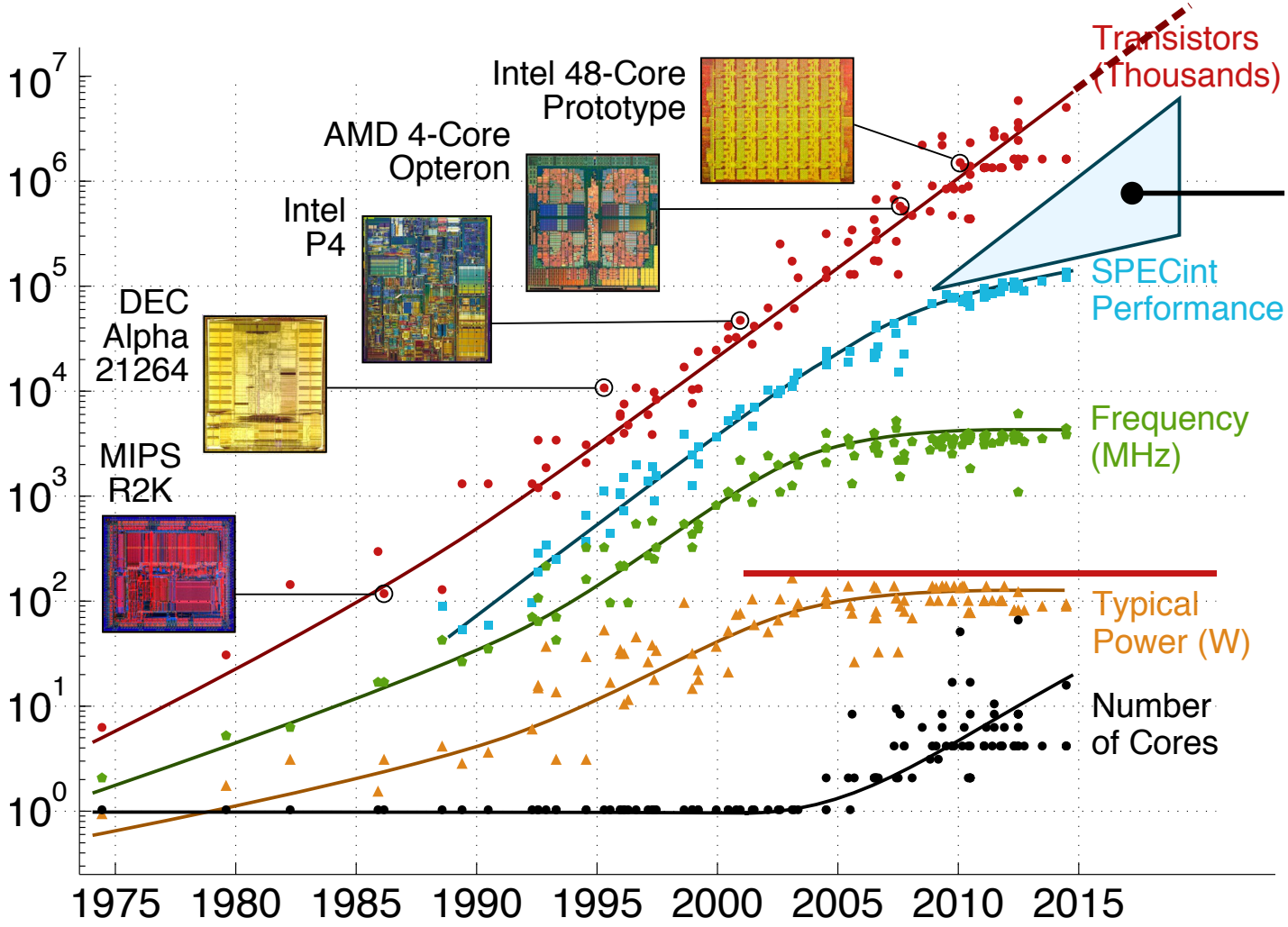
Architectural Specialization for Inter-Iteration Loop Dependence Patterns

Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

Spring 2015

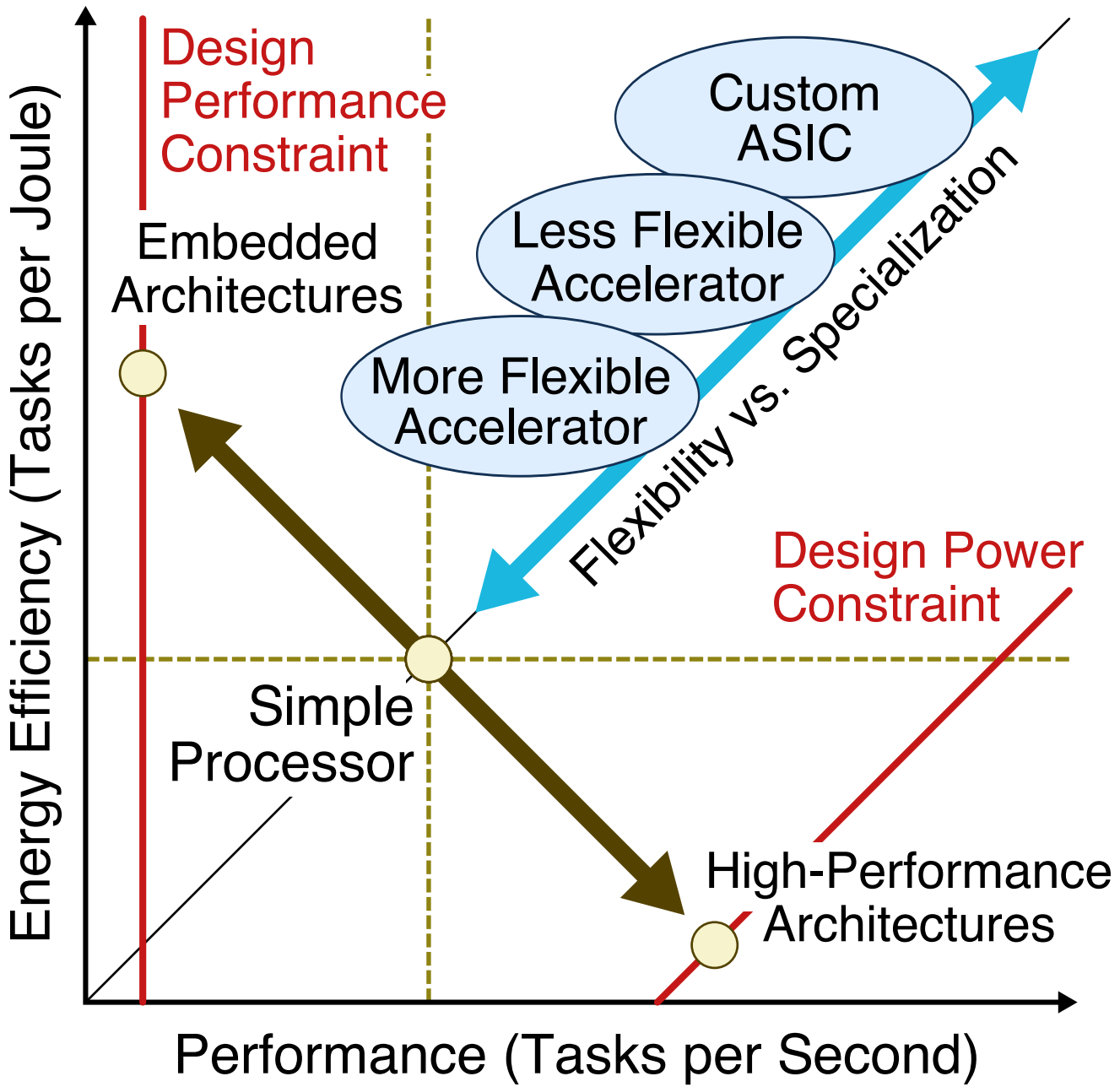
Motivating Trends in Computer Architecture



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

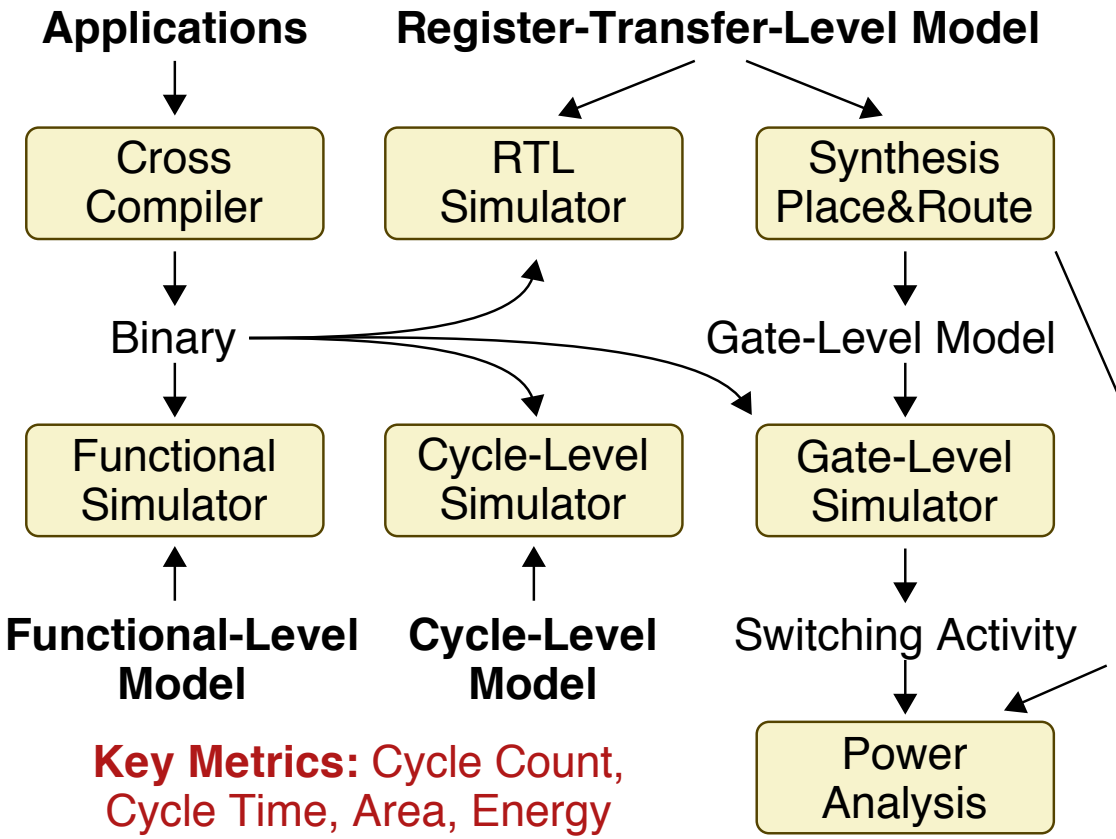
Specialization

- 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)

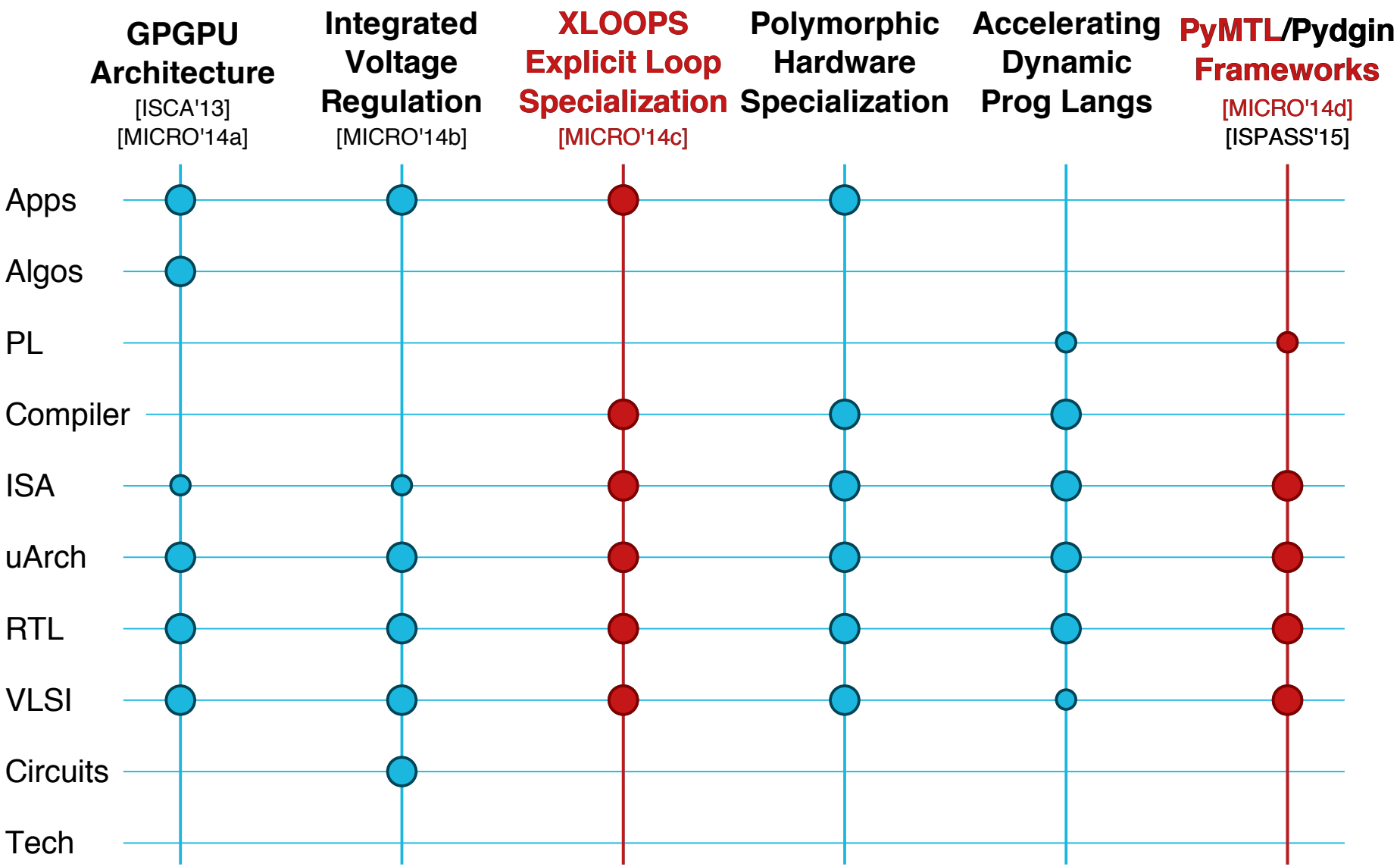
Vertically Integrated Research Methodology

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



Experimenting with full-chip layout, FPGA prototypes, and test chips is a key part of our research methodology

Projects Within the Batten Research Group

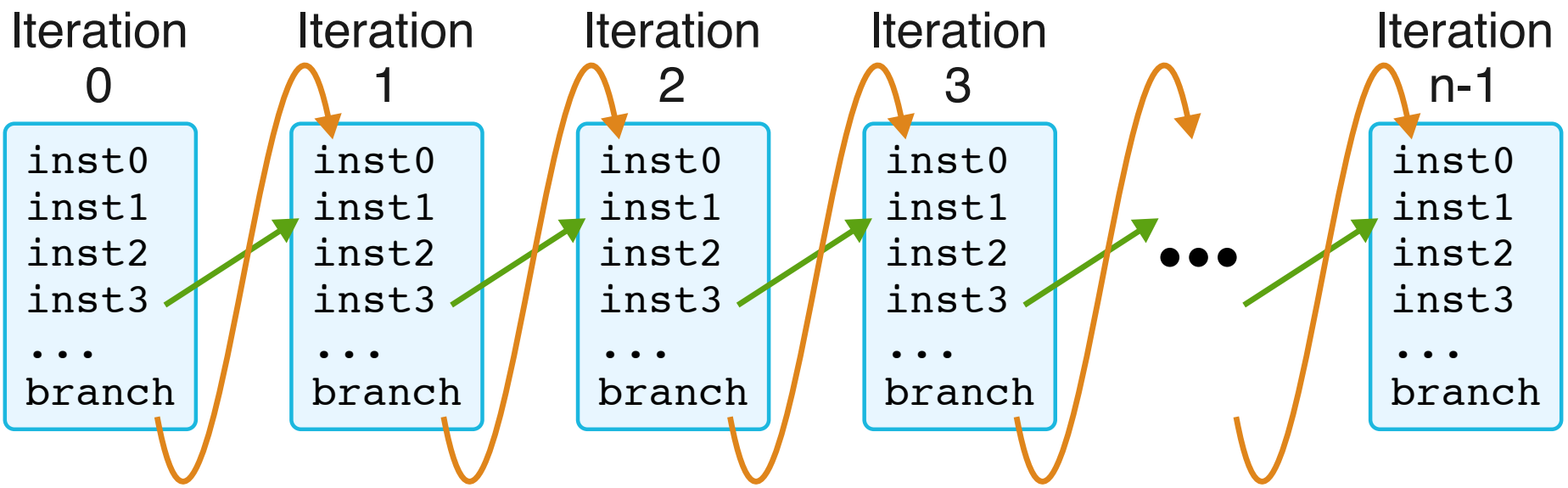


XLOOPS: Architectural Specialization for Inter-Iteration Loop Dependence Patterns

Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten

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

Loop Dependence Pattern Specialization



Intra-Iteration
Micro-op Fusion,
ASIPs, CCA, BERET

Inter-Iteration
Vector, GPU,
HELIX-RC

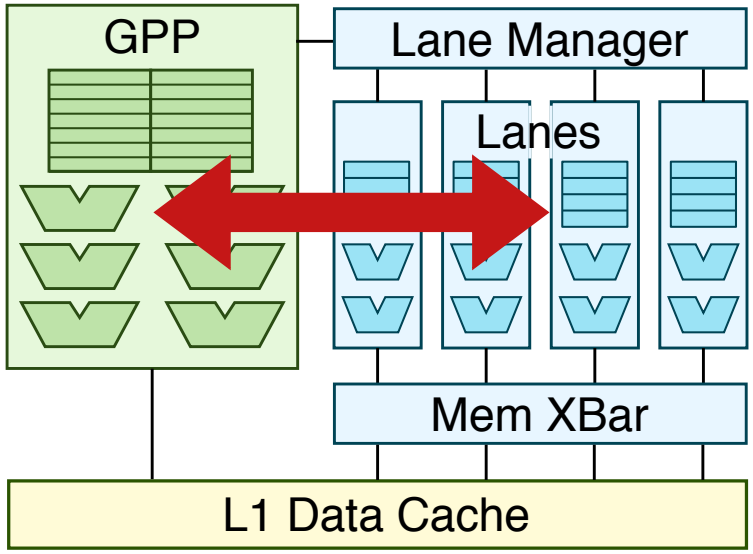
Both
DySER, C-Cores,
Qs-Cores

Key Challenge: Creating HW/SW abstractions that are flexible and enable performance-portable execution

Explicit Loop Specialization (XLOOPS)

Key Idea 1: Expose fine-grained parallelism by elegantly encoding inter-iteration loop dependence patterns in the ISA

Key Idea 2: Single-ISA hetereogenous architecture with a new execution paradigm supporting traditional, specialized, and adaptive execution



- **Traditional Execution**
- **Specialized Execution**
- **Adaptive Execution**

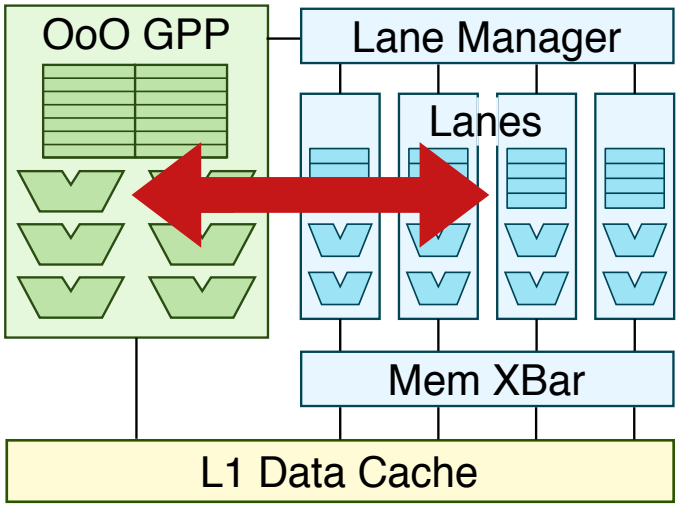
1. XLOOPS Instruction Set

```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 addiu.xi  rA, 4
 addiu.xi  rB, 4
 addiu     r1, r1, 1
 xloop.uc  r1, rN, loop
```

