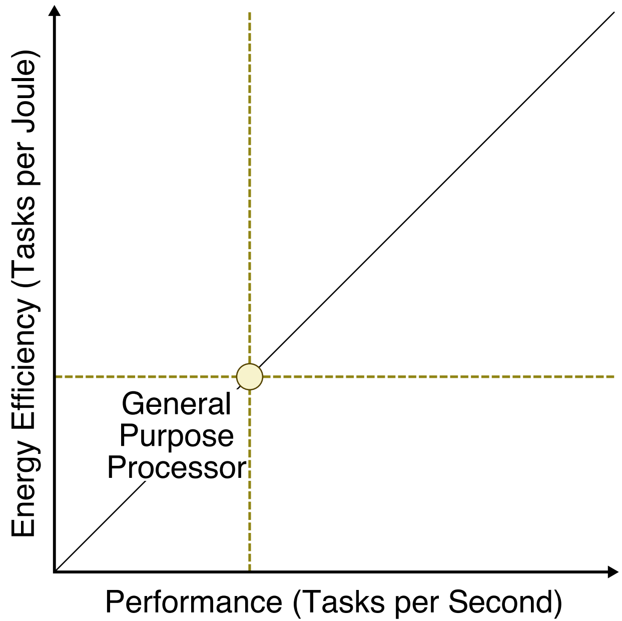
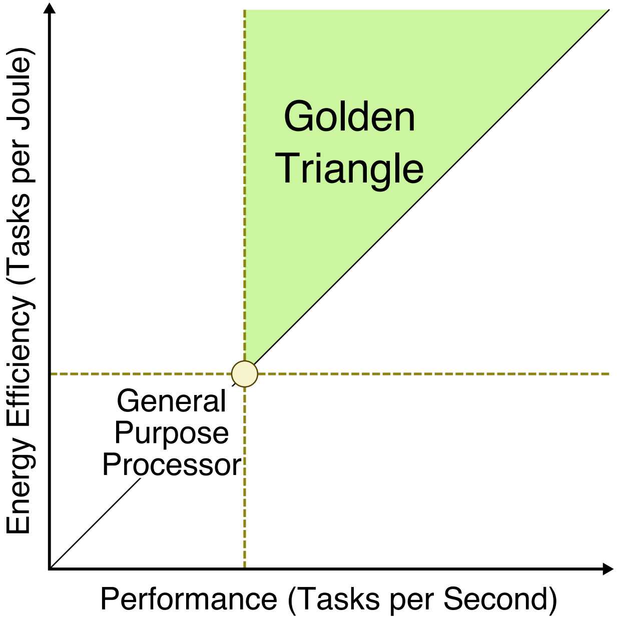
Architectural Specialization for Inter-Iteration Loop Dependence Patterns

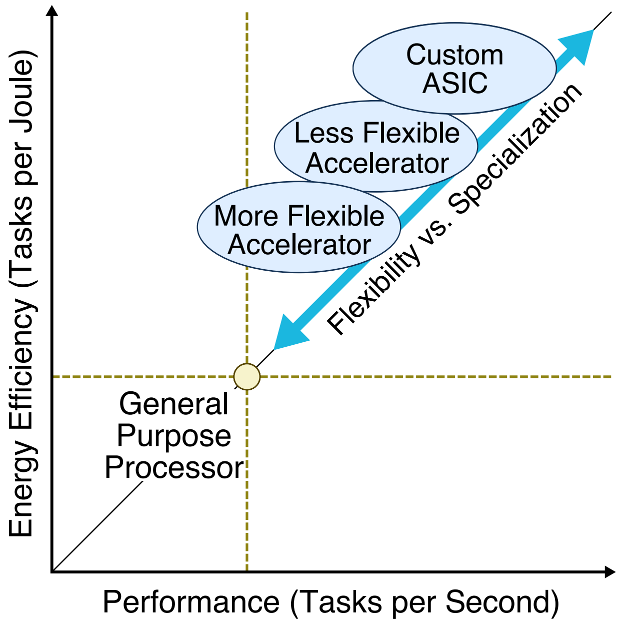
Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu
Zhiru Zhang, Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

