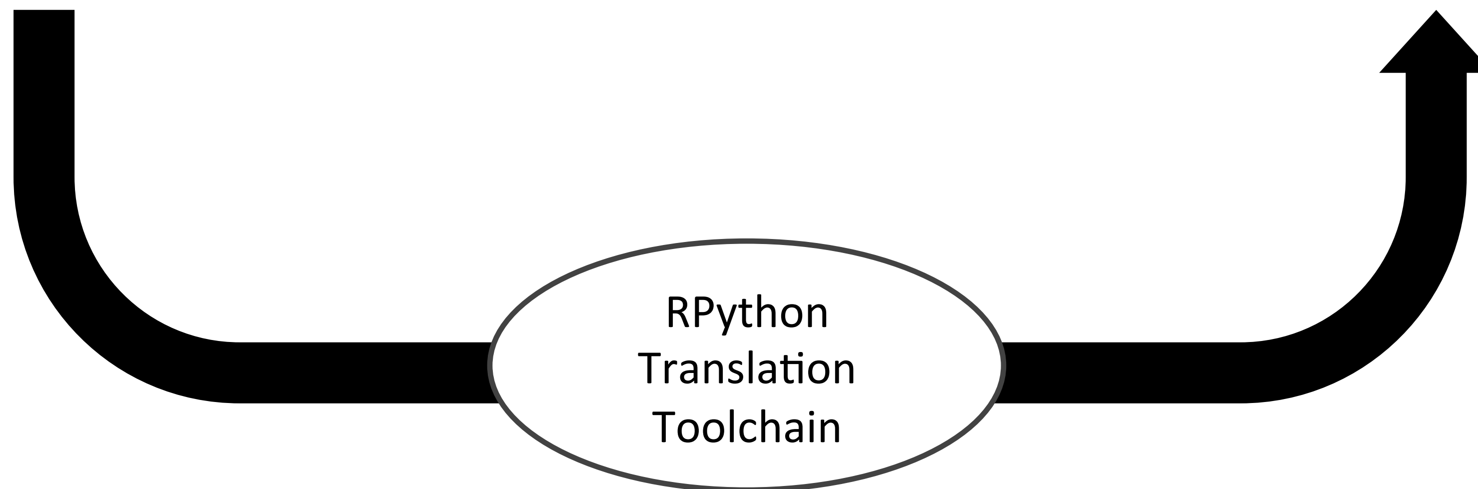
Meta-Tracing JIT:
makes JIT generation generic across languages

Productivity

Performance

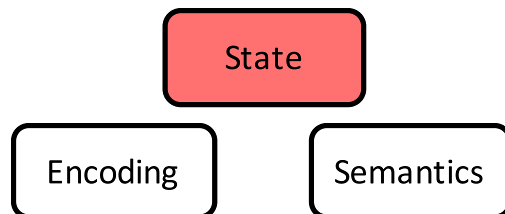
Architectural Description Language

Instruction Set Interpreter in C with DBT



Just-in-Time Compilation $\approx$ Dynamic Binary Translation

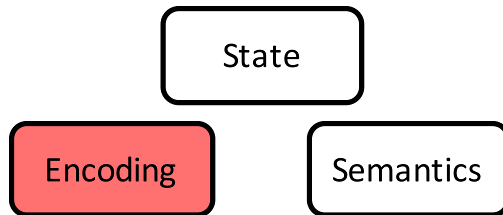
Pydgin Architecture Description Language



Architectural State

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):

    self.pc  = reset_addr
    self.rf  = RiscVRegisterFile()
    self.mem = memory

    # optional state if floating point is enabled
    if ENABLE_FP:
      self.fp   = RiscVFPRegisterFile()
      self.fcsr = 0
```

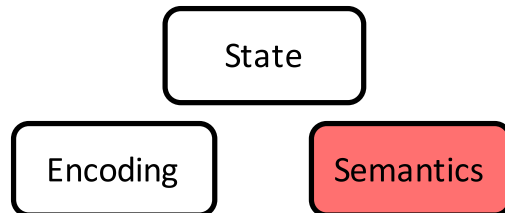
Pydgin Architecture Description Language



Instruction Encoding

```
encodings = [
  # ...