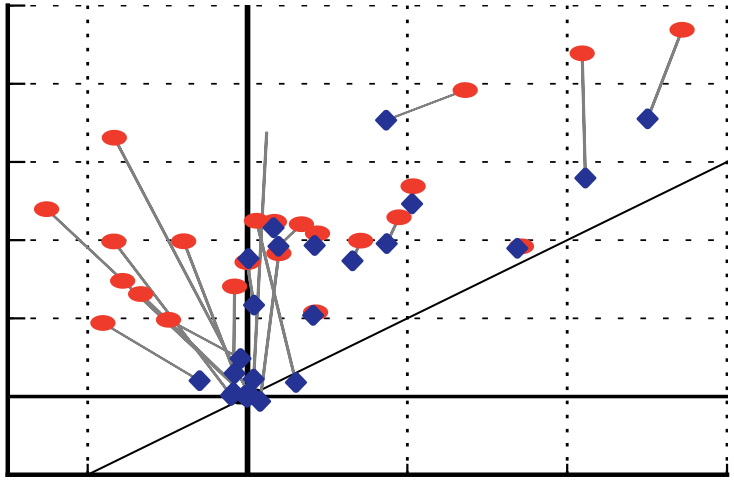
2. XLOOPS Compiler

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```

3. XLOOPS Microarchitecture



4. Evaluation



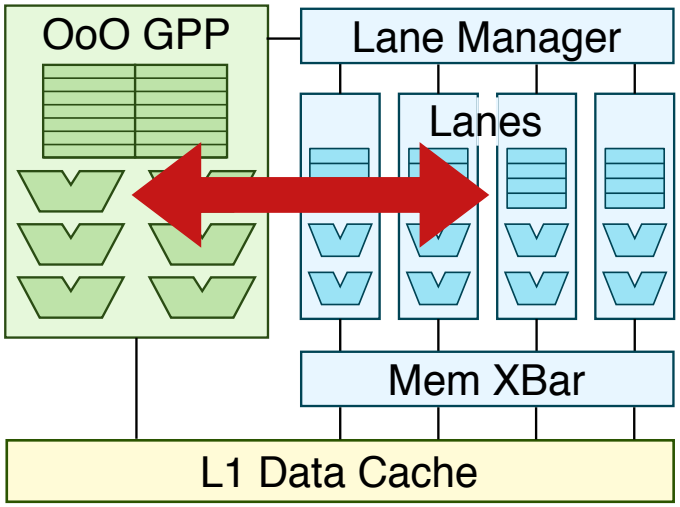
1. XLOOPS Instruction Set

```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 addiu.xi   rA, 4
 addiu.xi   rB, 4
 addiu      r1, r1, 1
 xloop.uc   r1, rN, loop
```

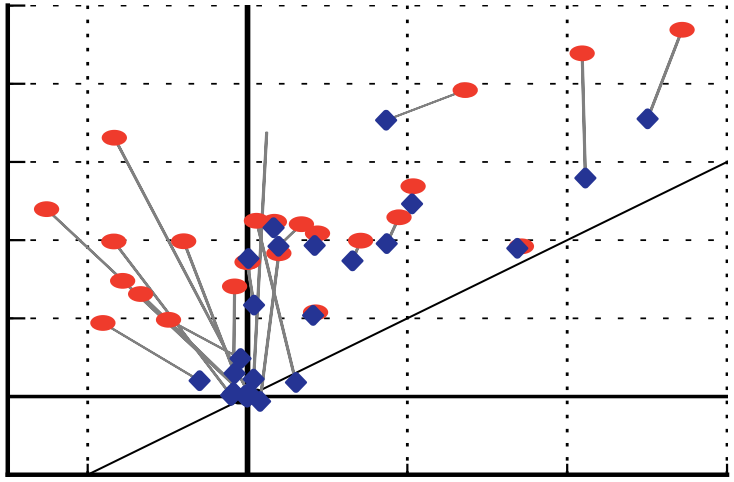
2. XLOOPS Compiler

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```

3. XLOOPS Microarchitecture



4. Evaluation



XLOOPS Instruction Set Extensions

XLOOP Instruction

```
xloop.{d}.{c}   rI,   rN,   L
```

Data Dependence

Control Dependence

Induction Variable

Loop Bound

Loop Label

```
xloop.uc.fb   r2, r3, 0x8000
```

Unordered Concurrent

Fixed Bound

Cross-Iteration Instructions

```
addiu.xi    rX, imm
addu.xi     rX, rT
```

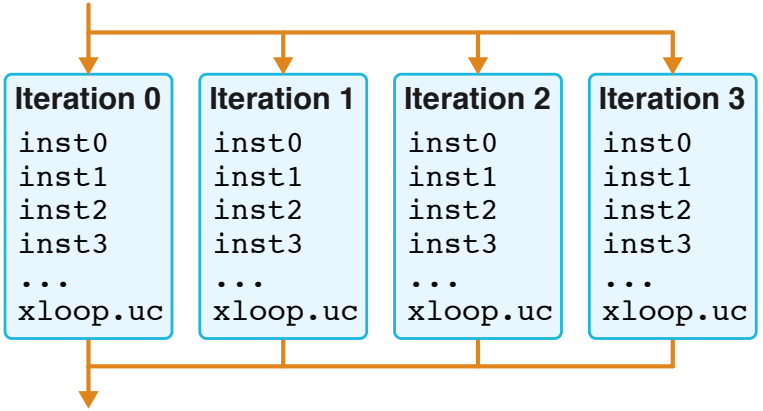
Variables that can be computed as linear functions of the induction variable

XLOOPS Instruction Set: Unordered Concurrent

Element-wise Vector Multiplication

```
for ( i=0; i<N; i++ )
  C[i] = A[i] * B[i]
```

```
loop:
  lw         r2, 0(rA)
  lw         r3, 0(rB)
  mul        r4, r2, r3
  sw         r4, 0(rC)
  addiu.xi   rA, 4
  addiu.xi   rB, 4
  addiu.xi   rC, 4
  addiu      r1, r1, 1
  xloop.uc   r1, rN, loop
```



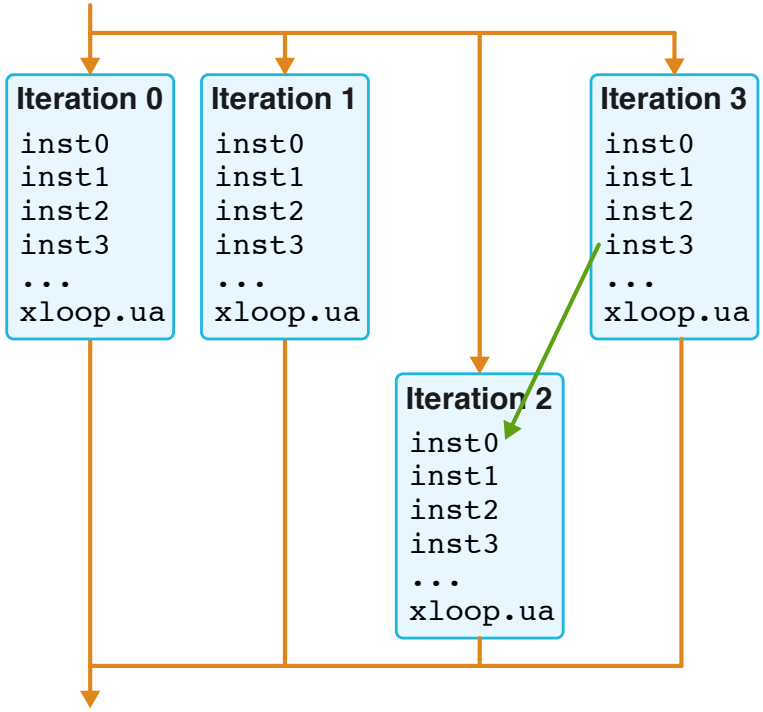
- Instructions in loop cannot write live-in registers
- Live-out values must be stored to memory
- Data-races are possible

XLOOPS Instruction Set: Unordered Atomic

Histogram Updates

```
for ( i=0; i<N; i++ )
  B[A[i]]++; D[C[i]]++;
```

```
 loop:
   lw        r6, 0(rA)
   lw        r7, 0(r6)
   addiu     r7, r7, 1
   sw        r7, 0(r6)
   addiu.xi  rA, 4
   ...
   addiu     r1, r1, 1
   xloop.ua  r1, rN, loop
```



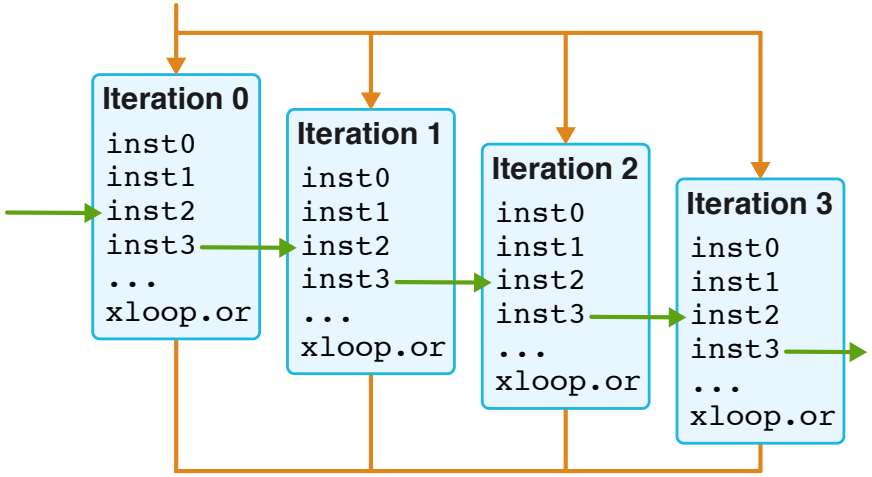
- Iterations execute atomically
- No race conditions
- Results can be non-deterministic
- Inspired by Transactional Memory

XLOOPS Instruction Set: Ordered-Through-Registers

Parallel-Prefix Summation

```
for ( i=0; i<N; i++ )
  X += A[i]; B[i] = X
```

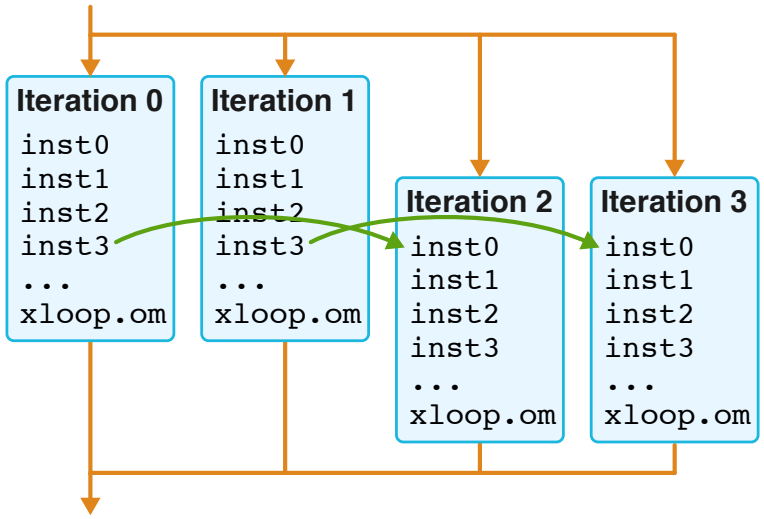
```
 loop:
   lw         r2, 0(rA)
   addu       rX, r2, rX
   sw         rX, 0(rB)
   addiu.xi   rA, 4
   addiu.xi   rB, 4
   addiu      r1, r1, 1
   xloop.or   r1, rN, loop
```



- rX - Cross Iteration Register
- CIRs are guranteed to have the same value as a serial execution
- Inspired by Multiscalar

XLOOPS Instruction Set: Ordered-Through-Memory

```
for ( i=0; i<N; i++ )
  A[i] = A[i] * A[i-k];

   # r1 = rK
   # r3 = rA + 4*rK
 loop:
   lw         r4, 0(r3)
   lw         r5, 0(rA)
   mul        r6, r4, r5
   sw         r6, 0(r3)
   addiu.xi   r3, 4
   addiu.xi   rA, 4
   addiu      r1, r1, 1
   xloop.om   r1, rN, loop
```

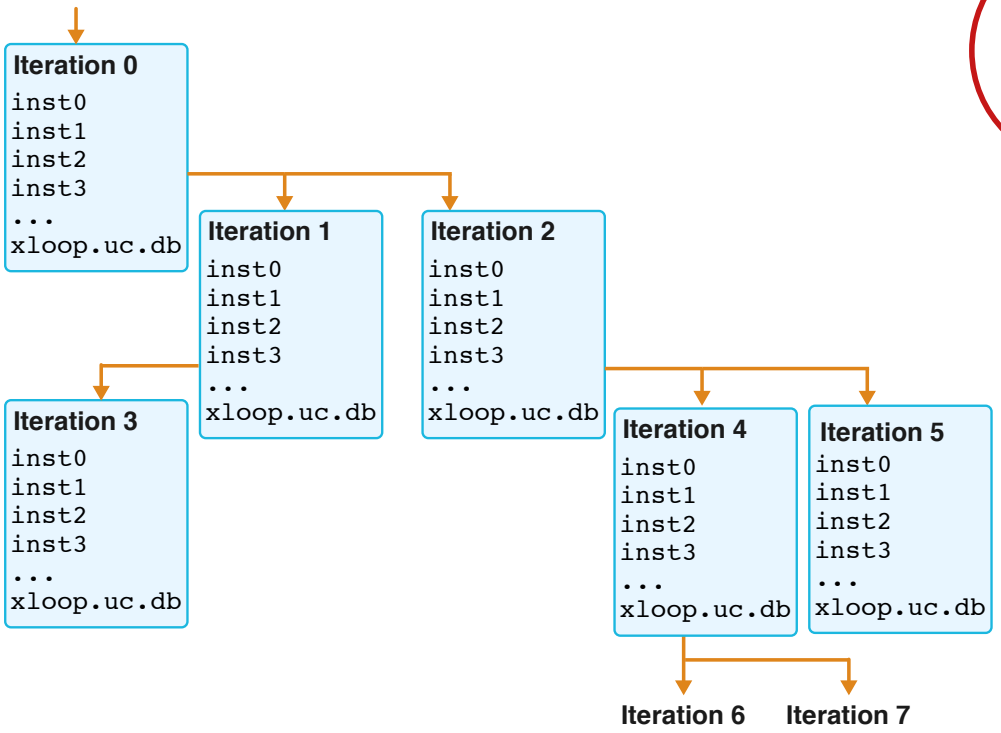


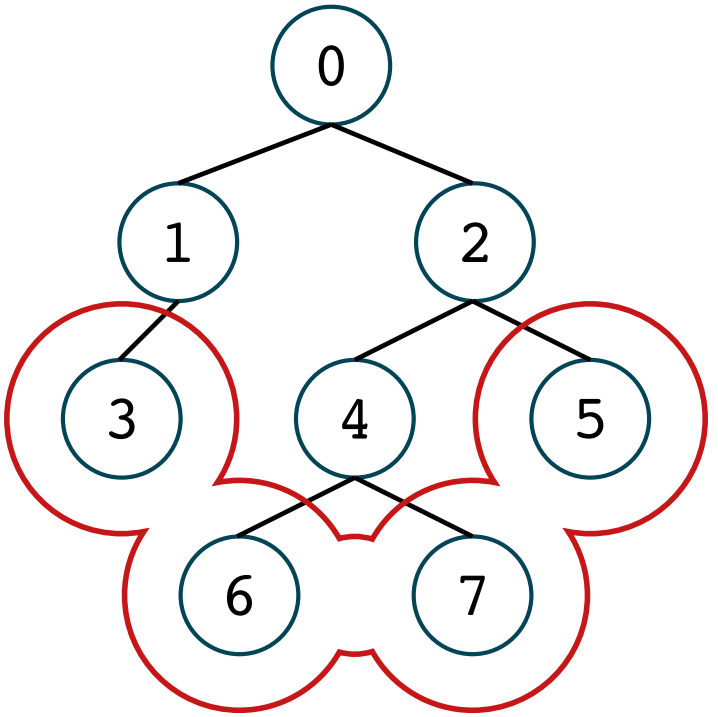
- Updates to memory defined by serial iteration order
- No race conditions
- Inspired by Multiscalar, TLS

XLOOPS Instruction Set: Dynamic Bound

- Parallelize using `xloop.uc.db`

```
for ( i=0; i<N; i++ )
  ...
  if ( cond ) N++;
```





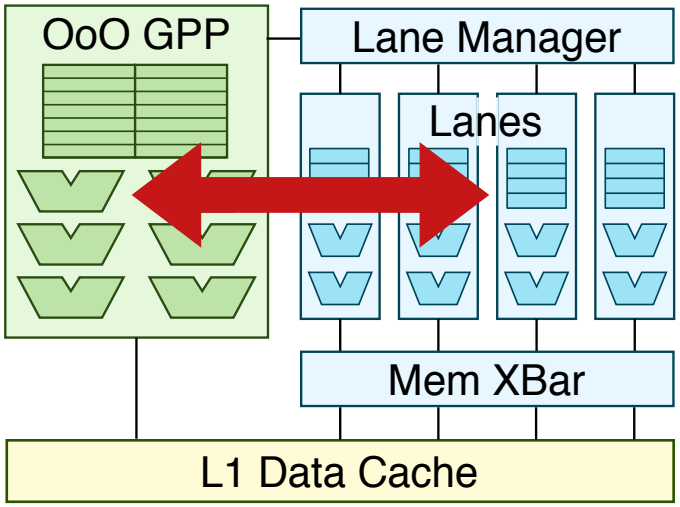
1. XLOOPS Instruction Set

```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 addiu.xi   rA, 4
 addiu.xi   rB, 4
 addiu      r1, r1, 1
 xloop.uc   r1, rN, loop
```

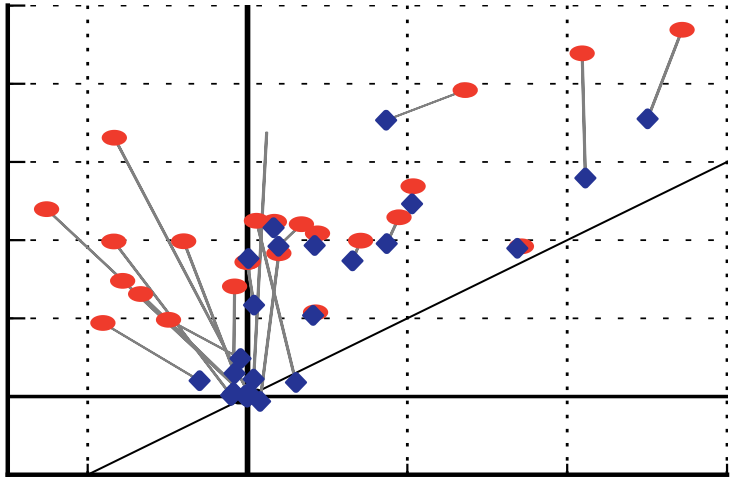
2. XLOOPS Compiler

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```

3. XLOOPS Microarchitecture



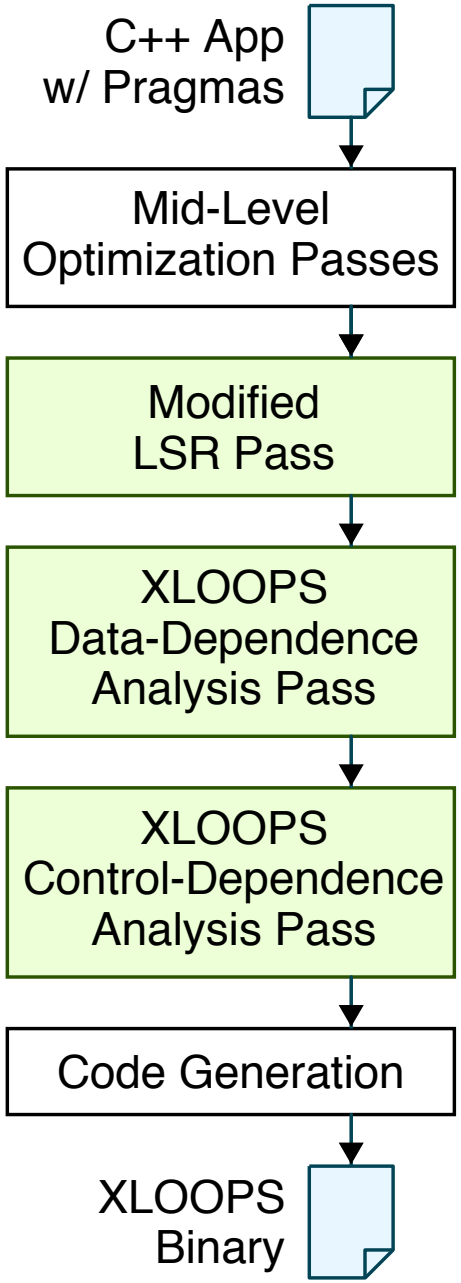
4. Evaluation



XLOOPS Compiler

Kernel implementing Floyd-Warshall shortest path algorithm

```
for ( int k = 0; k < n; k++ )

  #pragma xloops ordered
  for ( int i = 0; i < n; i++ )

    #pragma xloops unordered
    for ( int j = 0; j < n; j++ )
      path[i][j] = min( path[i][j], path[i][k] + path[k][j] );
```