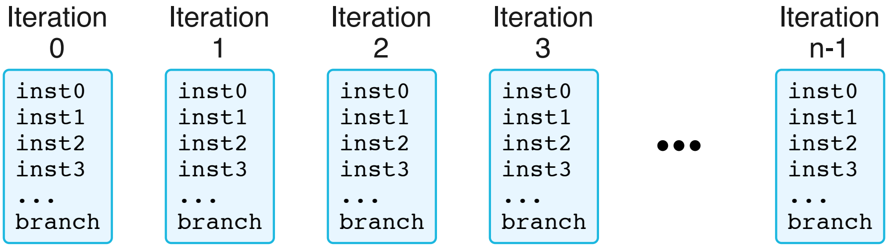
47th Int'l Symp. on Microarchitecture, Dec 2014


Energy Efficiency (Tasks per Joule)
General Purpose Processor
Performance (Tasks per Second)


Golden Triangle
General Purpose Processor
Energy Efficiency (Tasks per Joule)
Performance (Tasks per Second)

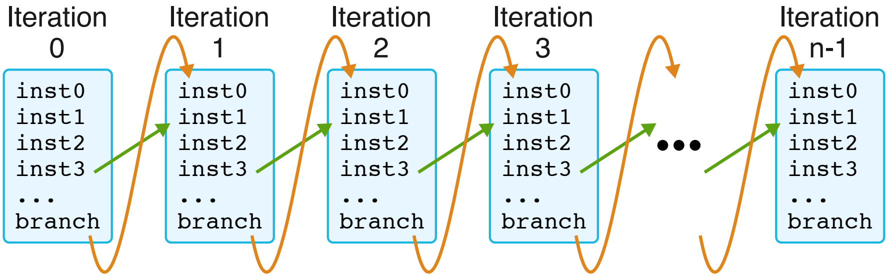

Custom ASIC
Less Flexible Accelerator
More Flexible Accelerator
Flexibility vs. Specialization
General Purpose Processor
Energy Efficiency (Tasks per Joule)
Performance (Tasks per Second)

Loop Dependence Pattern Specialization



Intra-Iteration
Micro-op Fusion,
ASIPs, CCA

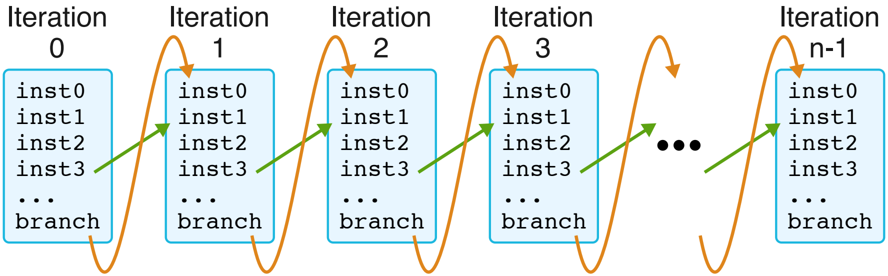
Loop Dependence Pattern Specialization



Intra-Iteration
Micro-op Fusion, ASIPs, CCA

Inter-Iteration
Vector, GPU, HELIX-RC

Loop Dependence Pattern Specialization

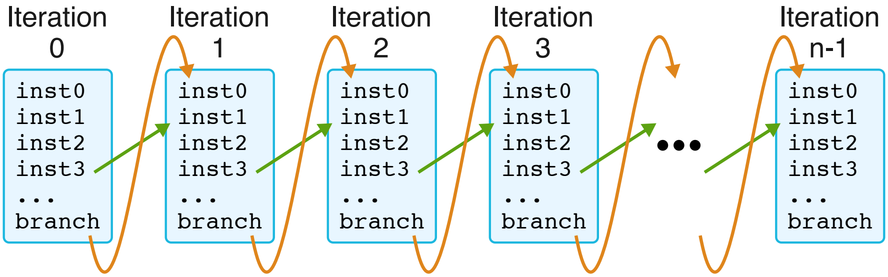


Intra-Iteration
Micro-op Fusion,
ASIPs, CCA

Inter-Iteration
Vector, GPU,
HELIX-RC

Both
DySER, Qs-Cores,
BERET

Loop Dependence Pattern Specialization



Intra-Iteration
Micro-op Fusion, ASIPs, CCA

Inter-Iteration
Vector, GPU, HELIX-RC

Both
DySER, Qs-Cores, BERET

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

Explicit Loop Specialization (XLOOPS)