  ['xori', 'xxxxxxxxxxxxxxxxx100xxxxx0010011'],
  ['ori',  'xxxxxxxxxxxxxxxxx110xxxxx0010011'],
  ['andi', 'xxxxxxxxxxxxxxxxx111xxxxx0010011'],
  ['slli', '000000xxxxxxxxxxx001xxxxx0010011'],
  ['srli', '000000xxxxxxxxxxx101xxxxx0010011'],
  ['srai', '010000xxxxxxxxxxx101xxxxx0010011'],
  ['add',  '0000000xxxxxxxxxx000xxxxx0110011'],
  ['sub',  '0100000xxxxxxxxxx000xxxxx0110011'],
  ['sll',  '0000000xxxxxxxxxx001xxxxx0110011'],

  # ...
]
```

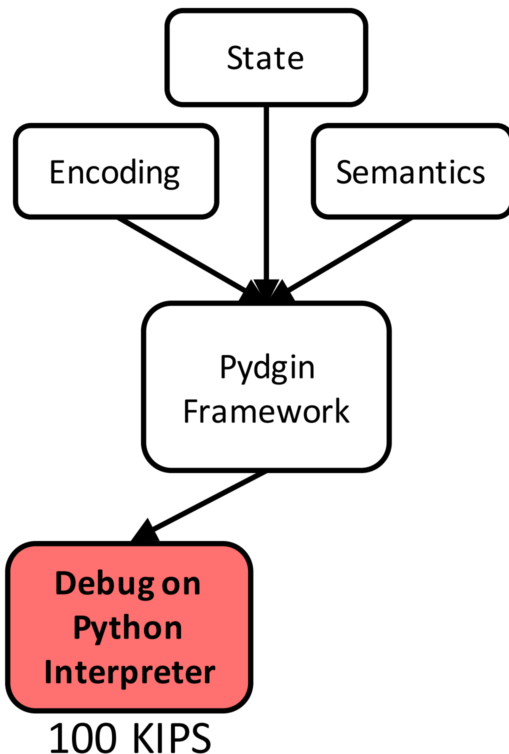
Pydgin Architecture Description Language



Instruction Semantics

```
def execute_addi( s, inst ):
  s.rf[inst.rd] = s.rf[inst.rs1] + inst.i_imm
  s.pc += 4

def execute_sw( s, inst ):
  addr = trim_xlen( s.rf[inst.rs1] + inst.s_imm )
  s.mem.write( addr, 4, trim_32( s.rf[inst.rs2] ) )
  s.pc += 4