- Programmer annotations
  - `unordered`: no data-dependences
  - `ordered`: preserve data-dependences
  - `atomic`: atomic memory updates
- Loop strength reduction pass encodes MIVs as `xi` instructions
- XLOOPS data-dependence analysis pass
  - Register-dependence: analysing use-definition chains through PHI nodes
  - Memory-dependence: well known dependence analysis techniques
- Detect updates to the loop bound to encode dynamic-bound control-dependence pattern

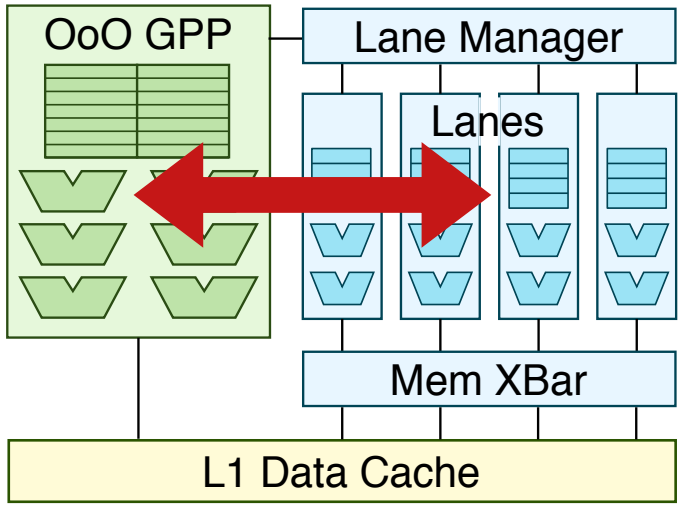
1. XLOOPS Instruction Set

```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 addiu.xi   rA, 4
 addiu.xi   rB, 4
 addiu      r1, r1, 1
 xloop.uc   r1, rN, loop
```

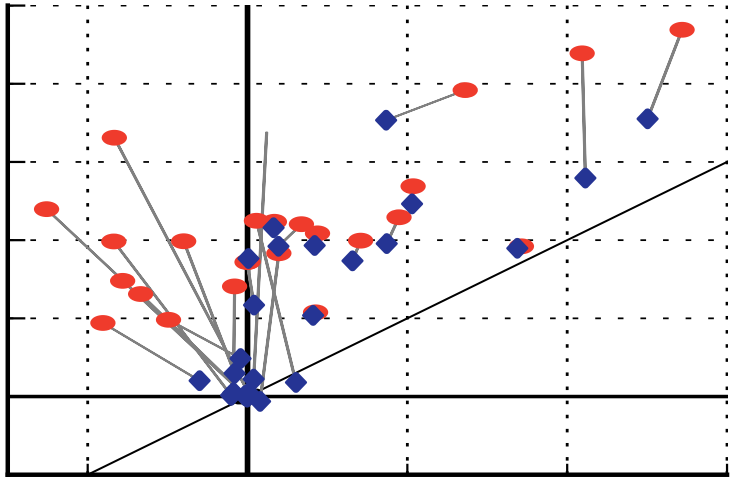
2. XLOOPS Compiler

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```

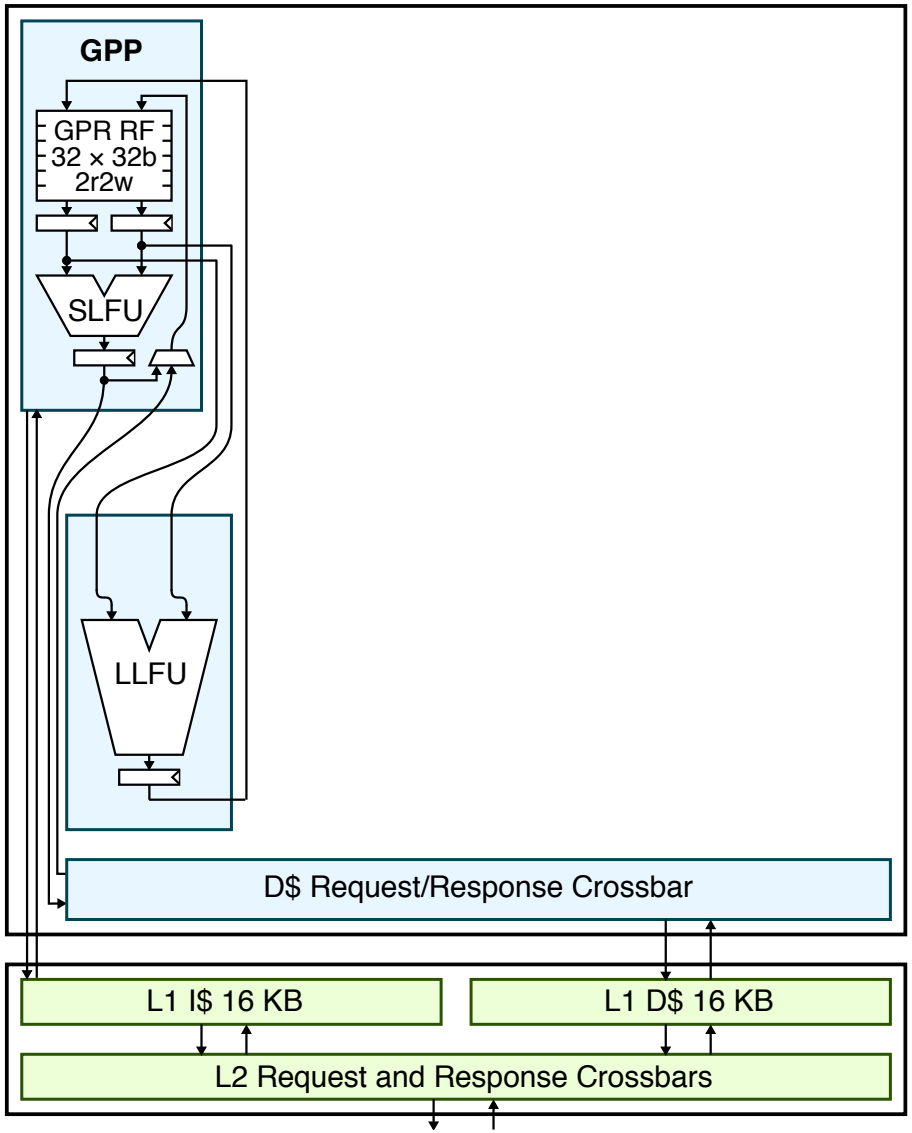
3. XLOOPS Microarchitecture



4. Evaluation



Traditional Execution



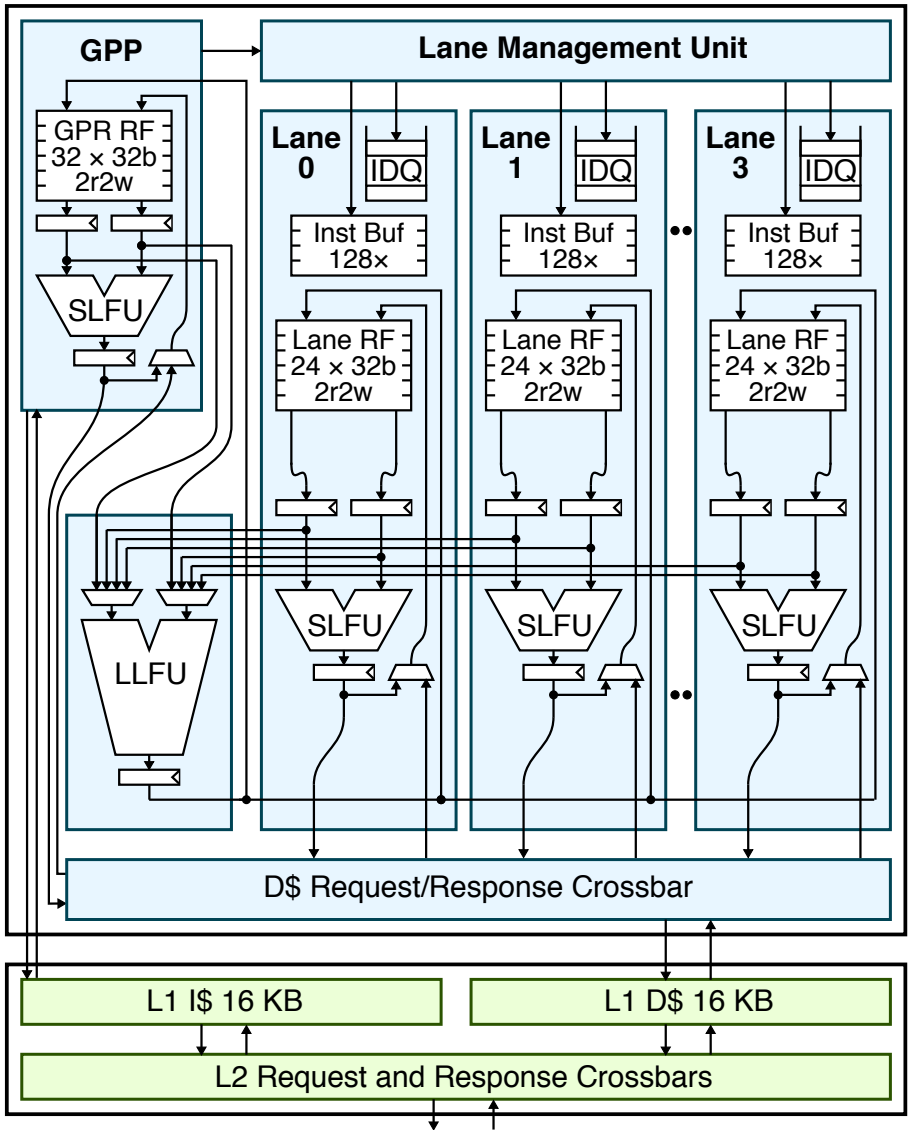
Minimal changes to a general-purpose processor (GPP)

- `xloop` → `bne`
- `addiu.xi` → `addiu`
- `addu.xi` → `addu`

Efficient traditional execution

- Enables gradual adoption
- Enables adaptive execution to migrate an `xloop` instruction

Specialized Execution – `xloop.uc`



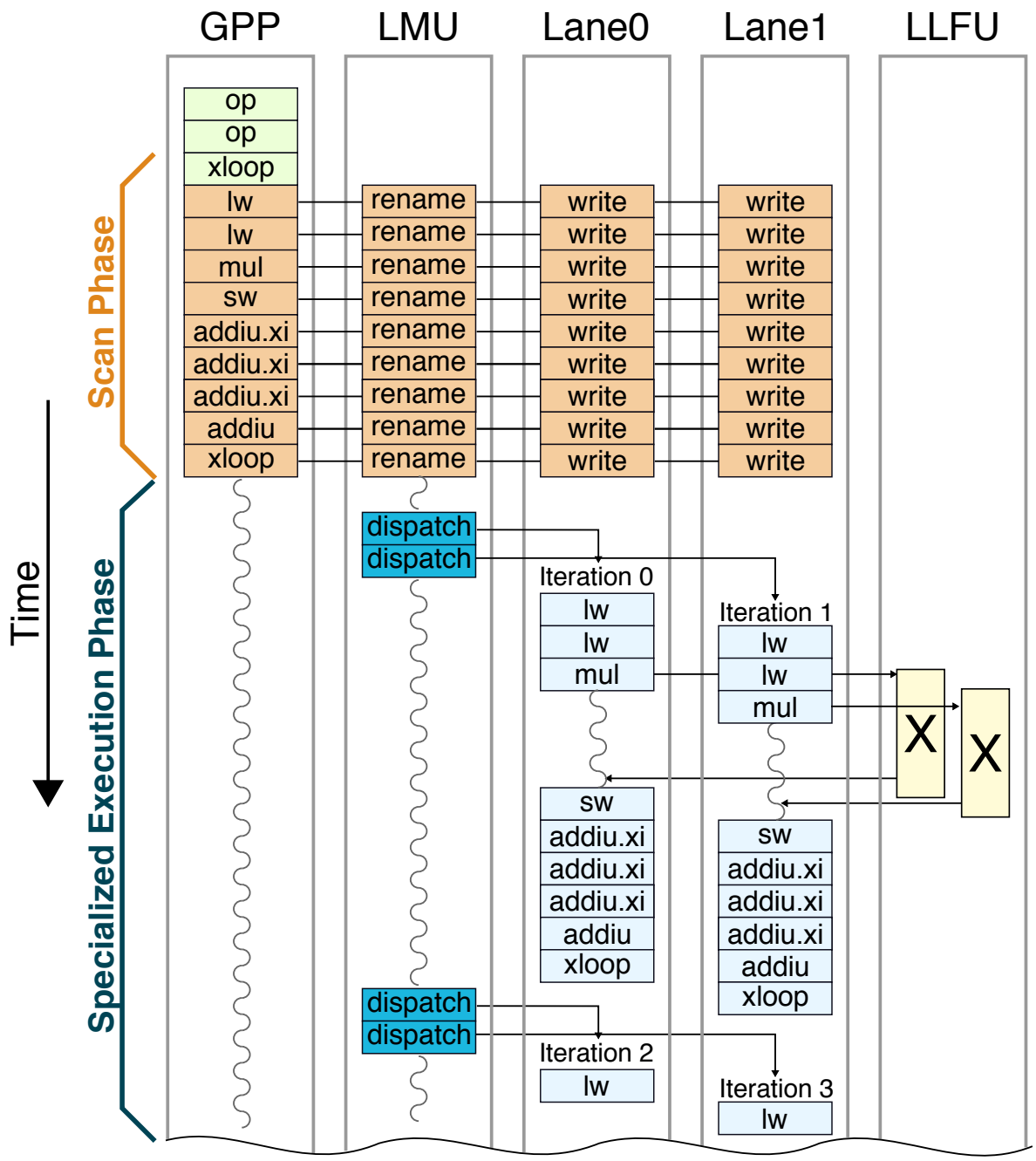
Loop Pattern Specialization Unit

- Lane Management Unit (LMU)
- Four decoupled in-order lanes
- Lanes contain instruction buffers and index queues
- Lanes and the GPP arbitrate for data-memory port and long-latency functional unit

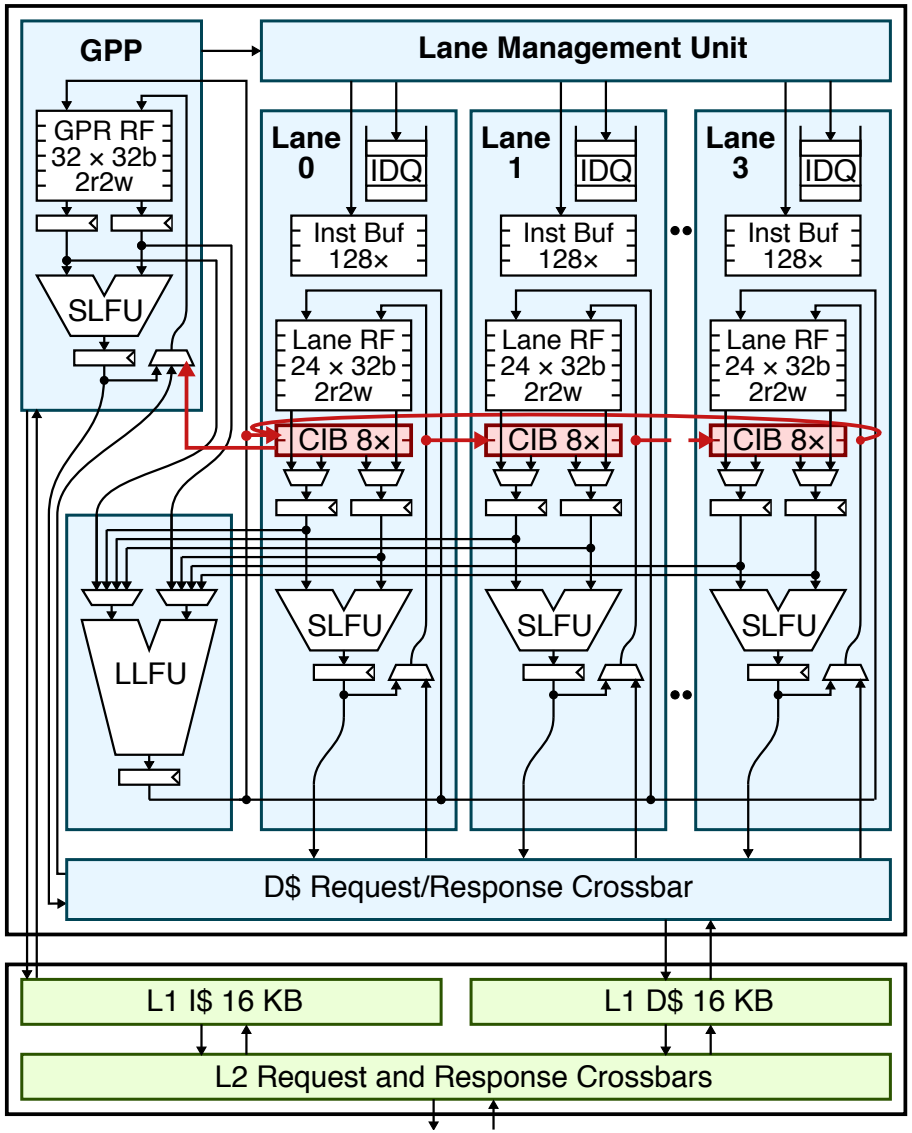
Specialized execution

- Scan phase
- Specialized execution phase

```
loop:
  lw          r2, 0(rA)
  lw          r3, 0(rB)
  mul         r4, r2, r3
  sw          r4, 0(rC)
  addiu.xi    rA, 4
  addiu.xi    rB, 4
  addiu.xi    rC, 4
  addiu       r1, r1, 1
  xloop.uc    r1, rN, loop
```