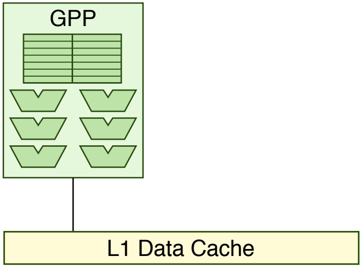
Key Idea 1: Expose fine-grained parallelism by elegantly encoding inter-iteration loop dependence patterns in the ISA

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

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

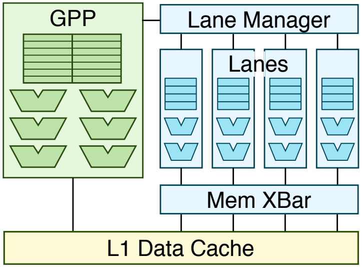


- **Traditional Execution**

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

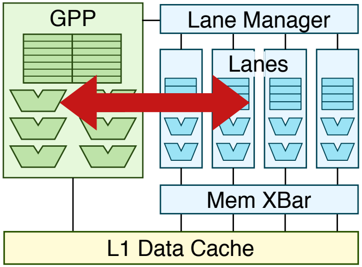


- **Traditional Execution**
- **Specialized Execution**

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 ISA

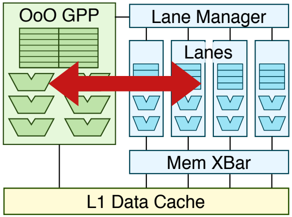
```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 ...
 addiu.xi rA, 4
 addiu.xi rB, 4
 addiu    r1, r1, 1
 xloop.uc r1, rN, loop
```

2. XLOOPS Compiler

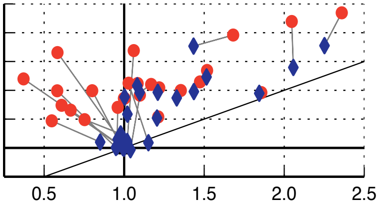
```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

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

3. XLOOPS Microarchitecture



4. Evaluation



1. XLOOPS ISA

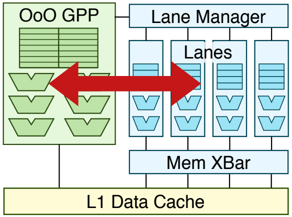
```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 ...
 addiu.xi   rA, 4
 addiu.xi   rB, 4
 addiu      r1, r1, 1
 xloop.uc   r1, rN, loop
```

2. XLOOPS Compiler

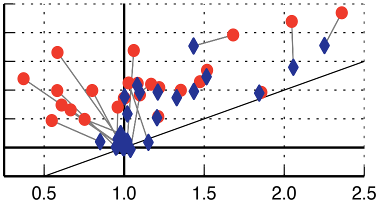
```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

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

3. XLOOPS Microarchitecture

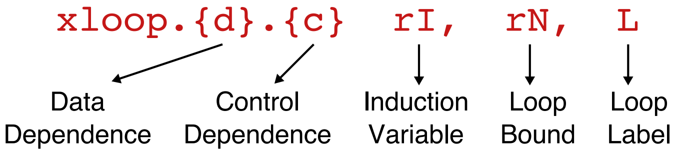


4. Evaluation



XLOOPS Instruction Set Extensions

XLOOP Instruction



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

XLOOPS Instruction Set Extensions

XLOOP Instruction

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

`{d}`	`{c}`	`rI`	`rN`	`L`
Data Dependence	Control Dependence	Induction Variable	Loop Bound	Loop Label