def execute_beq( s, inst ):
  if s.rf[inst.rs1] == s.rf[inst.rs2]:
    s.pc = trim_xlen( s.pc + inst.sb_imm )
  else:
    s.pc += 4
```

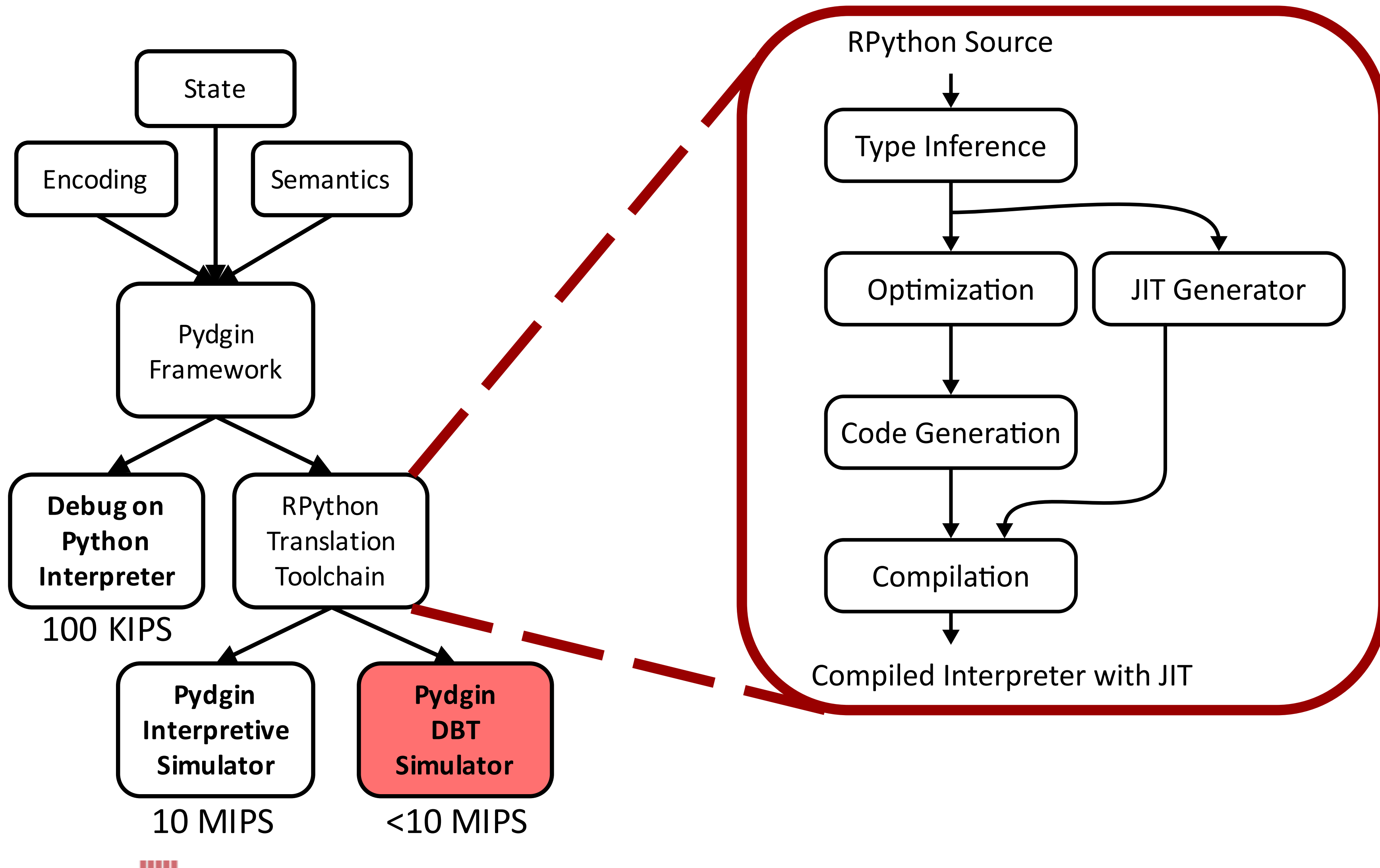
Pydgin Framework



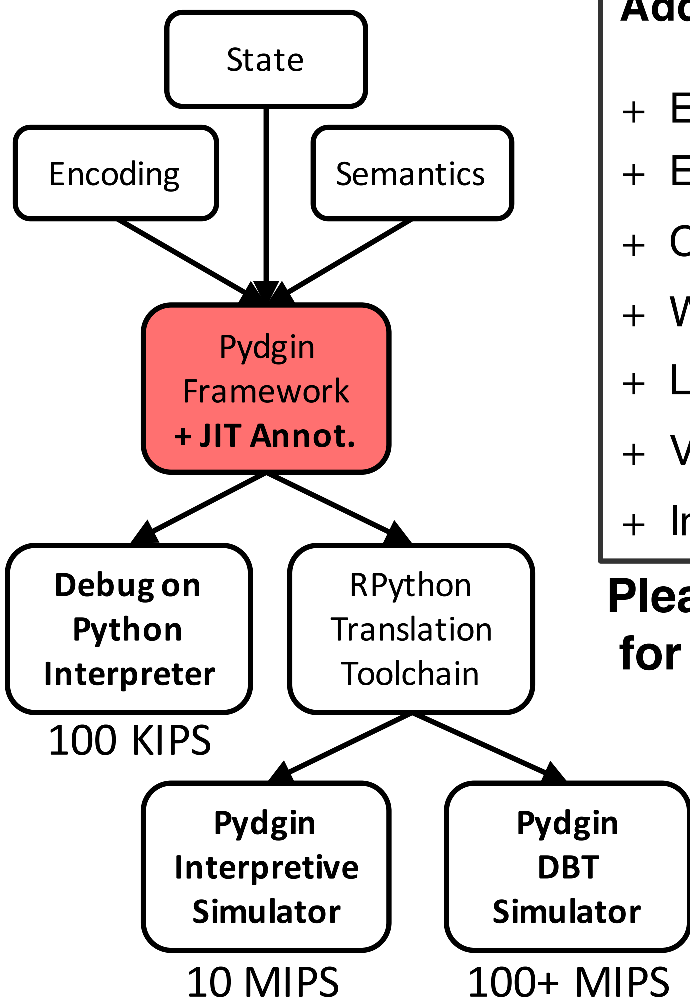
Interpreter Loop

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]       # fetch
    execute = decode( inst )     # decode
    execute( state, inst )       # execute
```