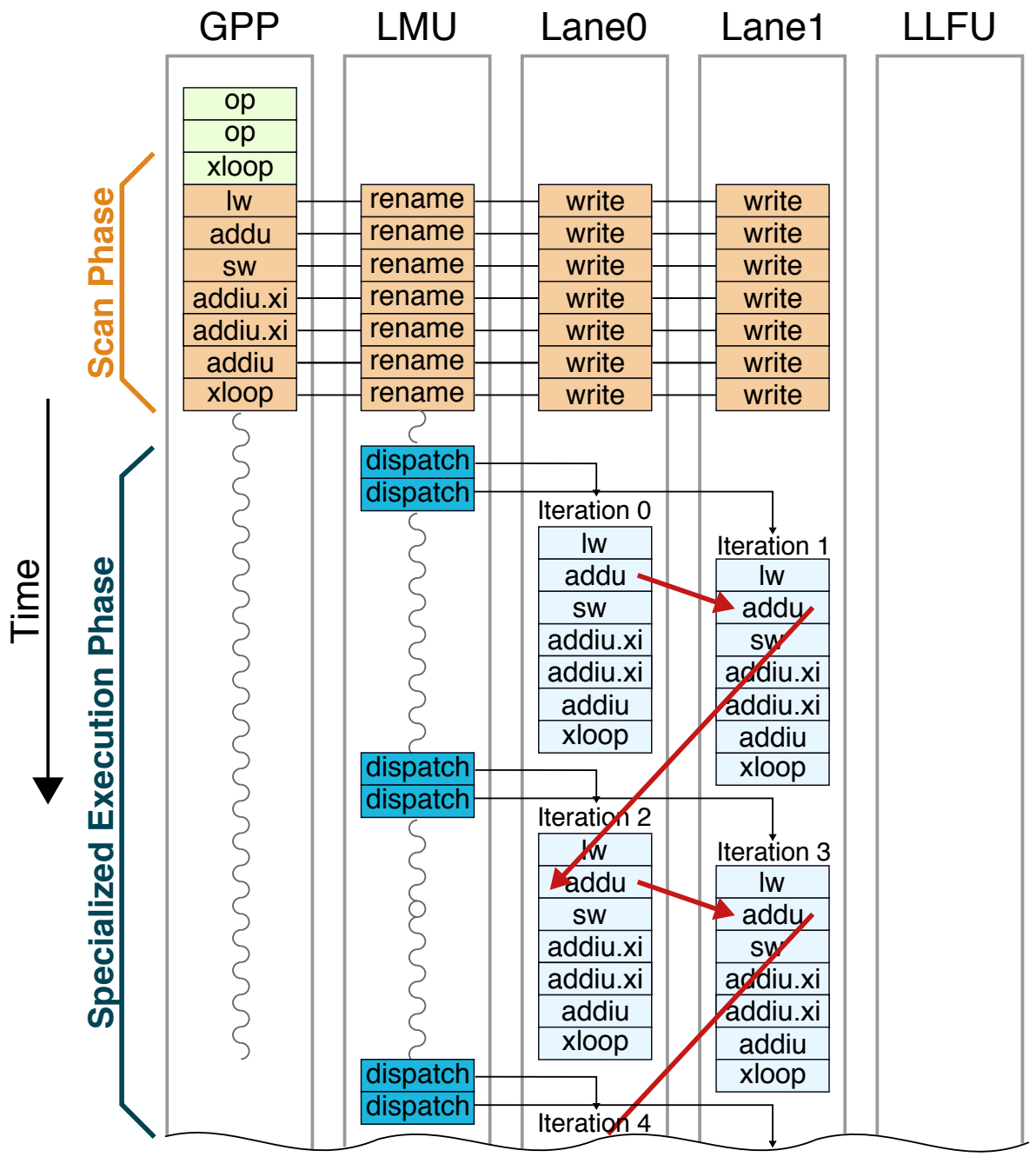


Specialized Execution – `xloop.or`

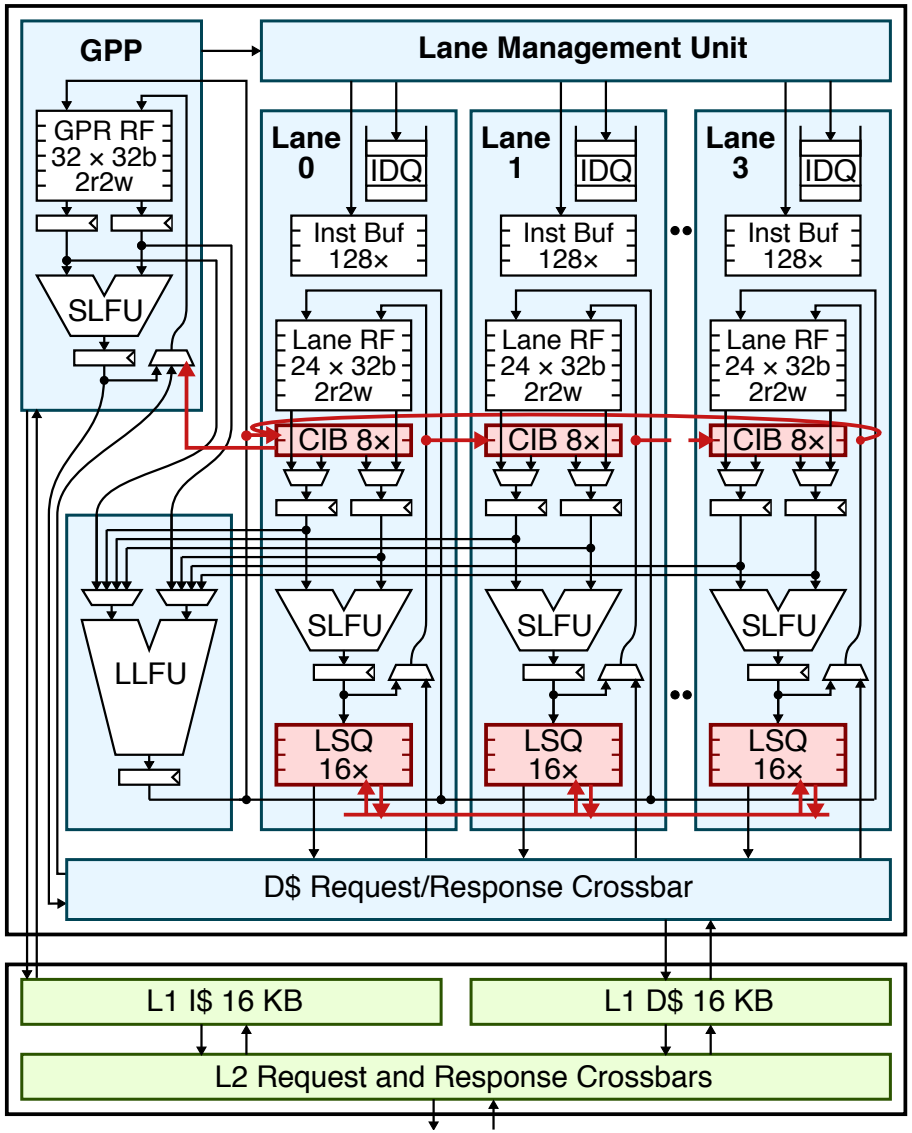


- **Cross-iteration buffers** (CIBs) forward register-dependences
- **LMU control logic**
  - Cross-iteration registers (CIRs)
  - Last update to a CIR
- **Lane control logic**
  - Stall if CIR is not available
  - If last update to CIR then write to the next CIB

```
loop:
  lw          r2, 0(rA)
  addu        rX, r2, rX
  sw          rX, 0(rB)
  addiu.xi    rA, 4
  addiu.xi    rB, 4
  addiu       r1, r1, 1
  xloop.or    r1, rN, loop
```

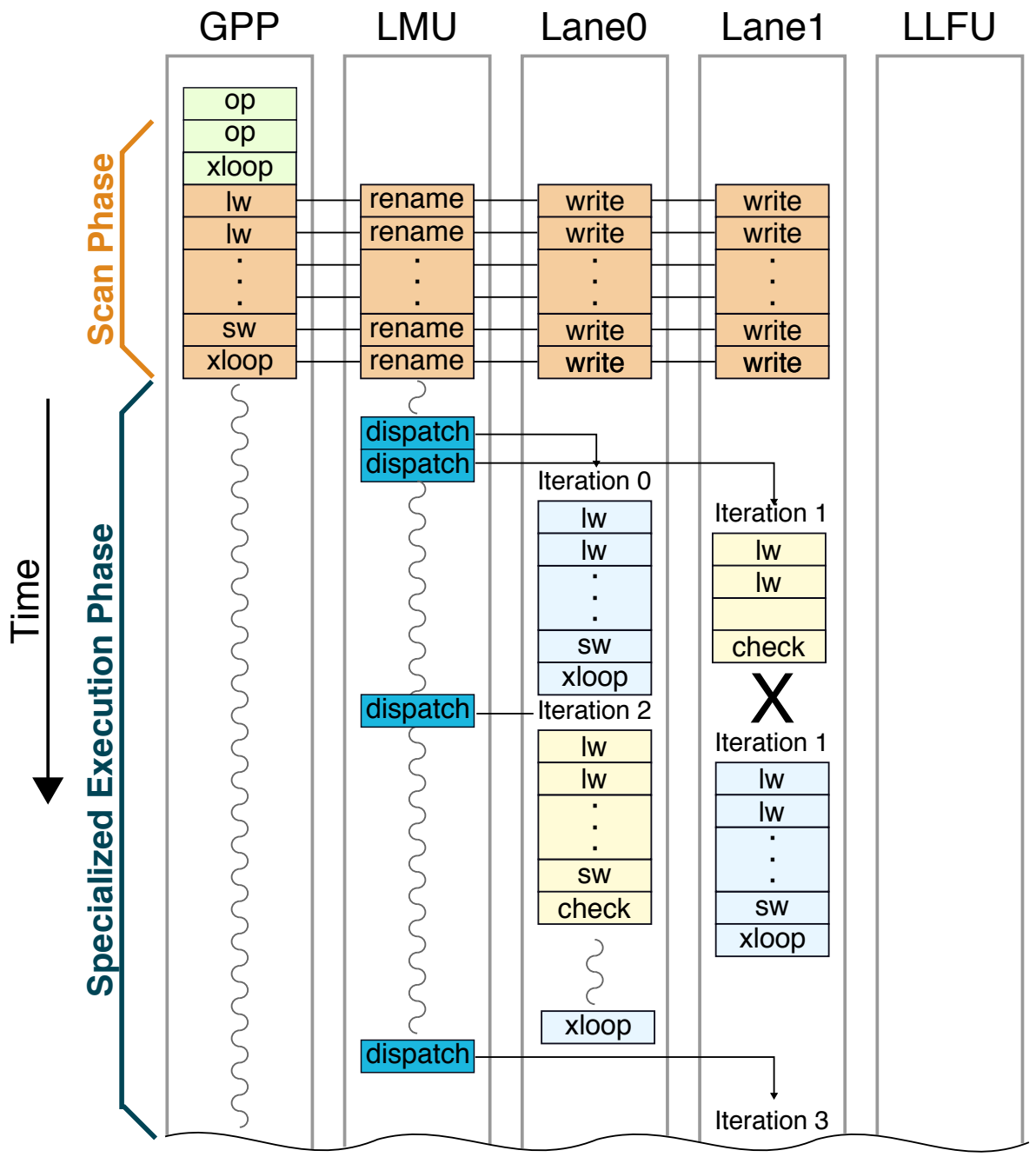


Specialized Execution – `xloop.om`

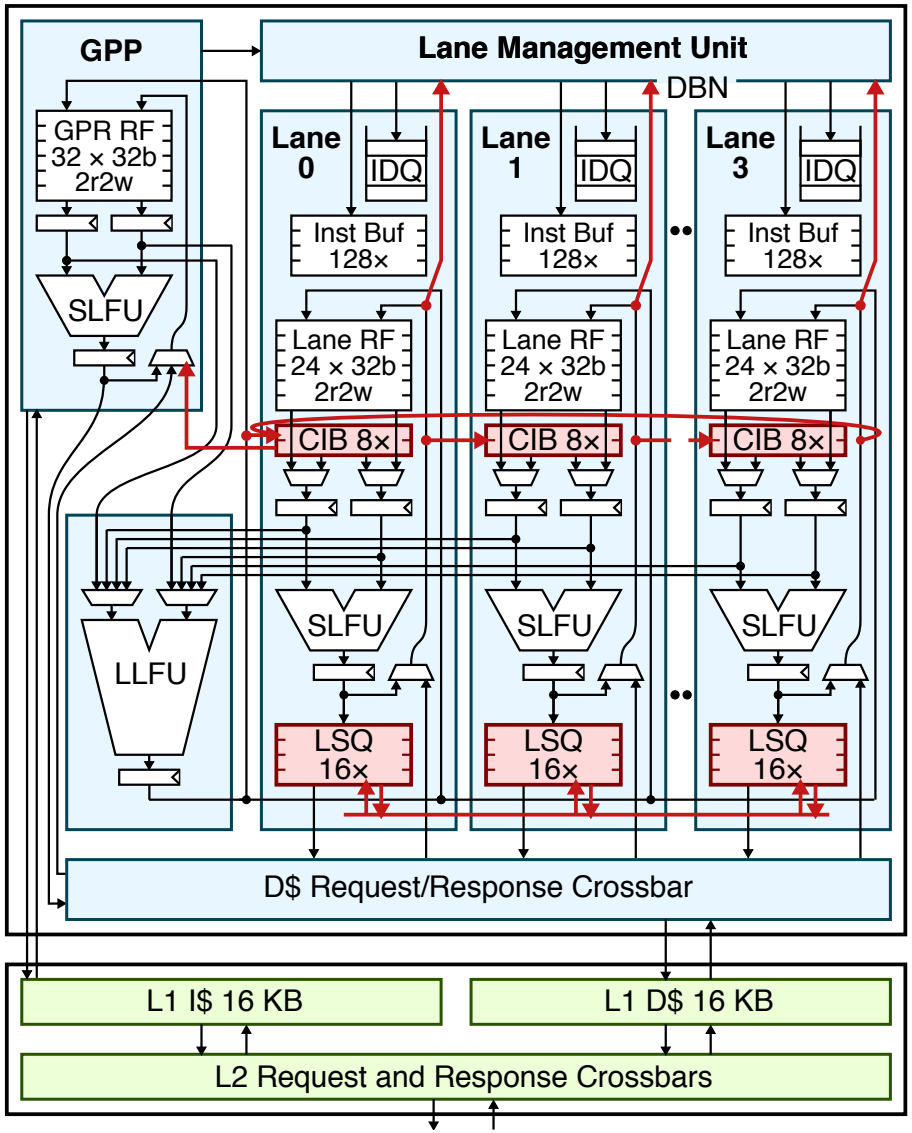


- **LSQ** to support hardware memory disambiguation
- **LMU control logic**
  - Track non-speculative vs. speculative lanes
  - Promote lanes to be non-speculative
- **Lane control logic**
  - Handle structural hazards
  - Handle dependence violations

```
loop:
 lw          r4, 0(r3)
 lw          r5, 0(rA)
 ...
 ...
 sw          r6, 0(r7)
 addiu       r1, r1, 1
 xloop.om    r1, rN, loop
```

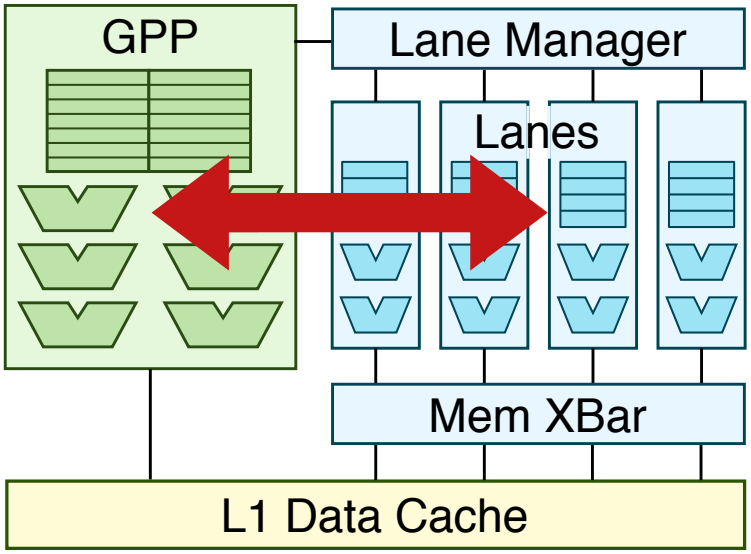


Supporting other patterns



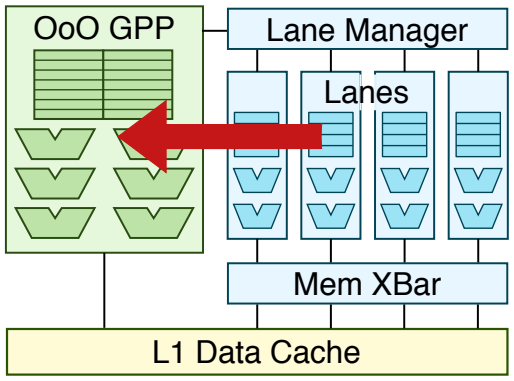
- `xloop.ua` – Using `xloop.om` mechanisms
- `xloop.orm` – Combine `xloop.or` and `xloop.om` mechanisms
- `xloop.*.db`
  - Lanes communicate updates to loop bound
  - LMU tracks maximum bound and generates additional work

Adaptive Execution

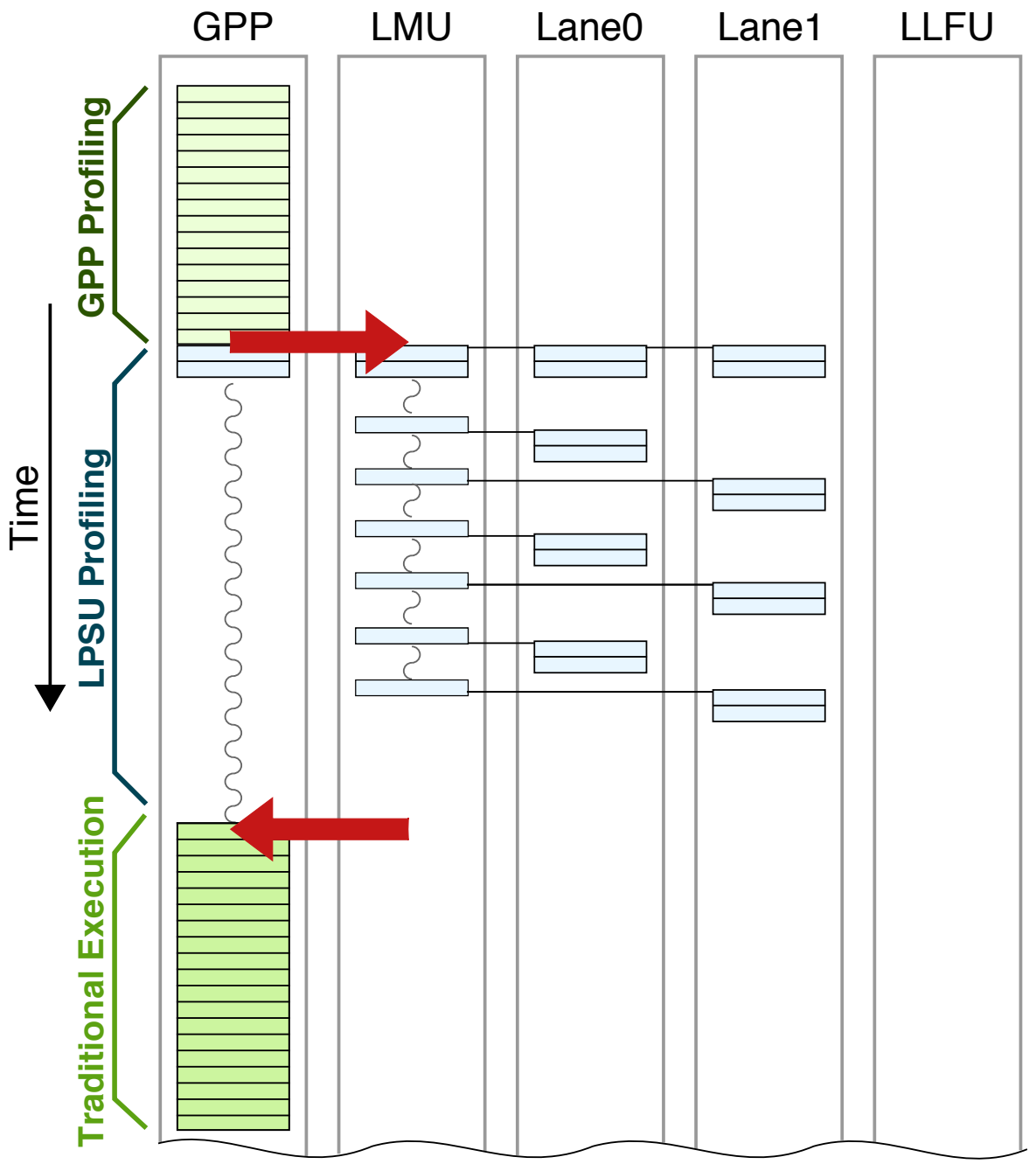


- Some kernels have higher performance on LPSU (e.g., significant inter-iteration parallelism)
- Some kernels have higher performance on GPP (e.g., limited inter-iteration parallelism, significant intra-iteration parallelism)