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

`uc`	`fb`
Unordered Concurrent	**Fixed Bound**

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 ISA: Unordered Concurrent

Element-wise Vector Multiplication

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

RISC ISA

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

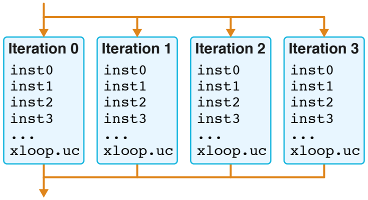
XLOOPS ISA: Unordered Concurrent

Element-wise Vector Multiplication

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

RISC ISA

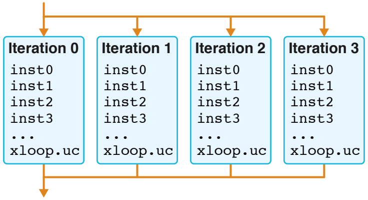
```
loop:
  lw       r2, 0(rA)
  lw       r3, 0(rB)
  mul      r4, r2, r3
  sw       r4, 0(rC)
  addiu    rA, rA, 4
  addiu    rB, rB, 4
  addiu    rC, rC, 4
  addiu    r1, r1, 1
  bne      r1, rN, loop
```



XLOOPS ISA: Unordered Concurrent

Element-wise Vector Multiplication

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



RISC ISA

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

→

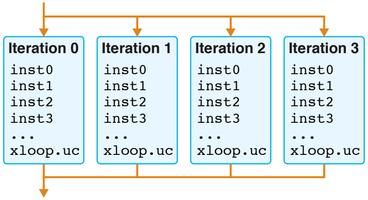
XLOOPS ISA

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

XLOOPS ISA: Unordered Concurrent

Element-wise Vector Multiplication

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



RISC ISA

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

→

XLOOPS ISA

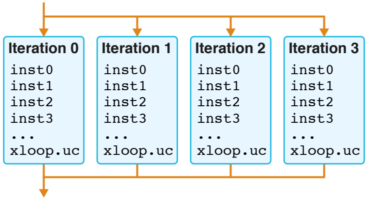
```
loop:
  lw        r2, 0(rA)
  lw        r3, 0(rB)
  mul       r4, r2, r3
  sw        r4, 0(rC)
  addiu.xi   rA, 4
  addiu.xi   rB, 4
  addiu.xi   rC, 4
  addiu     r1, r1, 1
  xloop.uc  r1, rN, loop
```

XLOOPS ISA: 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
```



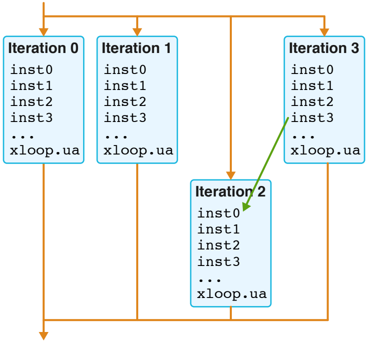
- Instructions in loop cannot write live-in registers
- Live-out values are stored in memory
- Data-races are possible

XLOOPS ISA: Unordered Atomic

Histogram Updates

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

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



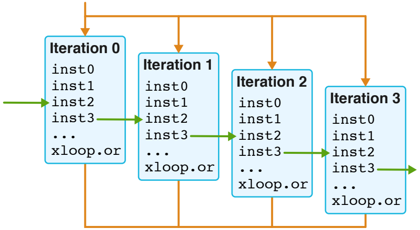
- Iterations execute atomically
- No race conditions
- Results can be non-deterministic
- Inspired by Transactional Memory

XLOOPS ISA: 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
```

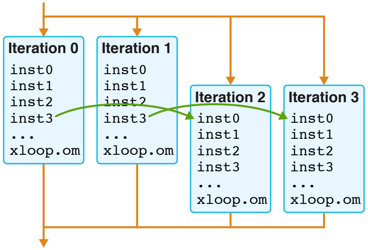


- rX - Cross Iteration Register
- CIRs are guranteed to have the same value as a serial execution
- Inspired by Multiscalar

XLOOPS ISA: Ordered-Through-Memory

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

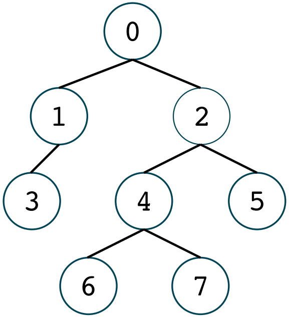
```
   # r1 = rK
   # r3 = rA + 4*rK
 loop:
   lw        r4, 0(r3)
   lw        r5, 0(rA)
   mul       r6, r4, r5
   sw        r6, 0(r3)
   addiu.xi  r3, 4
   addiu.xi  rA, 4
   addiu     r1, r1, 1
   xloop.om  r1, rN, loop
```