The RPython Translation Toolchain



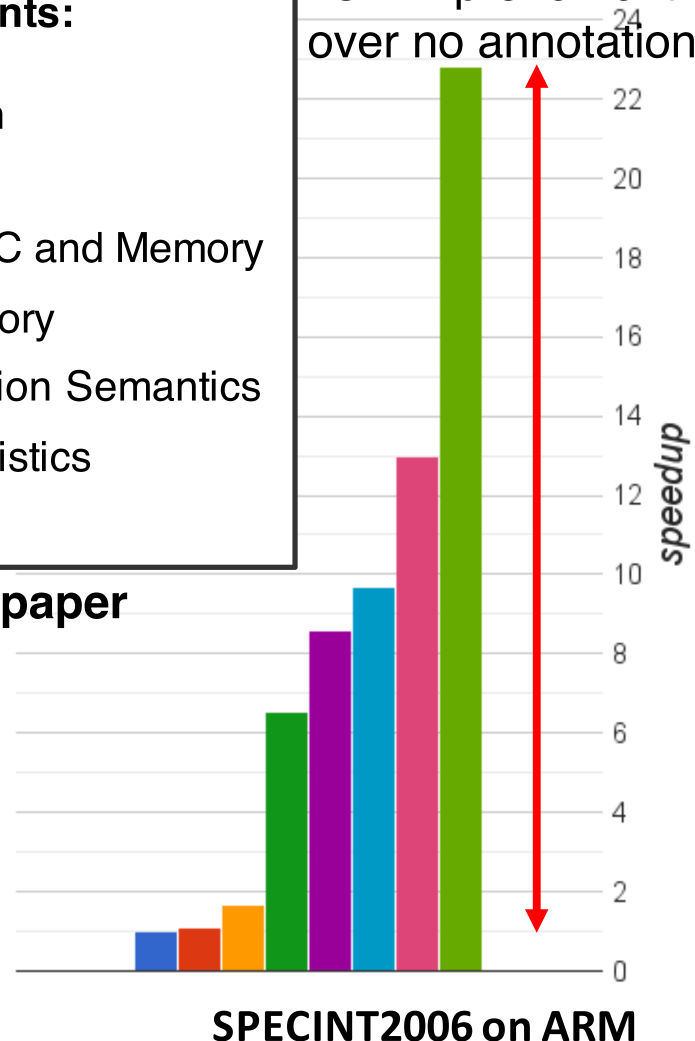
JIT Annotations and Optimizations



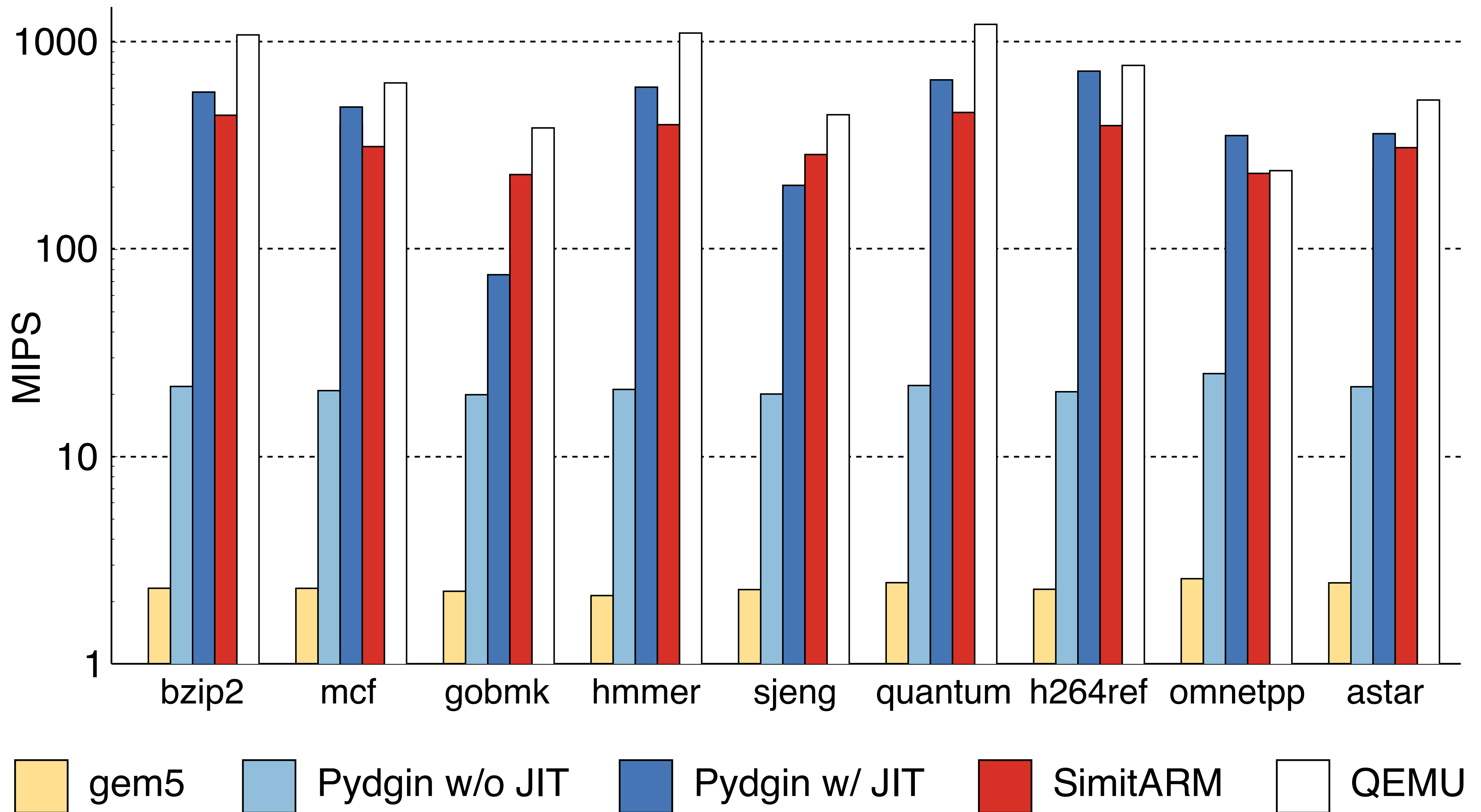
Additional RPython JIT hints:

- + Elidable Instruction Fetch
- + Elidable Decode
- + Constant Promotion of PC and Memory
- + Word-Based Target Memory
- + Loop Unrolling in Instruction Semantics
- + Virtualizable PC and Statistics
- + Increased Trace Limit

Please see our ISPASS paper for more details!



Pydgin Results: ARMv5 Instruction Set

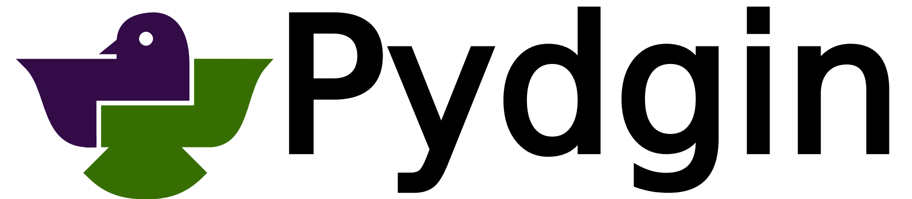


Porting Pydgin to a new user-level ISA takes just a few weeks

PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, Christopher Batten

47th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)
Cambridge, UK, Dec. 2014



Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

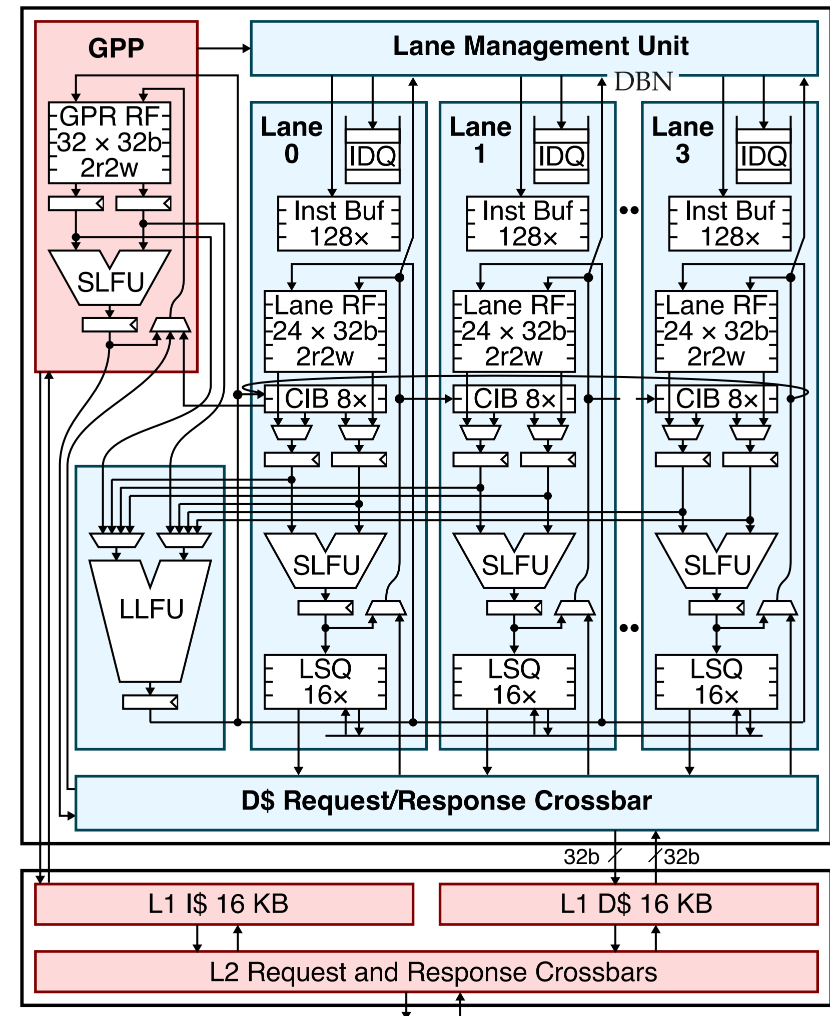
Derek Lockhart, Berkin Ilbeyi, Christopher Batten