- **Approach #1:** Move to more complicated superscalar or out-of-order lanes to better exploit both inter- and intra-iteration parallelism
- **Approach #2:** Adaptively migrate between traditional and specialized execution to achieve best performance



- Migrating loop on iteration boundaries is very cheap and usually only requires sending the next iteration index
- An adaptive profiling table in GPP records profiling progress for small number of recently seen `xloop` instructions



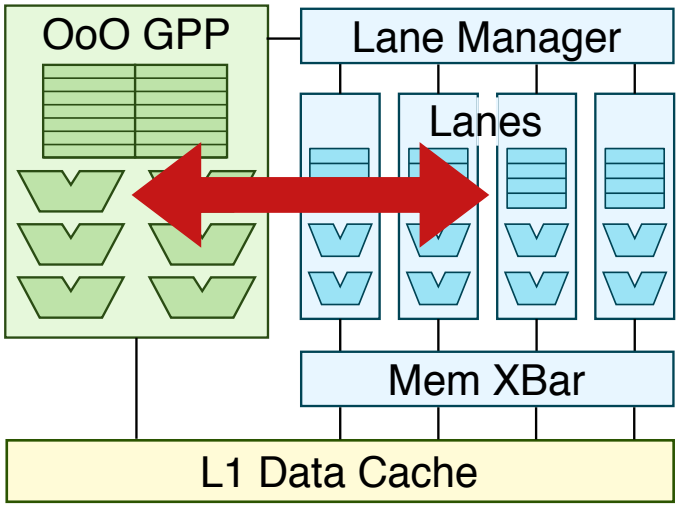
1. XLOOPS Instruction Set

```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 addiu.xi   rA, 4
 addiu.xi   rB, 4
 addiu      r1, r1, 1
 xloop.uc   r1, rN, loop
```

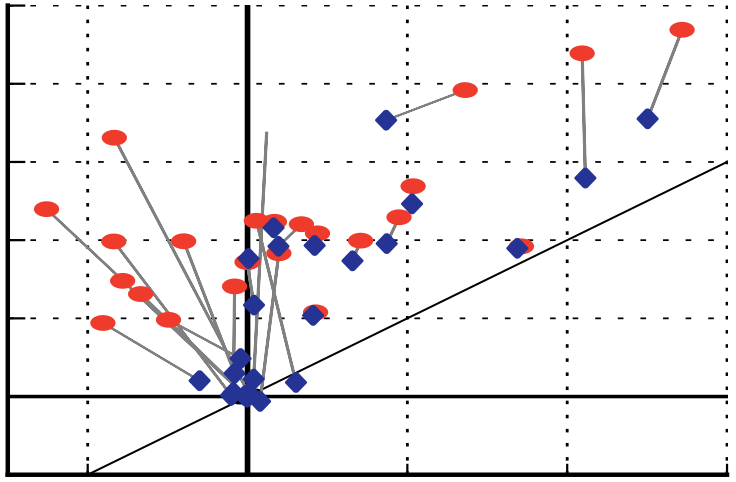
2. XLOOPS Compiler

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```

3. XLOOPS Microarchitecture



4. Evaluation



Application Kernels

`xloop.uc`

Color space conversion
Dense matrix-multiply
String search algorithm
Symmetric matrix-multiply
Viterbi decoding algorithm
Floyd-Warshall shortest path

`xloop.or`

ADPCM decoder
Covriance computation
Floyd-Steinberg dithering
K-Means clustering
SHA-1 encryption kernel
Symmetric matrix-multiply

`xloop.om`

Dynamic-programming
K-Nearest neighbors
Knapsack kernel
Floyd-Warshall shortest path

`xloop.orm, xloop.ua`

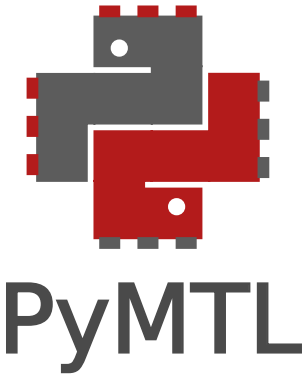
Greedy maximal-matching
2D Stencil computation
Binary tree construction
Heap-sort computation
Huffman entropy coding
Radix sort algorithm

`xloop.uc.db`

Breadth-first search
Quick-sort algorithm

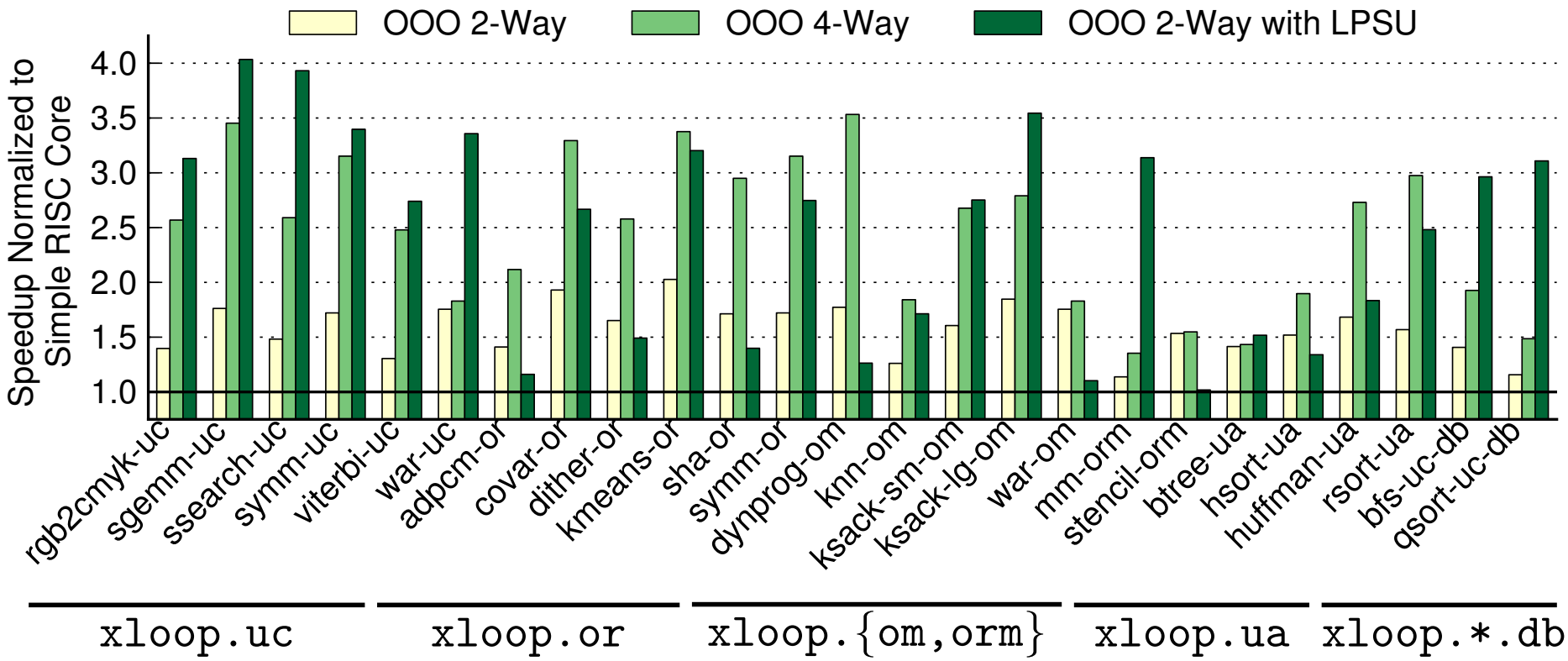
25 Kernels: MiBench, PolyBench, PBBS, custom

Cycle-Level Evaluation Methodology



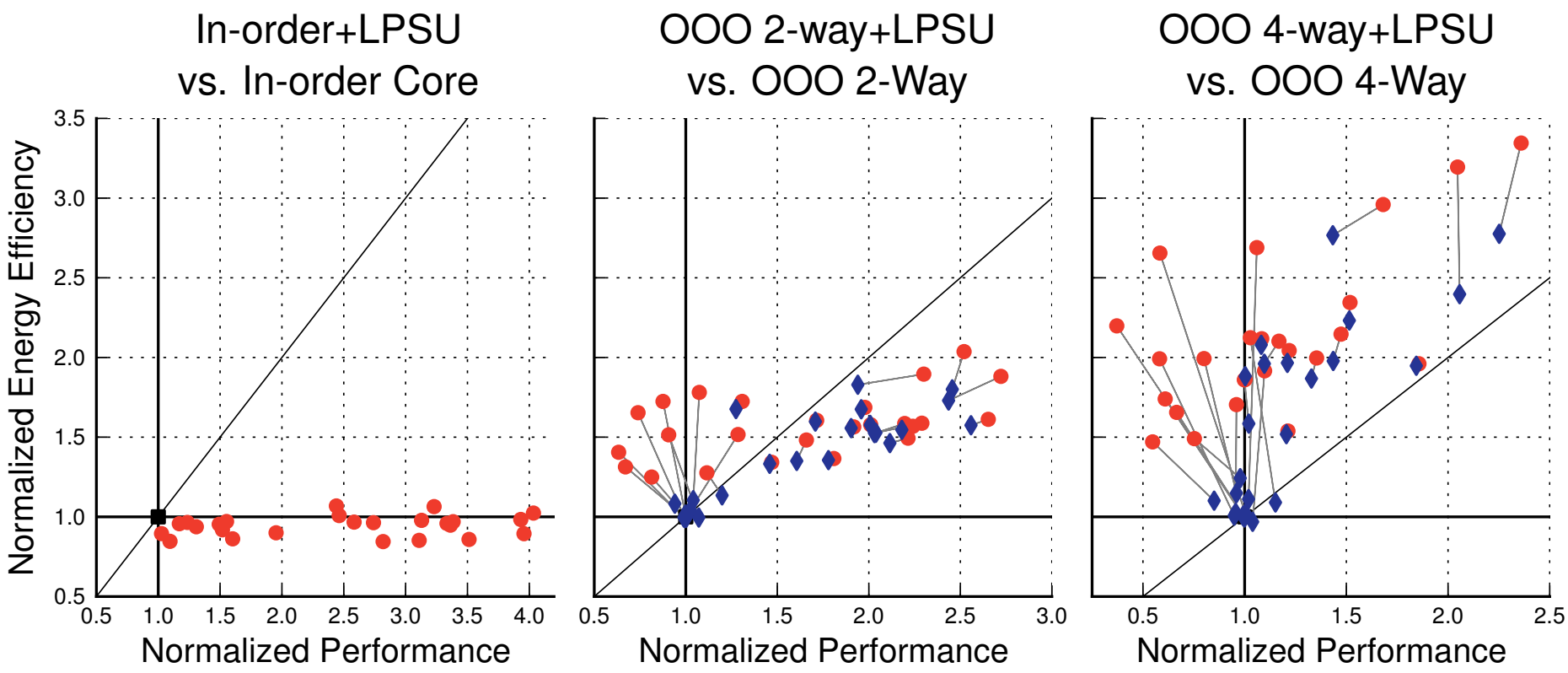
- LLVM-3.1 based compiler framework
- gem5 – in-order and out-of-order processors
- PyMTL – LPSU models
- McPAT-1.0 – 45nm energy models

XLOOPS Cycle-Level Performance Results

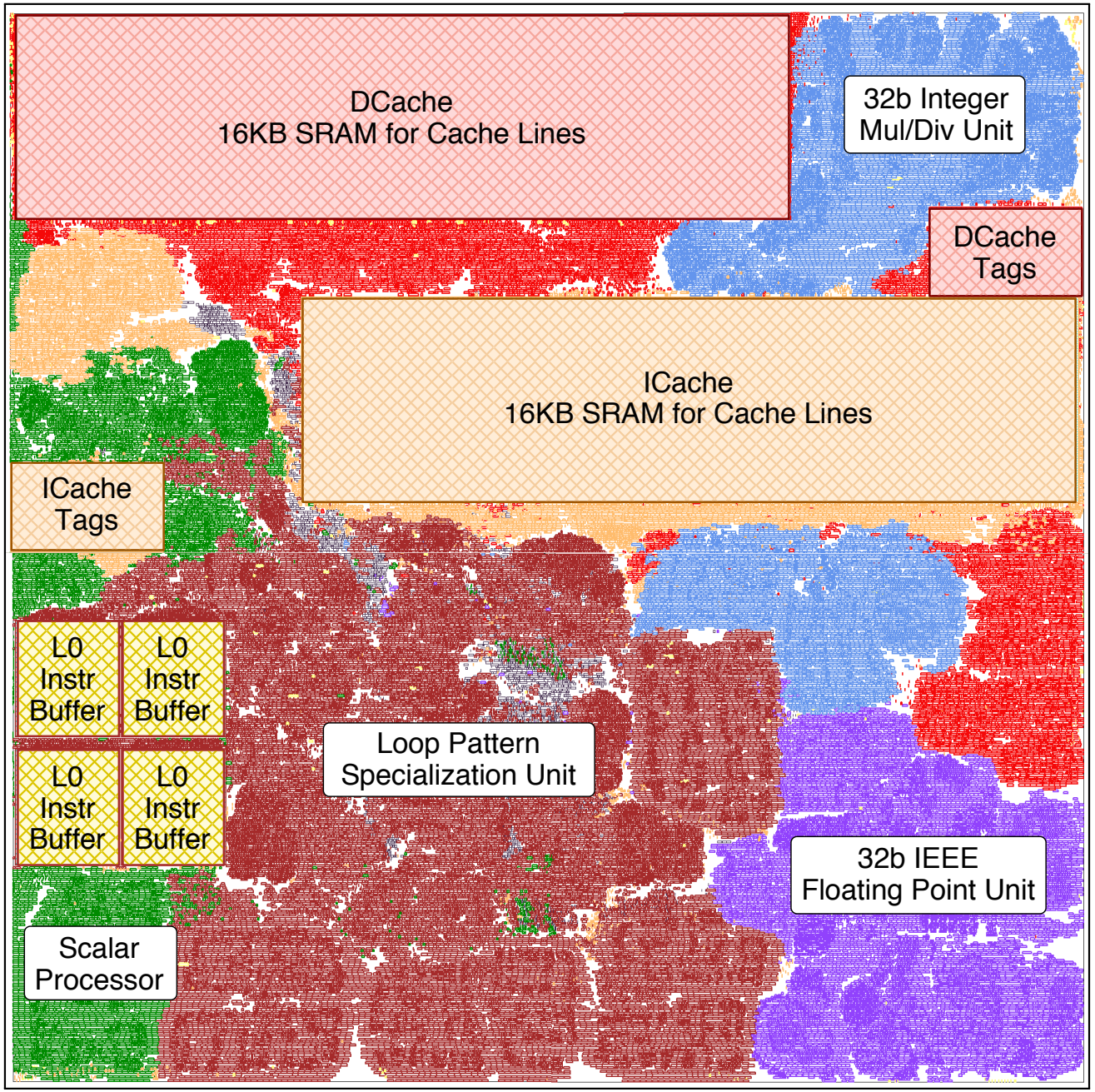


- XLOOPS vs. Simple Core : Always higher performance
- XLOOPS vs. OOO 2-way : Generally competitive or higher performance
- XLOOPS vs. OOO 4-way : Mixed results

Energy-Efficiency vs. Performance Results



- XLOOPS vs. Simple Core : Similar energy efficiency, higher power
- XLOOPS vs. OOO 2-way : Higher energy efficiency, mixed power
- XLOOPS vs. OOO 4-way : Higher energy efficiency, lower power
- Adaptive execution trades energy efficiency for performance
- Profiling and migration cause minimal performance degredation

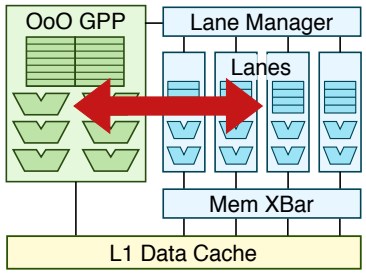


VLSI Implementation

- TSMC 40 nm standard-cell-based implementation
- RISC scalar processor with 4-lane LPSU
- Supports `xloop.uc`
- ≈40% extra area compared to simple RISC processor

```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 addiu.xi  rA, 4
 addiu.xi  rB, 4
 addiu     r1, r1, 1
 xloop.uc  r1, rN, loop
```

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```



XLOOPS Take-Away Points

- XLOOPS is an elegant new abstraction that enables performance-portable execution of loops
- XLOOPS enables a single-ISA heterogeneous architecture with a new execution paradigm
  - Traditional Execution
  - Specialized Execution
  - Adaptive Execution

This work was supported in part by the National Science Foundation (NSF), the Defense Advanced Research Projects Agency (DARPA), and donations from Intel Corporation, Synopsys, Inc., and Xilinx, Inc.

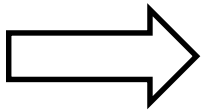
PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, and Christopher Batten