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

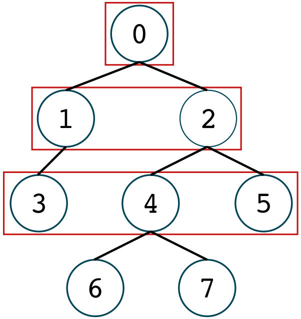
XLOOPS ISA: Dynamic Bound

- Recursive traversal



XLOOPS ISA: Dynamic Bound

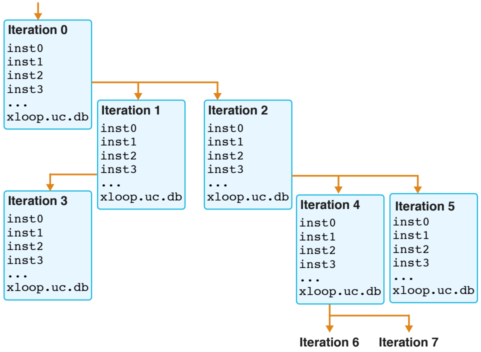
- Recursive traversal
- Parallelize across frontier using `xloop.uc`

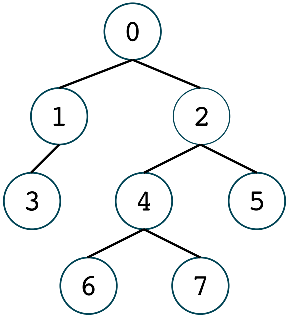


XLOOPS ISA: Dynamic Bound

- Parallelize using `xloop.uc.db`

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





1. XLOOPS ISA

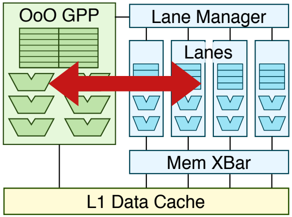
```
loop:
 lw         r2, 0(rA)
 lw         r3, 0(rB)
 ...
 ...
 addiu.xi rA, 4
 addiu.xi rB, 4
 addiu    r1, r1, 1
 xloop.uc r1, rN, loop
```

2. XLOOPS Compiler

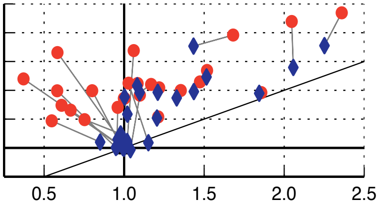
```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

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

3. XLOOPS Microarchitecture



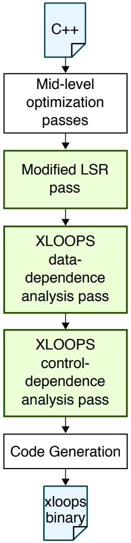
4. Evaluation

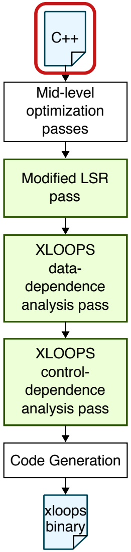


XLOOPS Compiler

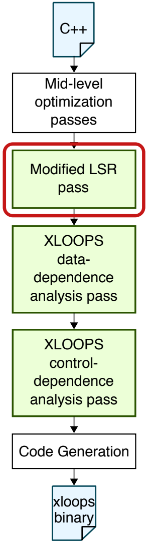
Kernel implementing Floyd-Warshall shortest path algorithm