IEEE Int'l Symp. on Perf Analysis of Systems and Software (ISPASS)
Philadelphia, PA, Mar. 2015

PyMTL In Practice: Explicit Loop Specialization

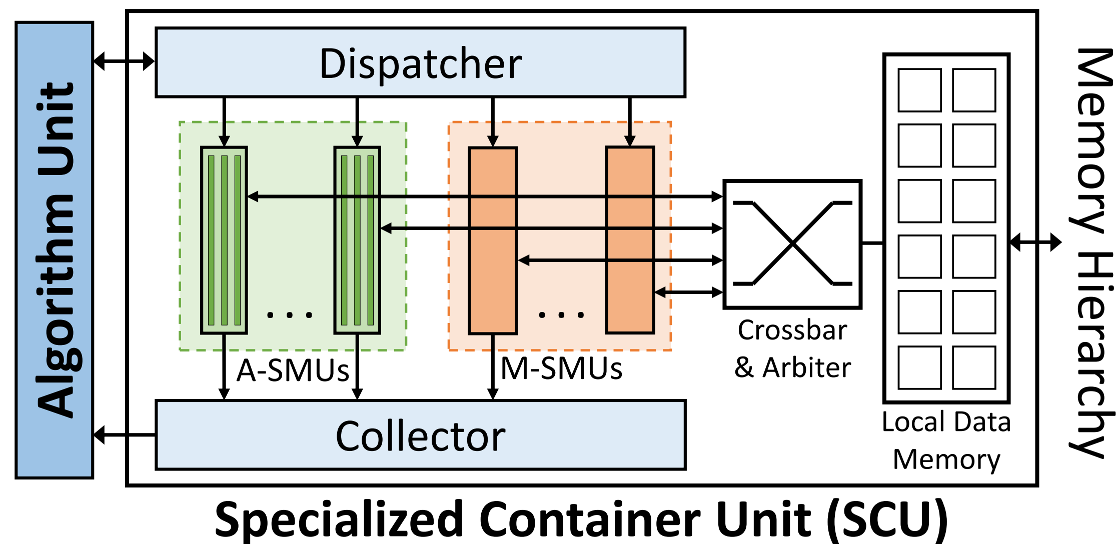
- Our MICRO'14 paper explored a new loop accelerator architecture
- gem5 provided access to complex out-of-order processor and memory system models (red).
- PyMTL was used to quickly build and iterate on a CL model for the loop acceleration unit (blue).

S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, C. Batten, "Architectural Specialization for Inter-Iteration Loop Dependence Patterns." 47th ACM/IEEE Int'l Symp. on Microarchitecture, Dec. 2014.



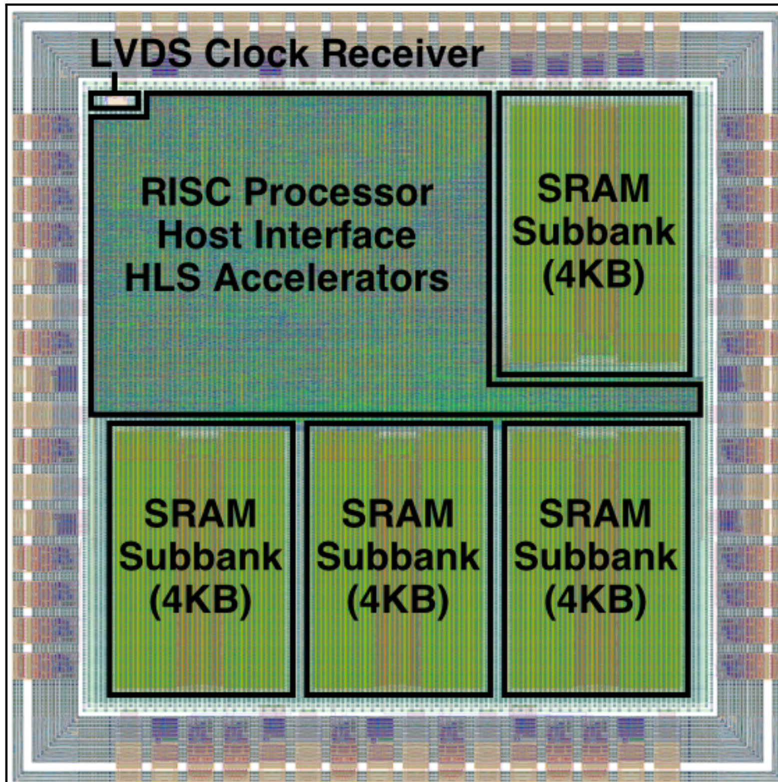
PyMTL In Practice: HLS for Data Structures