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

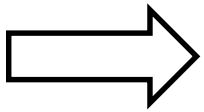
PyMTL Outline

The Computer Architecture Research Methodology Gap



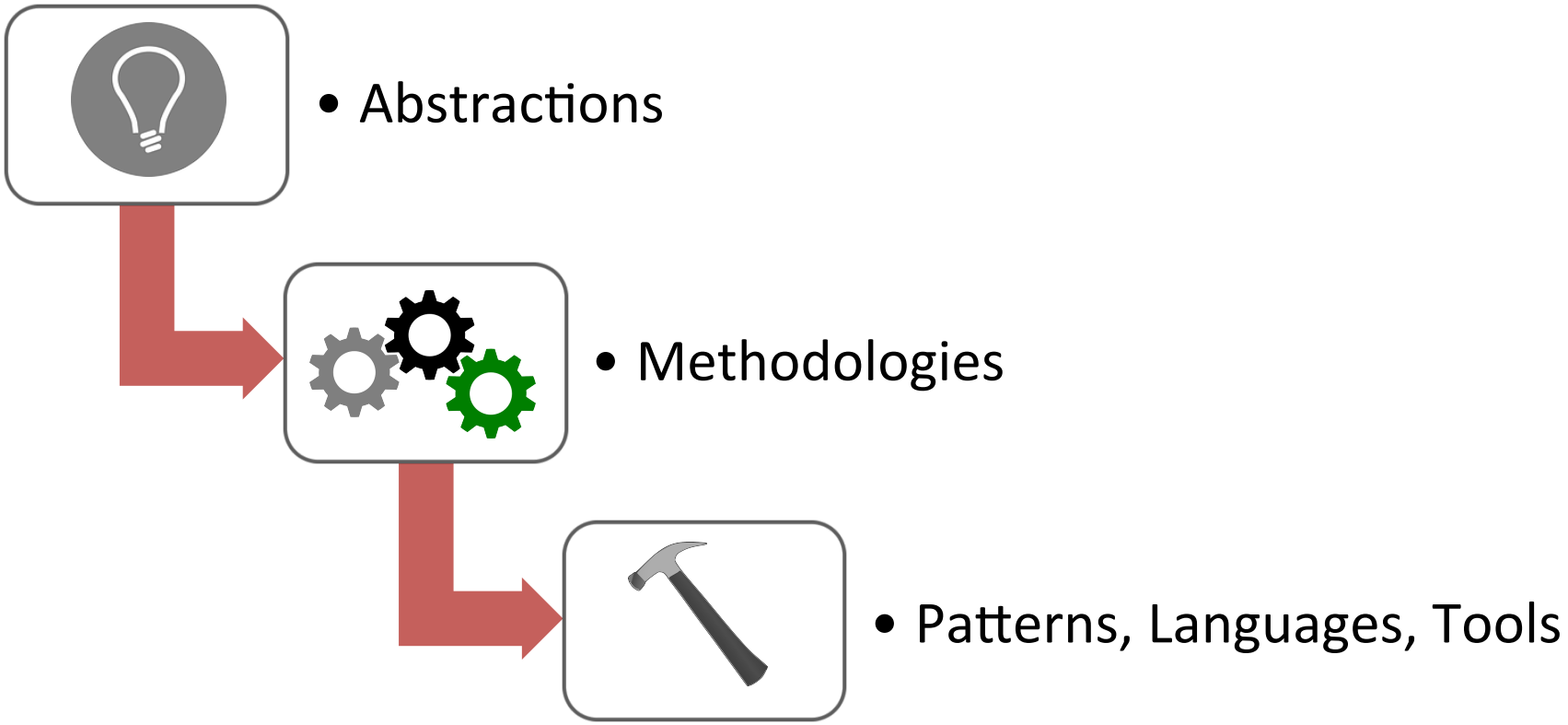
PyMTL

The Performance-Productivity Gap



SimJIT

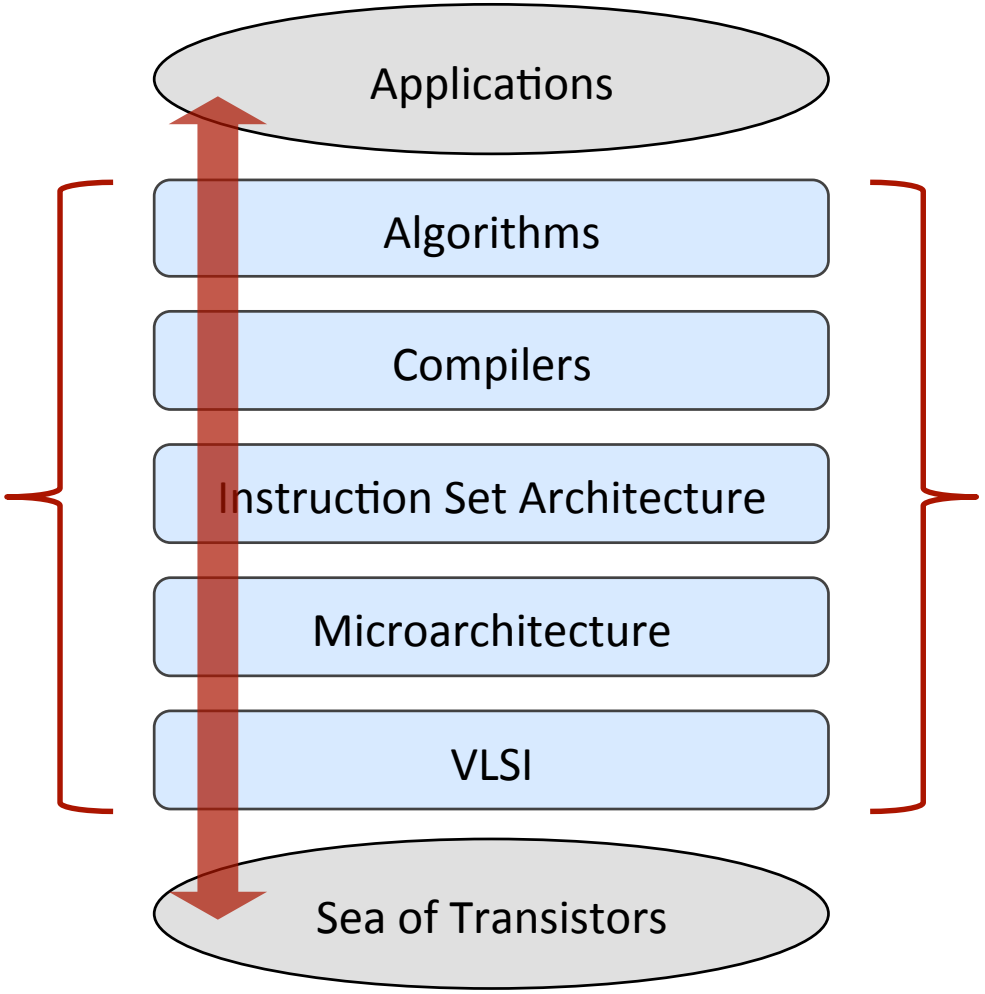
Managing Increasing Design Complexity



Computer Architecture Research Abstractions

Industry Development

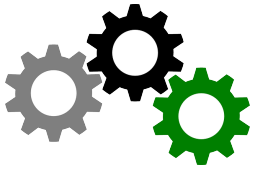
Hundreds of Engineers



Academic Research

A Few Researchers

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

VLSI

Sea of Transistors

Functional Level

- Behavior

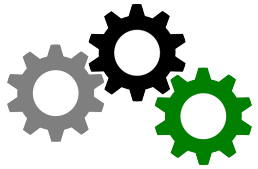
Cycle Level

- Behavior
- Timing

Register Transfer Level

- Behavior
- Timing
- Physical Resources

Computer Architecture Research Methodologies



Greater Simulation Speed

Modeling Towards Layout

Greater Model Detail

Functional Level

Algorithm and ISA Development

Cycle Level

Design Space Exploration

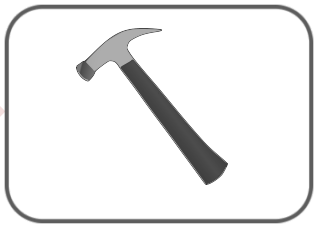
Register Transfer Level

Area/Energy/Timing Validation and Prototype Development

Computer Architecture Research Frameworks

• Abstractions

• Methodologies



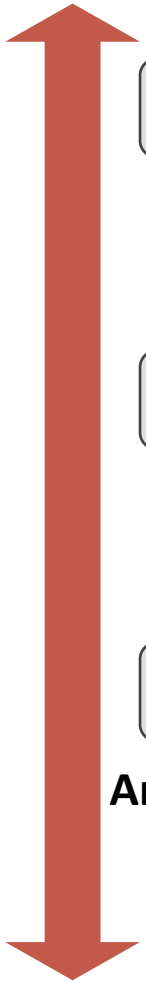
• Patterns, Languages, Tools

Computer Architecture Research Frameworks

MATLAB/Python Algorithm or
C++ Instruction Set Simulator

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

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



Functional Level

Algorithm and ISA Development

Cycle Level

Design Space Exploration

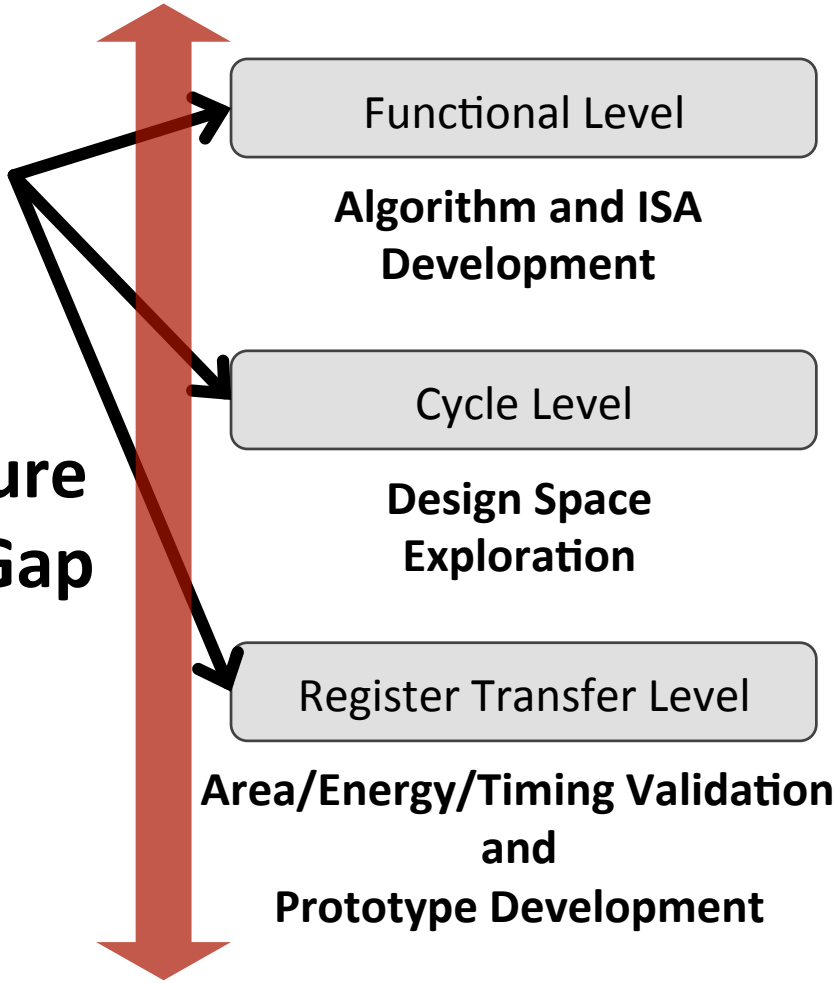
Register Transfer Level

Area/Energy/Timing Validation and Prototype Development

Computer Architecture Research Frameworks

Different languages, patterns, and tools!

The Computer Architecture Research Methodology Gap

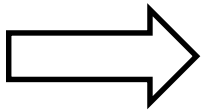


Great Ideas From Prior Work

• **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

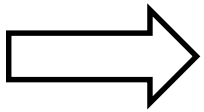
PyMTL Outline

The Computer Architecture Research Methodology Gap



PyMTL

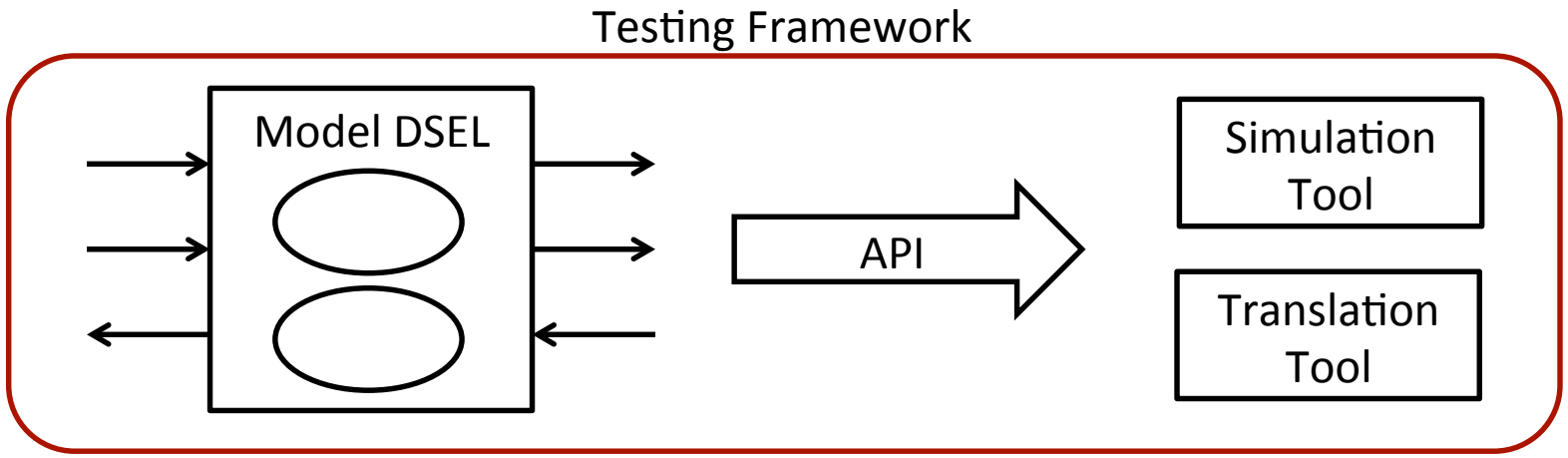
The Performance-Productivity Gap



SimJIT

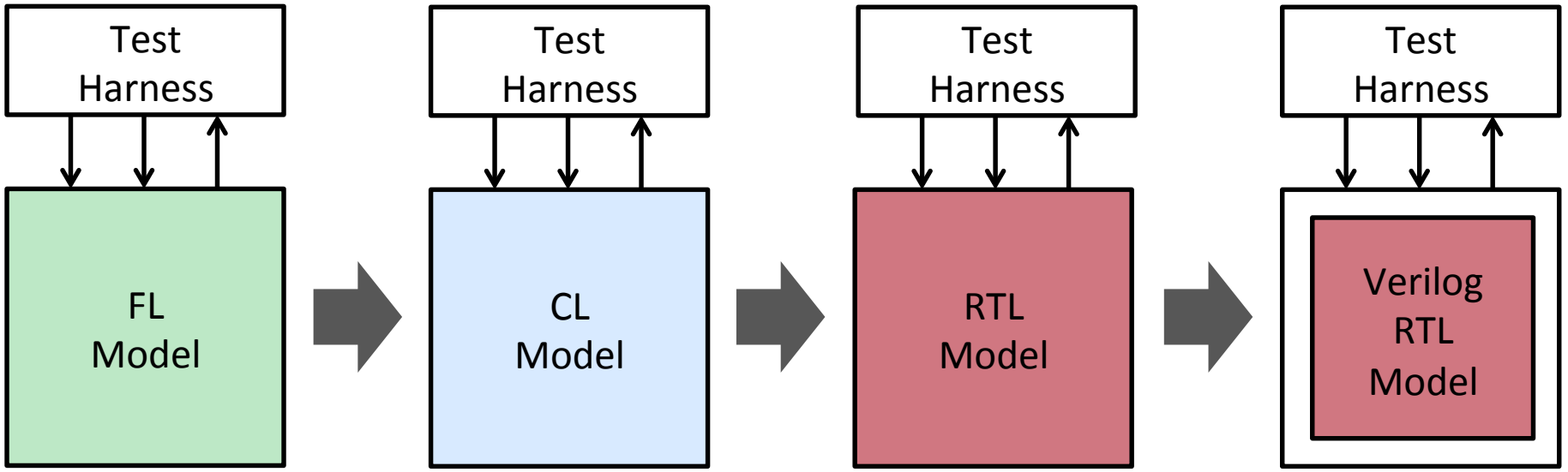
What is PyMTL?

- 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



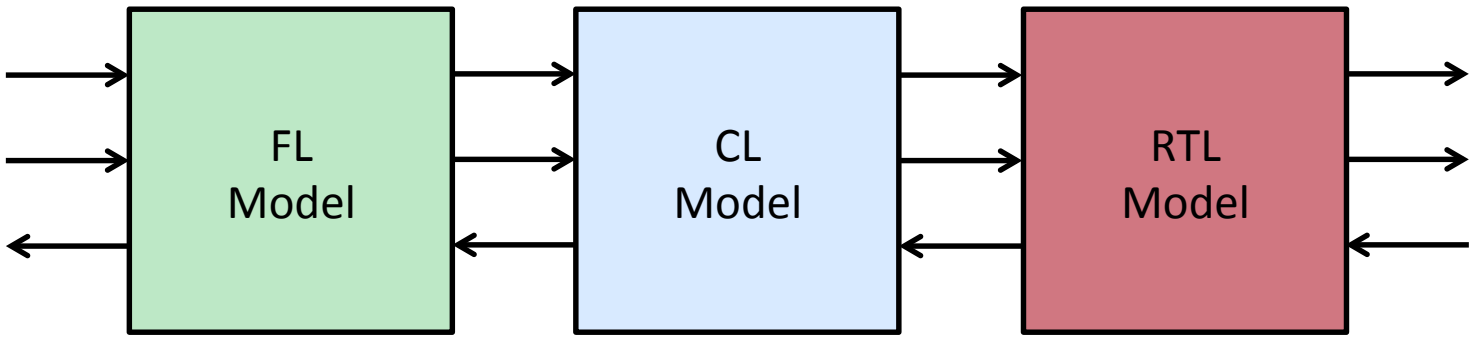
What Does PyMTL Enable?

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



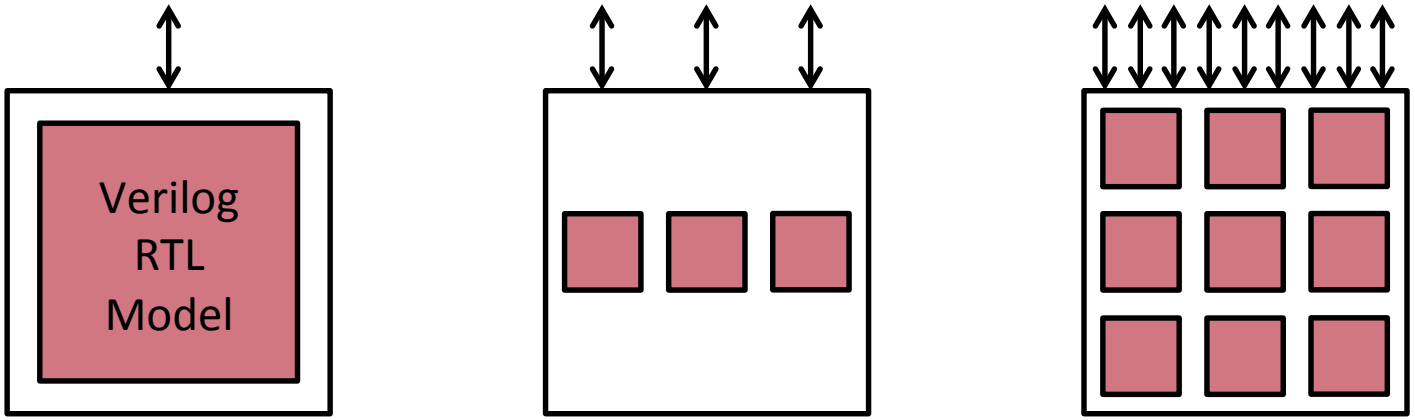
What Does PyMTL Enable?

- 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?

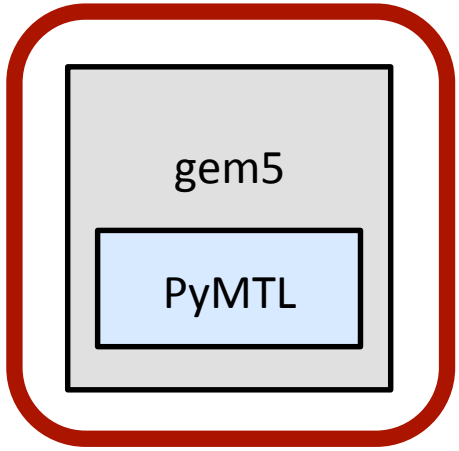
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators

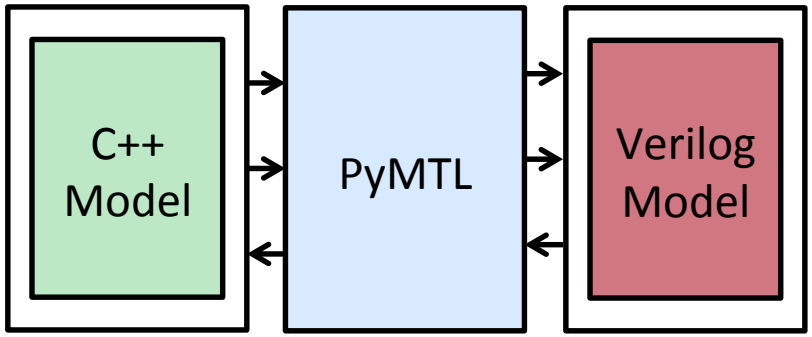


What Does PyMTL Enable?

