Architectural Specialization for Inter-Iteration Loop Dependence Patterns

Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

Fall 2015
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

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

- XLOOPS ISA
- XLOOPS Compiler
- XLOOPS uArch
- XLOOPS Evaluation
- PyMTL
- Pydgin

Design Performance Constraint

Embedded Architectures

Custom ASIC

Less Flexible Accelerator

More Flexible Accelerator

Flexibility vs. Specialization

Design Power Constraint

Energy Efficiency (Tasks per Joule)

Performance (Tasks per Second)

Simple Processor

High-Performance Architectures

Cornell University Christopher Batten 3 / 53
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

- **GPGPU Architecture**
  - [ISCA'13]
  - [MICRO'14a] (AFOSR)

- **Integrated Voltage Regulation**
  - [MICRO'14b] (under review)

- **XLOOPS Explicit Loop Specialization**
  - [MICRO'14c] (DARPA, NSF)

- **Polymorphic Hardware Specialization**
  - (NSF)

- **Accelerating Dynamic Prog Langs**
  - (NSF)

- **PyMTL/Pydgin Frameworks**
  - [MICRO'14d]
  - [ISPASS'15] (under review)

- **Projects**

  - Apps
  - Algos
  - PL
  - Compiler
  - ISA
  - uArch
  - RTL
  - VLSI
  - Circuits
  - Tech
XLOOPS: Architectural Specialization for Inter-Iteration Loop Dependence Patterns

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

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 heterogeneous architecture with a new execution paradigm supporting traditional, specialized, and adaptive execution

![Diagram of XLOOPS architecture]

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

3. XLOOPS Microarchitecture

![XLOOPS Microarchitecture Diagram]

4. Evaluation

![Evaluation Graph]

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++)

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

3. **XLOOPS Microarchitecture**

![Microarchitecture Diagram](image)

4. **Evaluation**

![Evaluation Graph](image)
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

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

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

```c
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 guaranteed to have the same value as a serial execution
- Inspired by Multiscalar
XLOOPS Instruction Set: Ordered-Through-Memory

```plaintext
for (i=k; i<N; i++)

    # 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
Parallelize using xloop.uc.db

\[
\begin{align*}
\text{for ( } & i=0; \ i<N; \ i++ \ ) \\
\text{...} \\
\text{if ( cond ) N++;}
\end{align*}
\]
1. **XLOOPS Instruction Set**

```plaintext
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**

```plaintext
#pragma xloops ordered
for(i = 0; i < N i++)

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

3. **XLOOPS Microarchitecture**

![XLOOPS Microarchitecture Diagram](image)

4. **Evaluation**

![Evaluation Graph](image)
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] );
```
Research Overview  XLOOPS ISA  •  XLOOPS Compiler  •  XLOOPS uArch  XLOOPS Evaluation  PyMTL  Pydgin

- XLOOPS Compiler
- XLOOPS uArch
- XLOOPS Evaluation
- PyMTL
- Pydgin

- 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++)

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

3. XLOOPS Microarchitecture

![Diagram of XLOOPS Microarchitecture]

4. Evaluation

![Graph showing performance results]
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 – \texttt{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

GPP  LMU  Lane0  Lane1  LLFU

Scan Phase

Time

Specialized Execution Phase
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
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
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++)

#pragma xloops atomic
for(i = 0; i < N; 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
- Covariance 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
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 degradation
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
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
- XLOOPS is able to achieve higher performance compared to simple in-order cores and improved energy efficiency compared to complex out-of-order cores

Derek Lockhart, Gary Zibrat, Christopher Batten

47th ACM/IEEE Int’l Symp. on Microarchitecture (MICRO)

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, NJ, Mar. 2015
Computer Architecture Research Methodologies

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

- Functional-Level Modeling
  - Behavior

- Cycle-Level Modeling
  - Behavior
  - Cycle-Approximate
  - Analytical Area, Energy, Timing

- Register-Transfer-Level Modeling
  - Behavior
  - Cycle-Accurate Timing
  - Gate-Level Area, Energy, Timing
Computer Architecture Research Methodologies

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

Computer Architecture Research Methodology Gap
FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies
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
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

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

**Elaborator**
- Model Instance

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

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

**Visualization**
- Static Analysis
- Dynamic Checking
- FPGA Simulation
- High Level Synthesis

**EDA Toolflow**
The PyMTL Framework

But isn’t Python too slow?
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 Tool

Verilog Source → Verilator → RTL C++ Source

Translation

C Interface Source → LLVM/GCC

Translation Cache

C Shared Library → Wrapper Gen

PyMTL CFFI Model Instance

PyMTL RTL Model Instance
PyMTL Results: 64-Node Mesh Network

Simulation Time Including Compile Time

Simulation Time Excluding Compile Time

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 ASIC Tapeout

Layout generated from PyMTL for simple processor, L1 memory system, dot product xcel
Target Tech: 2x2mm IBM 130nm

Xilinx ZC706 FPGA development board for FPGA prototyping
Custom designed FMC mezzanine card for ASIC test chips

Derek Lockhart, Gary Zibrat, Christopher Batten
47th ACM/IEEE Int’l Symp. on Microarchitecture (MICRO)

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, NJ, Mar. 2015
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
Productivity \hspace{200pt} Performance

Architectural Description Language \hspace{20pt} [SimIt-ARM2006] \hspace{20pt} [Wagstaff2013] \hspace{20pt} Instruction Set Interpreter in C with DBT

[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

J.D’Errico and W.Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. DATE’06.
Productivity → [SimIt-ARM2006] [Wagstaff2013] → Performance

Architectural Description Language → [SimIt-ARM2006] [Wagstaff2013] → 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)
Productivity \hspace{2cm} \textbf{Performance}

- Architectural Description Language
- [SimIt-ARM2006] [Wagstaff2013]
- Instruction Set Interpreter in C with DBT

- Dynamic-Language Interpreter in RPython
- RPython Translation Toolchain
- Dynamic Language Interpreter in C with JIT Compiler

\textbf{Meta-Tracing JIT:}
\textit{makes JIT generation generic across languages}
Productivity ⟷ Performance

- Flexible, productive, pseudocode-like ADL syntax
- ADL embedded in a popular, general-purpose language
- Tracing-JIT generator applies across many different ISAs
- Leverages advancements from dynamic-language JIT research
Porting Pydgin to a new user-level ISA takes just a few weeks
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.
Derek Lockhart, Ji Kim, Shreesha Srinath, Christopher Torng, Berkin Ilbeyi, Moyang Wang, and many M.S./B.S. students

Prof. Zhiru Zhang, Mingxing Tan, Gai Liu

Equipment and Tool Donations
Intel, NVIDIA, Synopsys, Xilinx
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++)

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

Performance (Tasks per Second) vs. Design Power Constraint

Energy Efficiency (Tasks per Joule) vs. Flexibility vs. Specialization

- Simple Processor
- High-Performance Architectures
- Less Flexible Accelerator
- More Flexible Accelerator
- Custom ASIC

---

OoO GPP
Lane Manager
Lanes
Mem XBar
L1 Data Cache

PyMTL
Pydgin