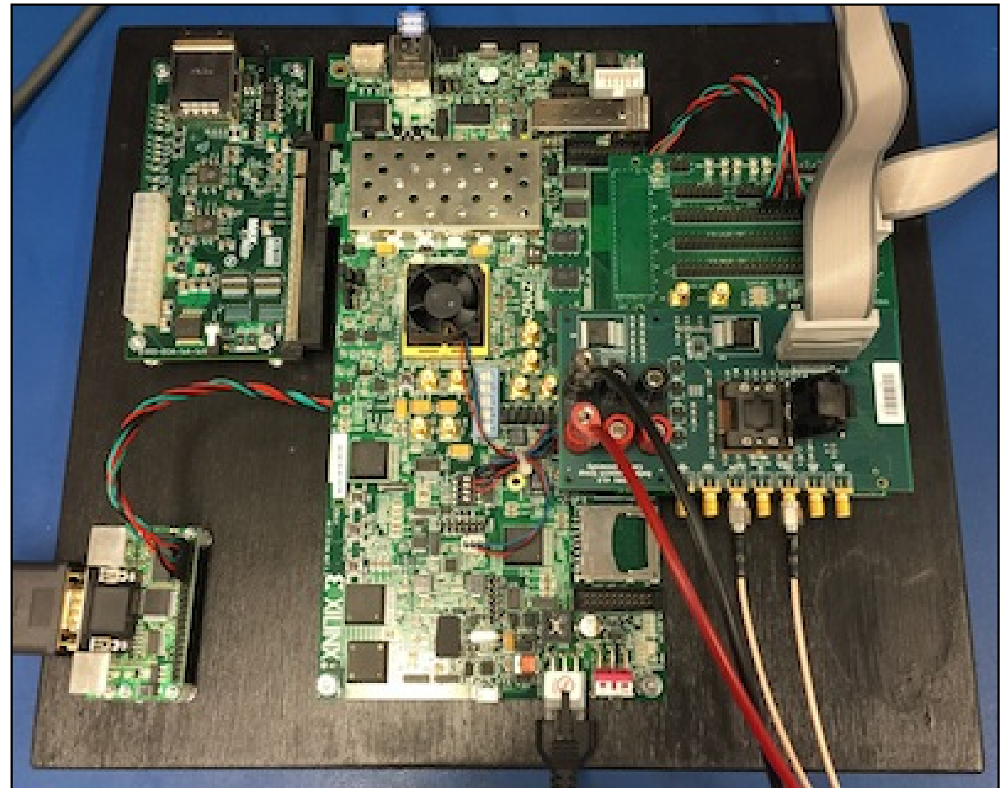
- Our DAC'16 paper explored new high-level synthesis (HLS) techniques for decoupling algorithm units from data-structure units
- PyMTL was used to create highly parameterized RTL templates for composing the various modules within the data-structure unit



R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang. "Improving High-Level Synthesis with Decoupled Data Structure Optimization." 53rd ACM/IEEE Design Automation Conference, June 2016.

PyMTL In Practice: ASIC Test Chip





Tapeout-ready layout for RISC processor, 16KB SRAM, and HLS-generated accelerators