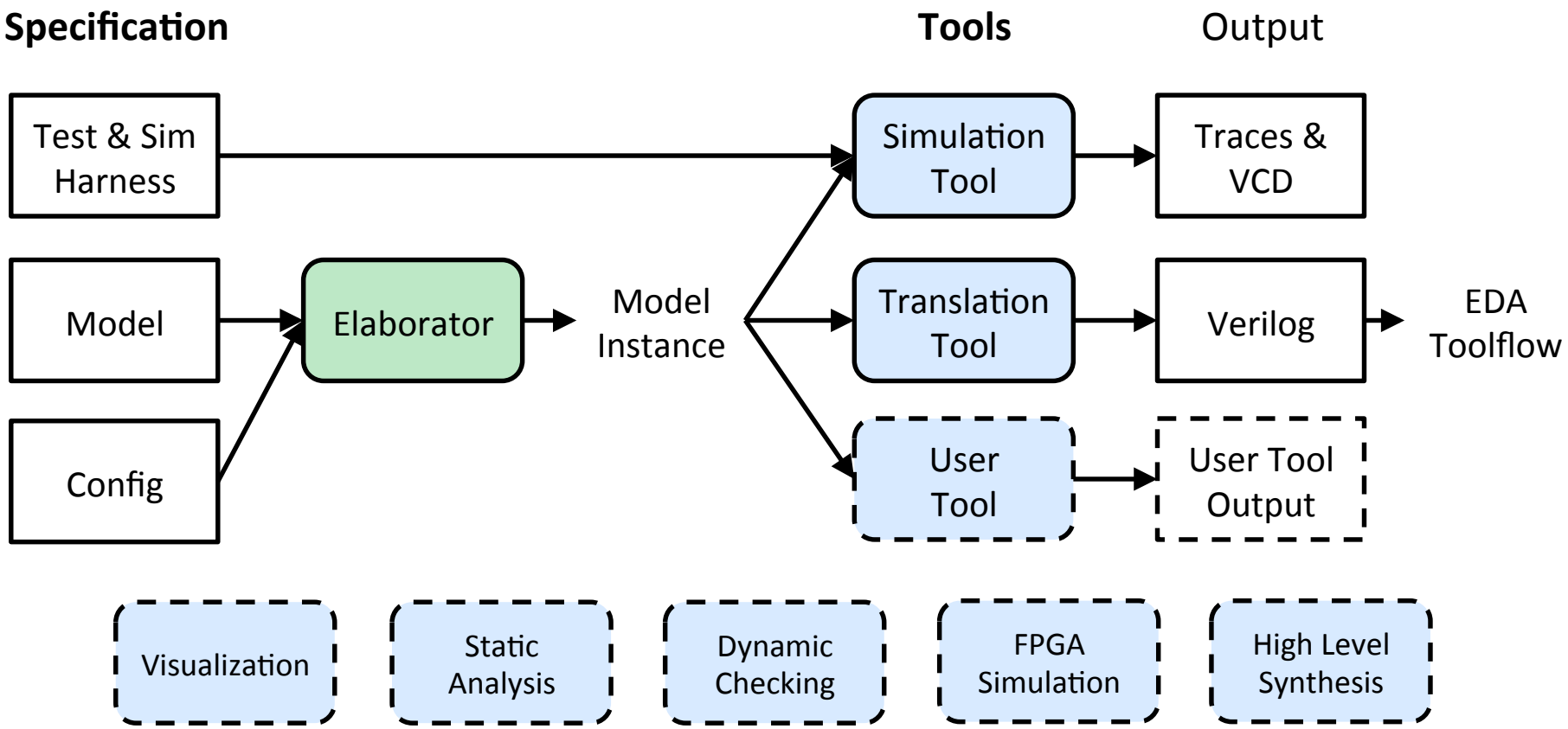
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog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

(Used to implement CL model for XLOOPS LPSU)





The PyMTL Framework

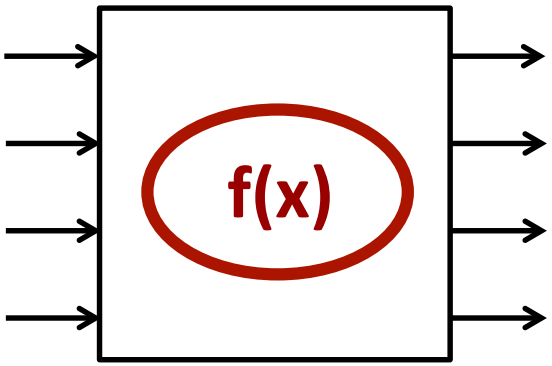


The PyMTL DSEL: FL Models

```
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)
```



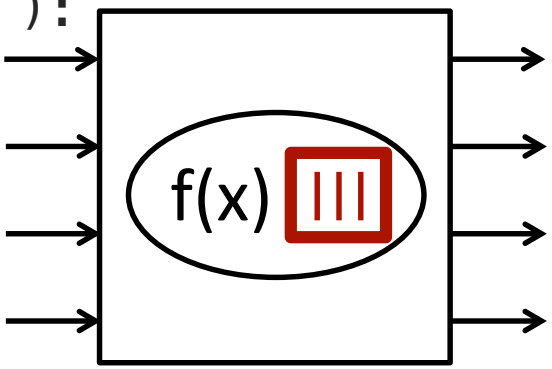

```
    @s.tick_fl
    def logic():
      for i, v in enumerate( sorted( s.in_ ) ):
        s.out[i].next = v
```

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
```

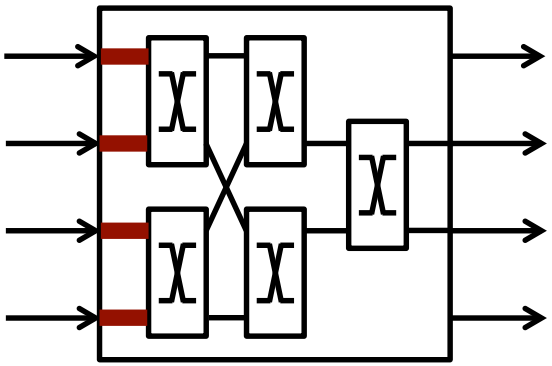


The PyMTL DSEL: RTL Models

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

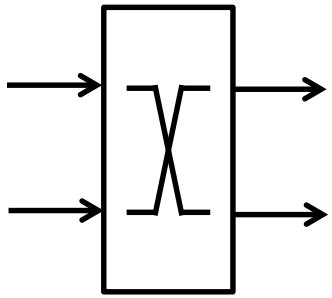
[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
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] )

    . . .
```

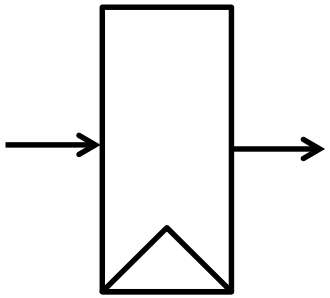


The PyMTL DSEL: RTL Models

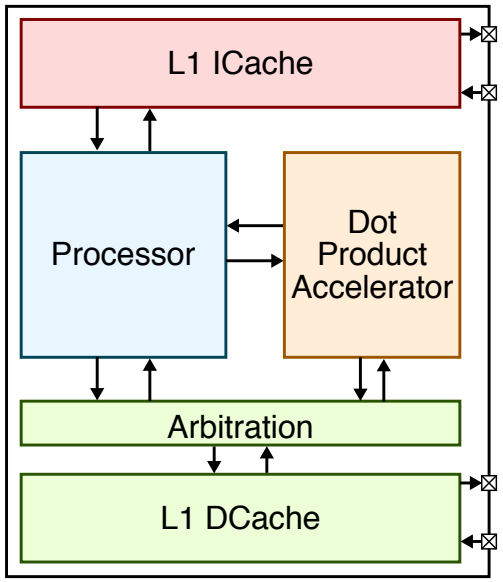
```
class MinMaxRTL( Model )
  def __init__( s, nbits ):
    s.in_ = InPort [2](nbits)
    s.out = OutPort[2](nbits)
    @s.combinational
    def logic():
      swap = s.in_[0] > s.in_[1]
      s.out[0].value = s.in[1] if swap else s.in[0]
      s.out[1].value = s.in[0] if swap else s.in[1]
```

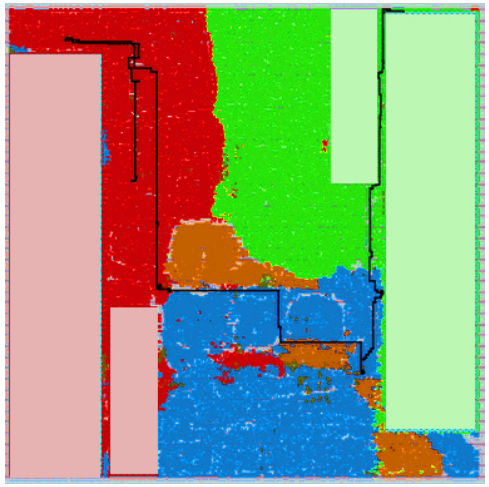


```
class RegRTL( Model )
  def __init__( s, nbits ):
    s.in_ = InPort (nbits)
    s.out = OutPort(nbits)
    @s.tick_rtl
    def logic():
      s.out.next = s.in_
```

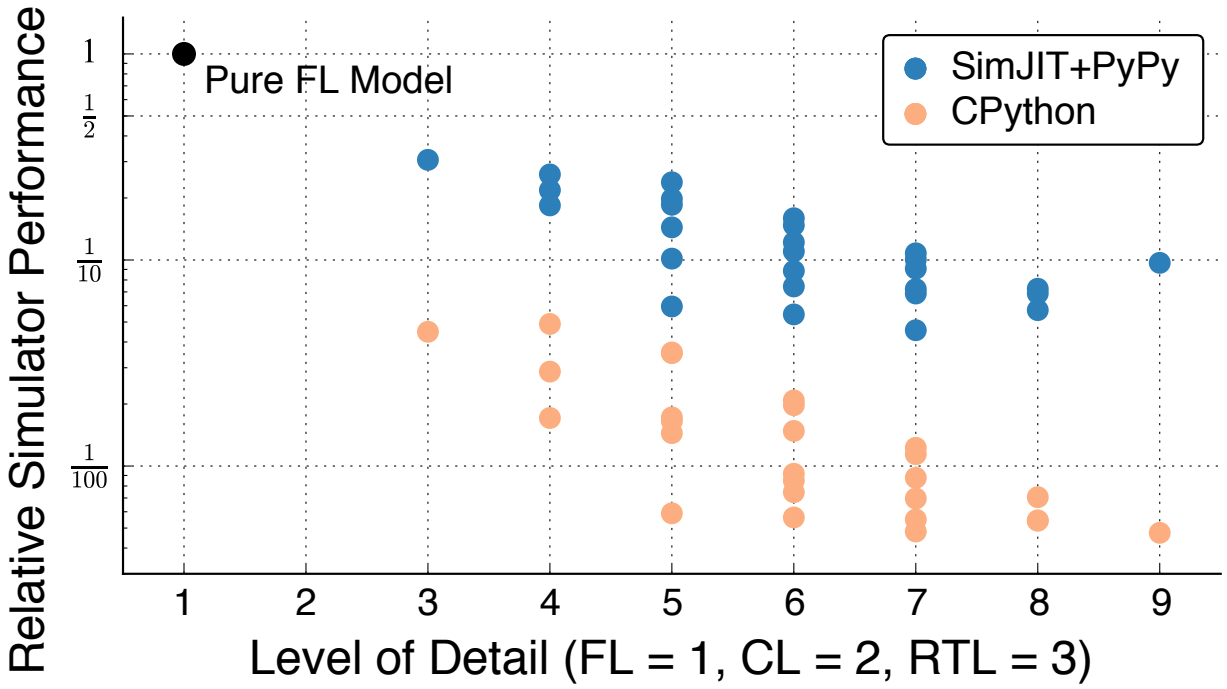


PyMTL Accelerator Case Study





- 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



Why Python?

Benefits:

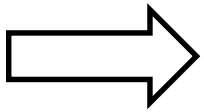
- Modern language features enable rapid prototyping (dynamic-typing, reflection, metaprogramming)
- Lightweight, pseudocode-like syntax
- Built-in support for integrating C/C++ code
- Large, active developer and support community

Drawbacks:

- Performance

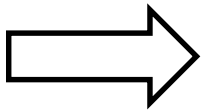
PyMTL Outline

The Computer Architecture Research Methodology Gap



PyMTL

The Performance-Productivity Gap



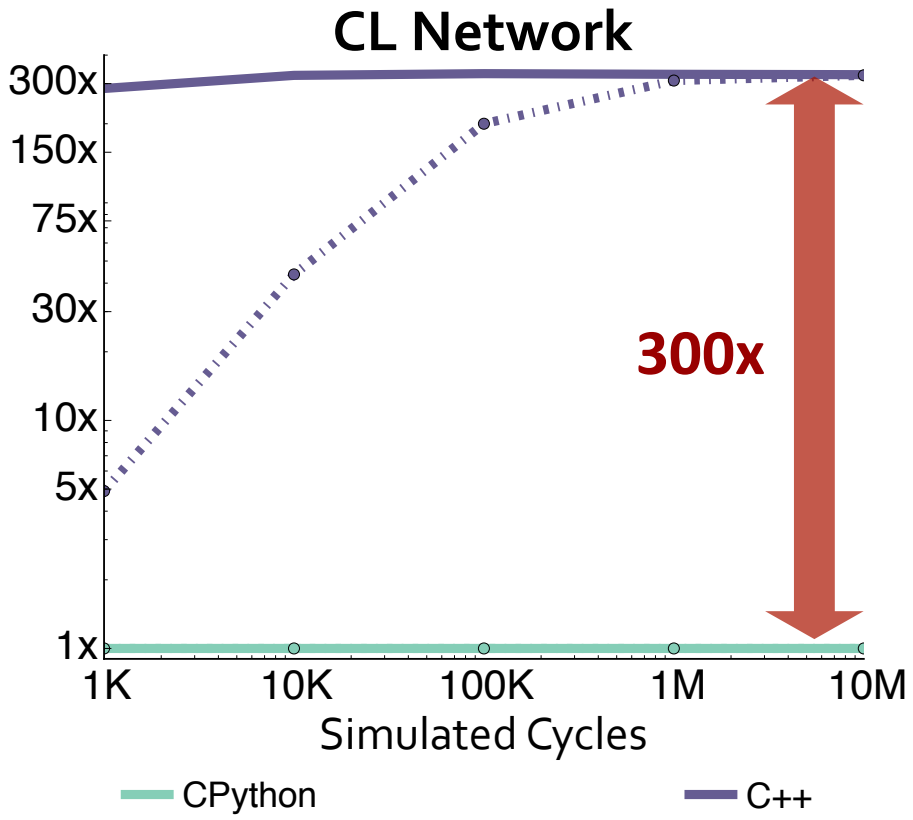
SimJIT

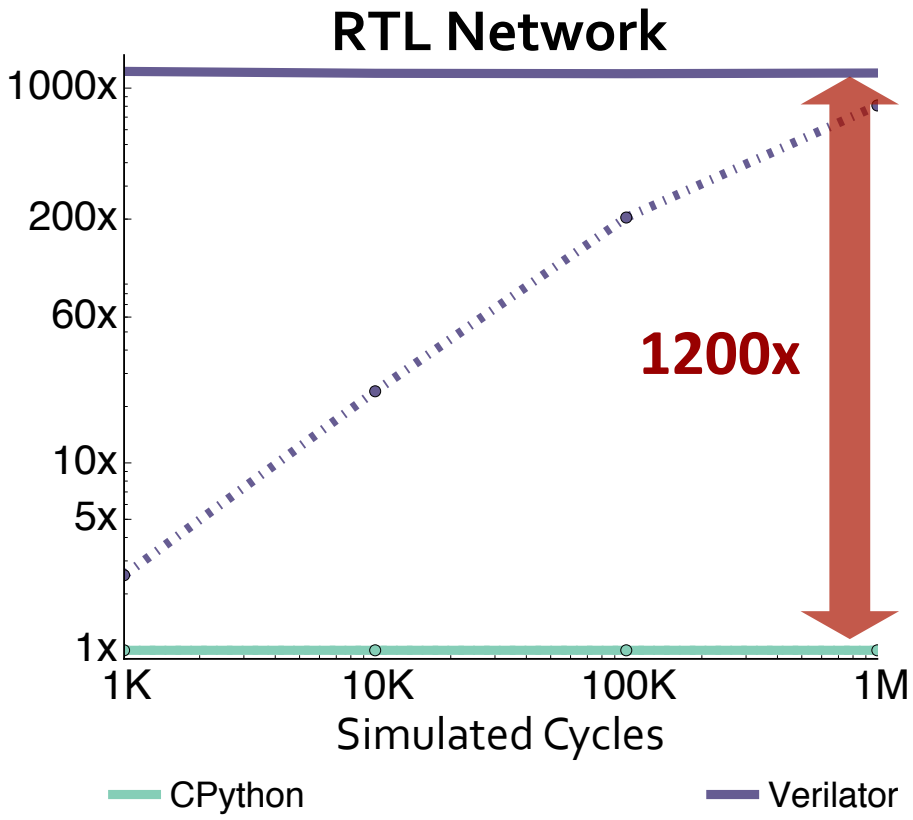
Performance-Productivity Gap

Experiment:

- Simple 8x8 Mesh Network Model
- Cycle-Precise CL Model:
  - PyMTL Model Simulated with the CPython Interpreter
  - Hand-Written C++ Model and Simulator
- Bit-Accurate RTL Model:
  - PyMTL Model Simulated with CPython Interpreter
  - Hand-Written Verilog RTL Simulated with Verilator

Performance-Productivity Gap



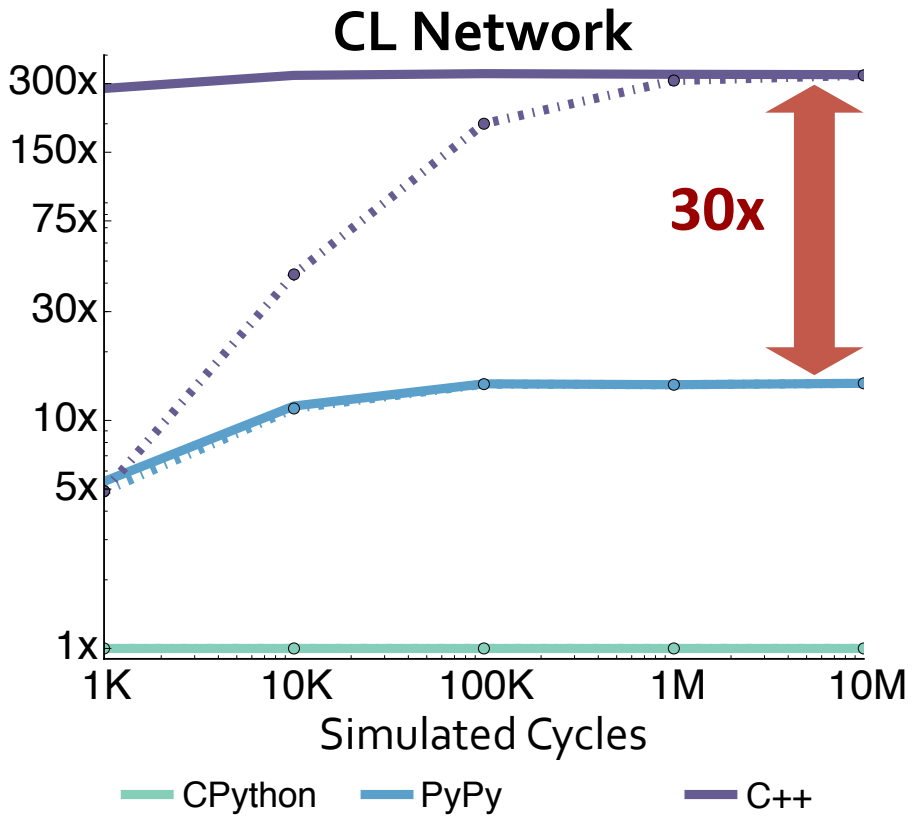


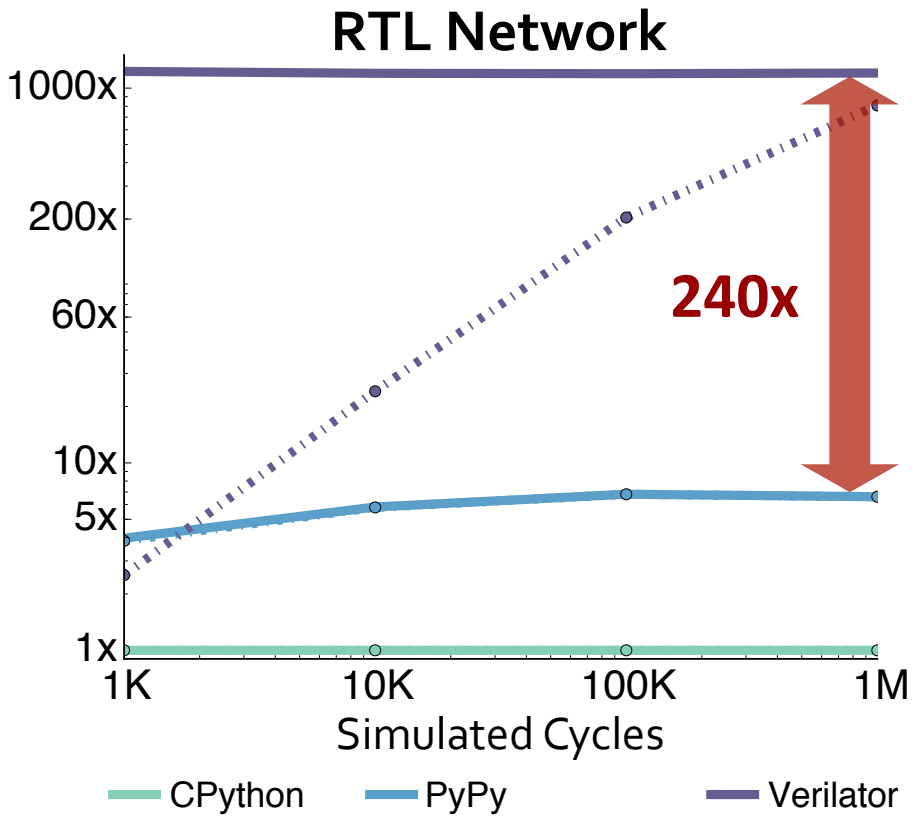
Performance-Productivity Gap

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

- **Python-Wrapped C/C++ Libraries**
  (NumPy, CVXOPT, NLPy, pythonOCC, GEM5)
- **Numerical Just-In-Time Compilers**
  (Numba, Parakeet)
- **Just-In-Time Compiled Interpreters**
  (PyPy, Pyston)
- **Selective Embedded Just-In-Time Specialization**
  (SEJITS)