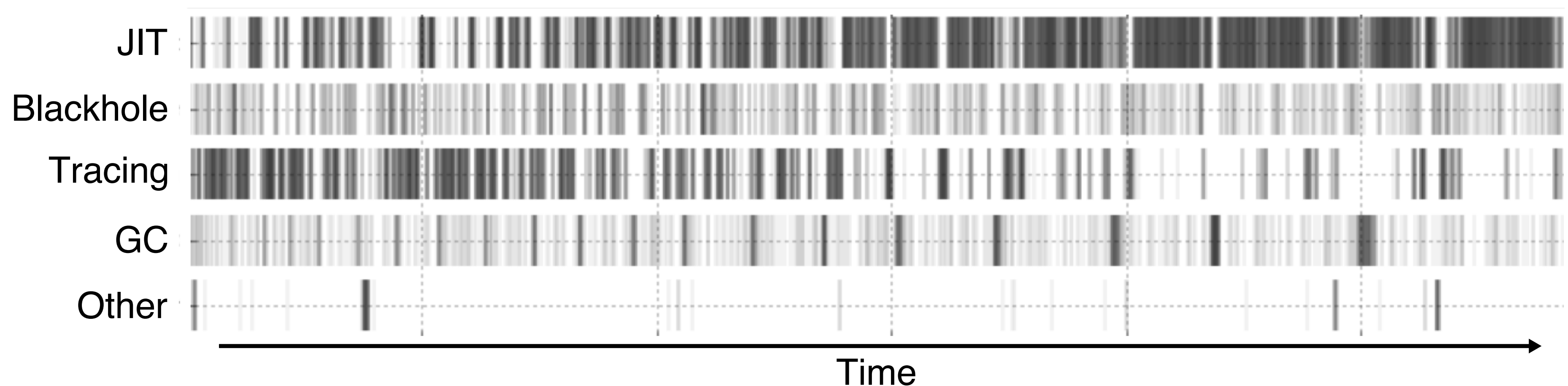
2x2mm 1.3M-trans in IBM 130nm

Xilinx ZC706 FPGA development board for FPGA prototyping

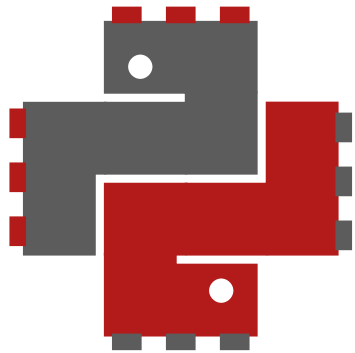
Custom designed FMC mezzanine card for ASIC test chips

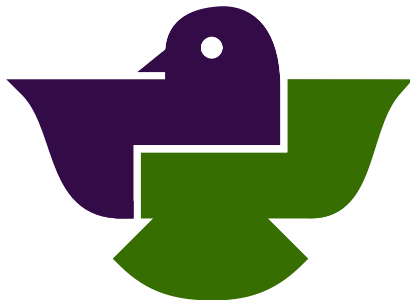
Pydgin In Practice: Hardware Specialization for Dynamic Programming Languages

- We are using Pydgin to run large Python programs to study the time spent in various parts of the Python JIT compilation framework
- Pydgin enables instrumentation of application phase behavior and collection of per-phase architectural statistics



PyMTL/Pydgin Take-Away Points

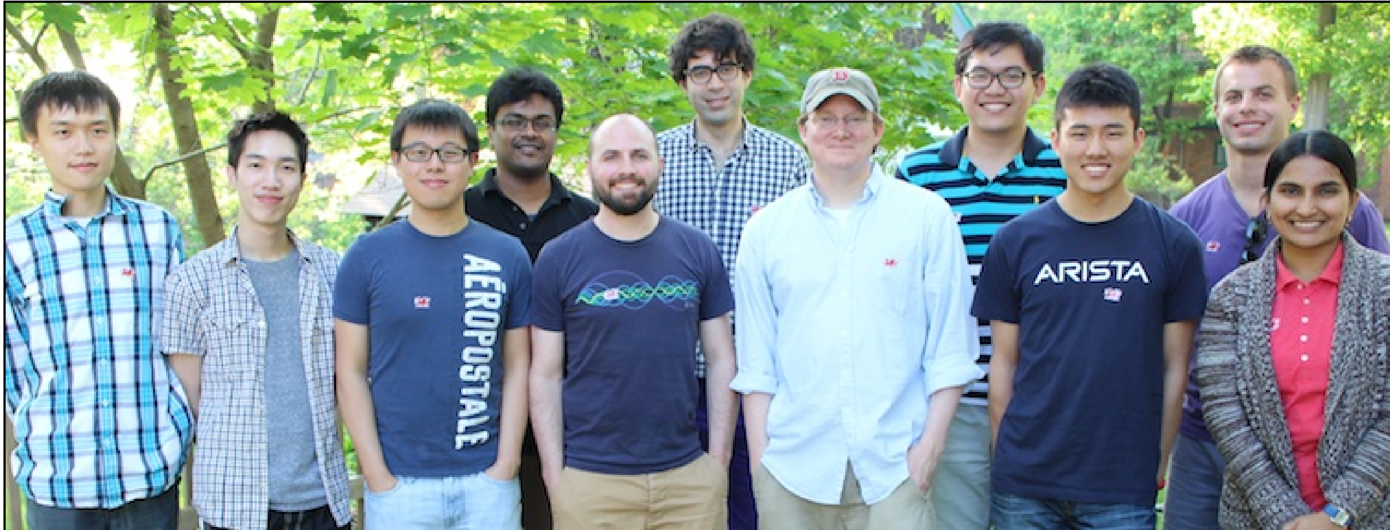




- PyMTL is a productive Python framework for FL, CL, and RTL modeling and hardware design
- Pydgin is a framework for rapidly developing very fast instruction-set simulators from a Python-based architecture description language
- PyMTL and Pydgin leverage novel application of JIT compilation to help close the performance/productivity gap
- Alpha versions of PyMTL and Pydgin are available for researchers to experiment with at

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

  `https://github.com/cornell-brg/pydgin`



Derek Lockhart, Ji Kim, Shreesha Srinath, Christopher Torng,
Berkin Ilbeyi, Moyang Wang, Shunning Jiang, Khalid Al-Hawaj,
and many M.S./B.S. students

Equipment and Tool Donations
Intel, NVIDIA, Synopsys, Xilinx