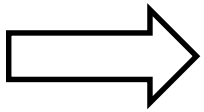
Performance-Productivity Gap





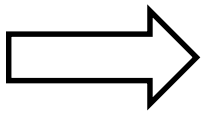
PyMTL Outline

The Computer Architecture Research Methodology Gap



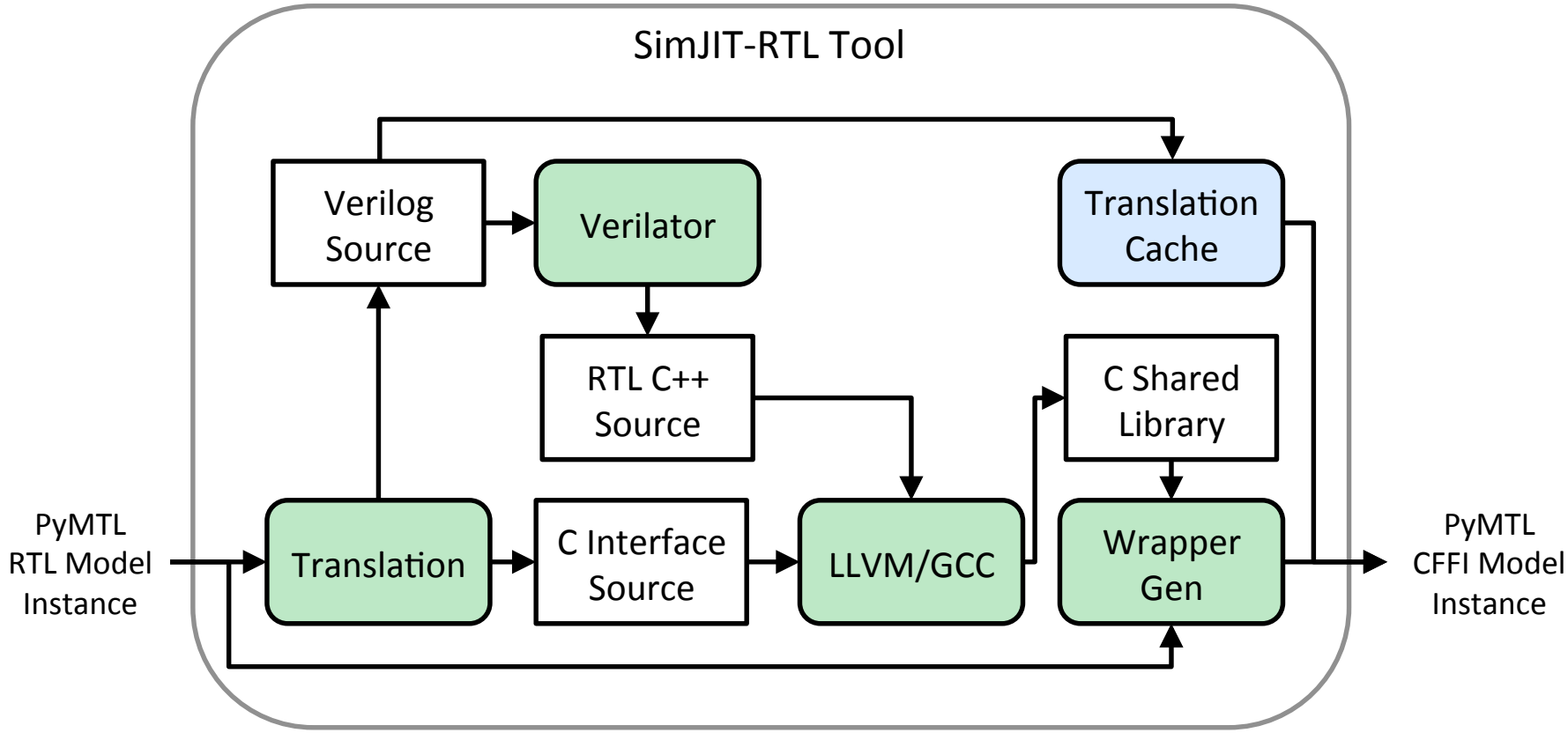
PyMTL

The Performance-Productivity Gap



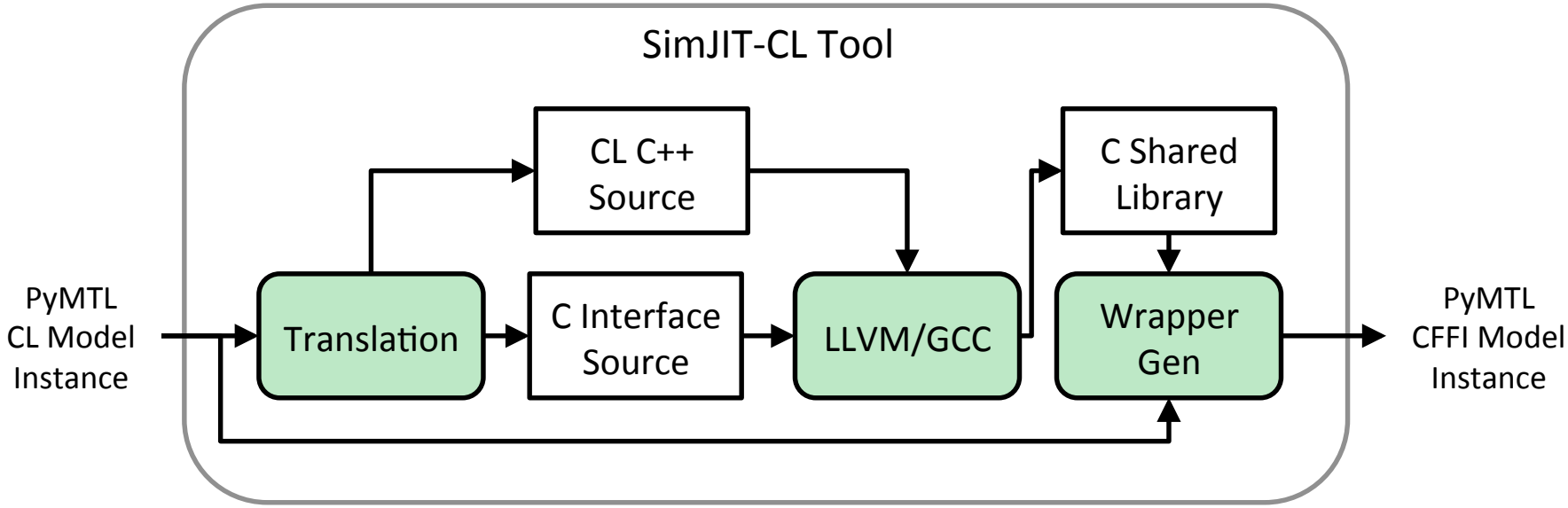
SimJIT

PyMTL SimJIT-RTL Architecture



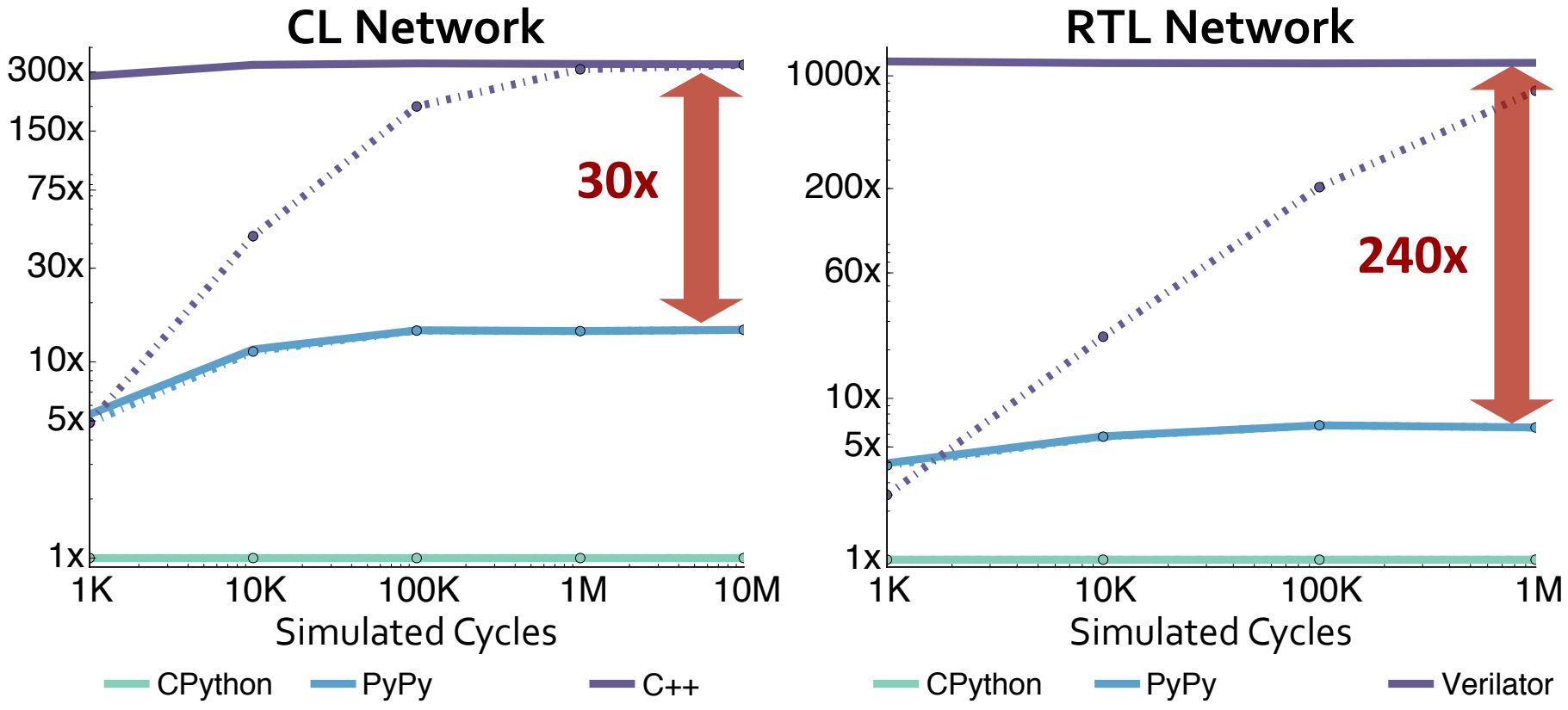
Fairly robust, ready for use in research!

PyMTL SimJIT-CL Architecture

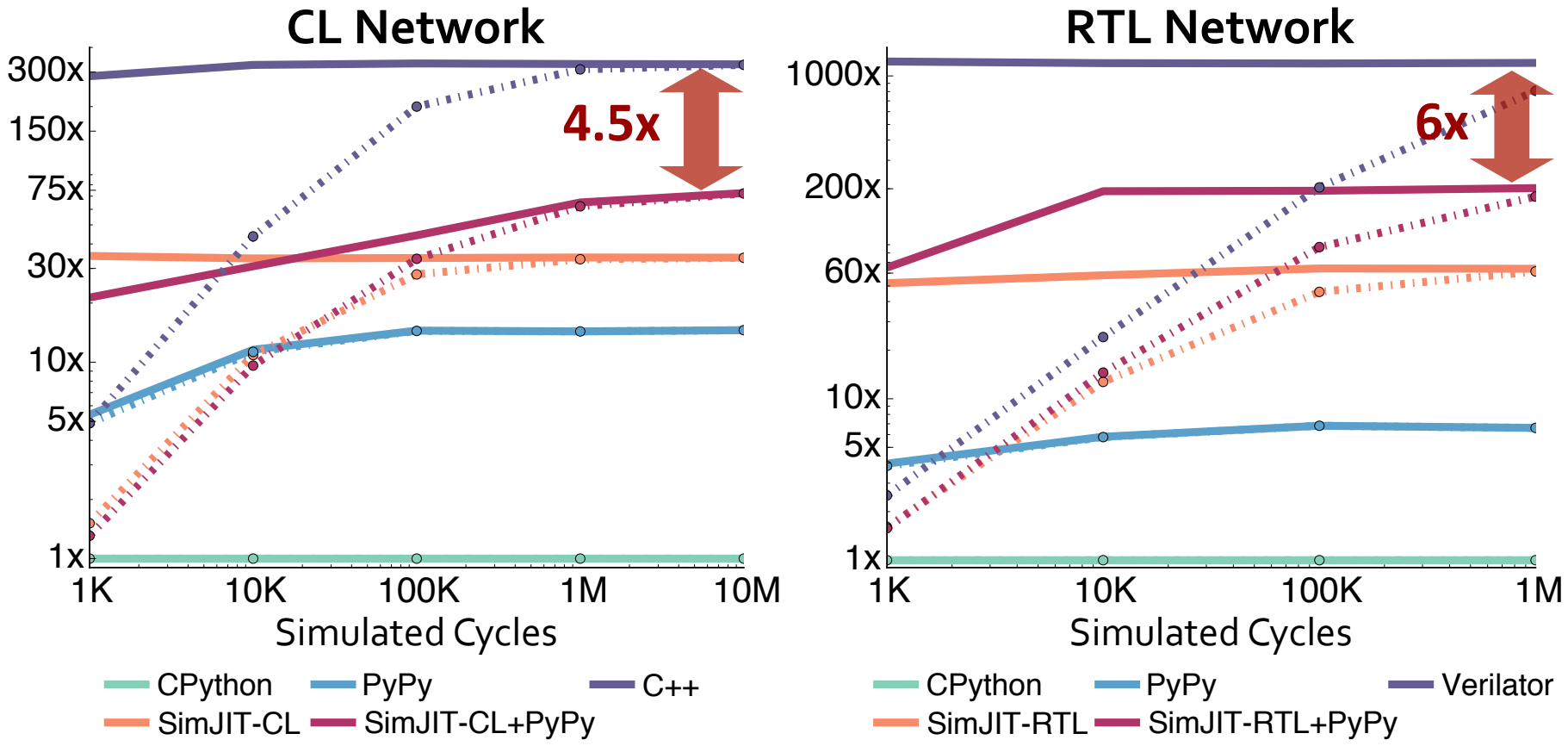


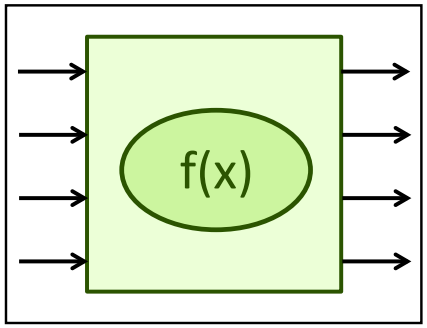
Just a prototype!

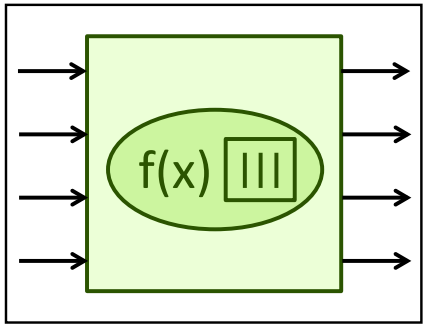
Performance-Productivity Gap

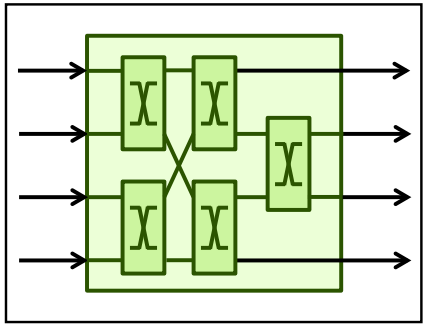


PyMTL SimJIT Performance







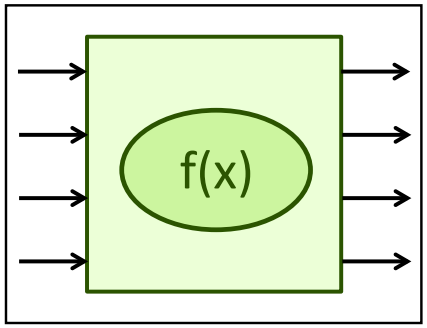


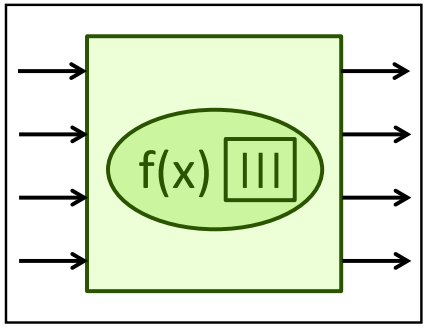
PyMTL Take-Away Points

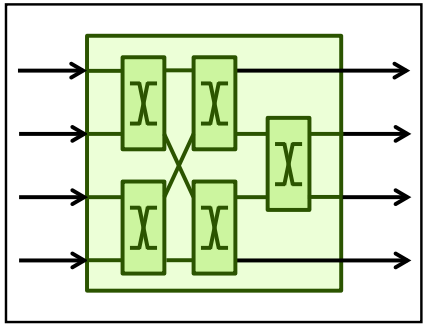
- PyMTL is a productive Python framework for FL, CL, and RTL modeling and hardware design
- SimJIT is a strong first step towards closing the performance-productivity gap between Python and C++ simulation
- An alpha version of PyMTL is open source and available for researchers to experiment with

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

This work was supported in part by the National Science Foundation (NSF), the Defense Advanced Research Projects Agency (DARPA), and donations from Intel Corporation and Synopsys, Inc.

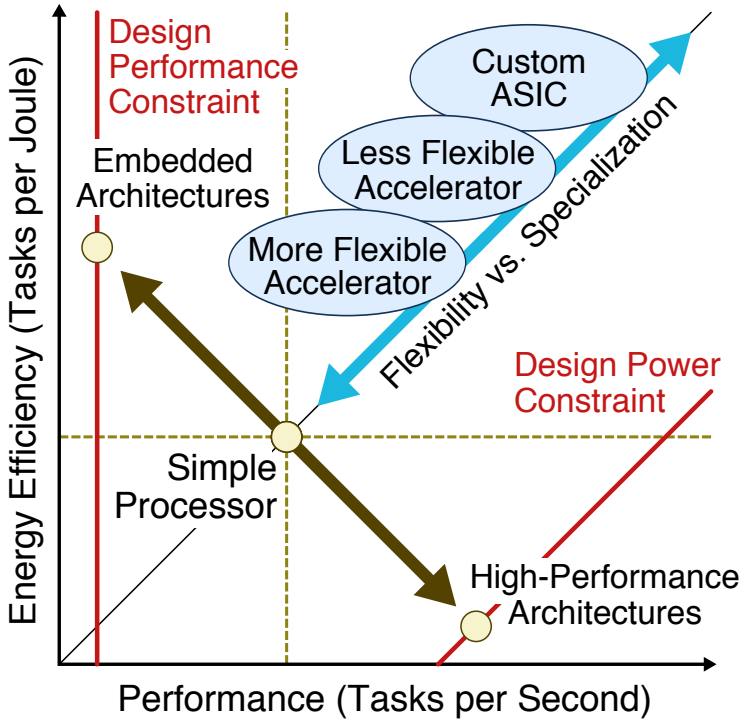






Batten Research Group

Exploring cross-layer hardware specialization using a vertically integrated research methodology




```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 addiu.xi  rA, 4
 addiu.xi  rB, 4
 addiu     r1, r1, 1
 xloop.uc  r1, rN, loop
```

```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#pragma xloops atomic
for(i = 0; i < N; i++)
  B[ A[i] ]++;
  D[ C[i] ]++;
```