```
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] );
```

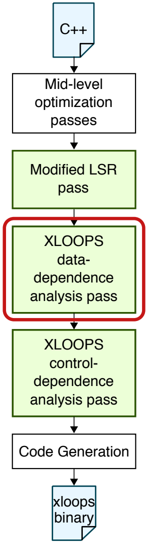

C++
Mid-level optimization passes
Modified LSR pass
XLOOPS data-dependence analysis pass
XLOOPS control-dependence analysis pass
Code Generation
xloops binary


C++
Mid-level optimization passes
Modified LSR pass
XLOOPS data-dependence analysis pass
XLOOPS control-dependence analysis pass
Code Generation
xloops binary

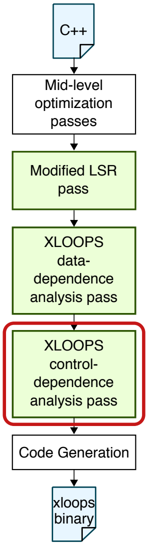
- Programmer annotations
  - `unordered`: no data-dependences
  - `ordered`: preserve data-dependences
  - `atomic`: atomic memory updates


C++
Mid-level optimization passes
Modified LSR pass
XLOOPS data-dependence analysis pass
XLOOPS control-dependence analysis pass
Code Generation
xloops binary

- Programmer annotations
  - `unordered`: no data-dependences
  - `ordered`: preserve data-dependences
  - `atomic`: atomic memory updates
- Loop strength reduction pass encodes MIVs as `xi` instructions


C++
Mid-level optimization passes
Modified LSR pass
XLOOPS data-dependence analysis pass
XLOOPS control-dependence analysis pass
Code Generation
xloops binary

- 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


C++
Mid-level optimization passes
Modified LSR pass
XLOOPS data-dependence analysis pass
XLOOPS control-dependence analysis pass
Code Generation
xloops binary

- 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 ISA

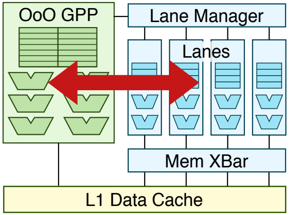
```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 ...
 addiu.xi rA, 4
 addiu.xi rB, 4
 addiu    r1, r1, 1
 xloop.uc r1, rN, loop
```

2. XLOOPS Compiler

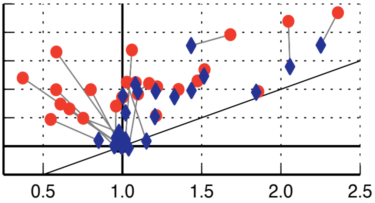
```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

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

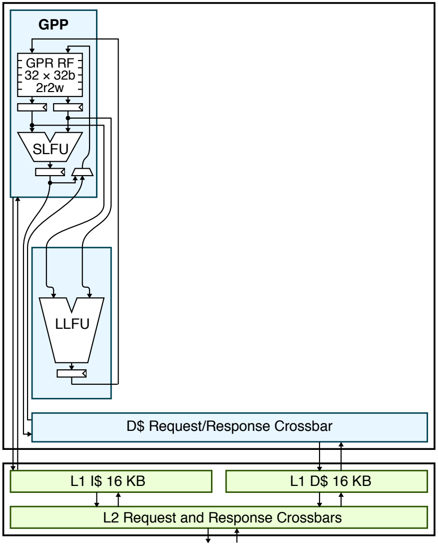
3. XLOOPS Microarchitecture



4. Evaluation



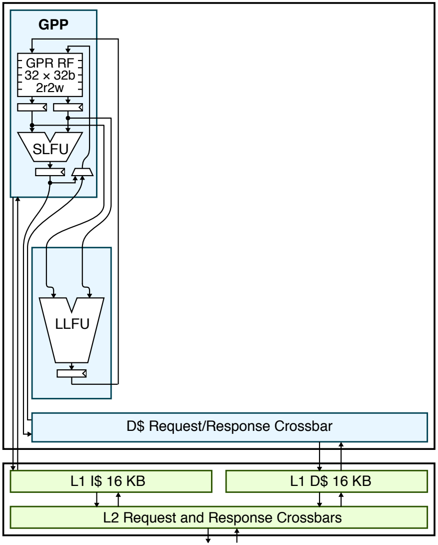
Traditional Execution


GPP
GPR RF
32 × 32b
2r2w
SLFU
LLFU
D$ Request/Response Crossbar
L1 I$ 16 KB
L1 D$ 16 KB
L2 Request and Response Crossbars

Minimal changes to a general-purpose processor (GPP)

- `xloop` $\rightarrow$ `bne`
- `addiu.xi` $\rightarrow$ `addiu`
- `addu.xi` $\rightarrow$ `addu`

Traditional Execution


GPP
GPR RF
32 × 32b
2r2w
SLFU
LLFU
D$ Request/Response Crossbar
L1 I$ 16 KB
L1 D$ 16 KB
L2 Request and Response Crossbars

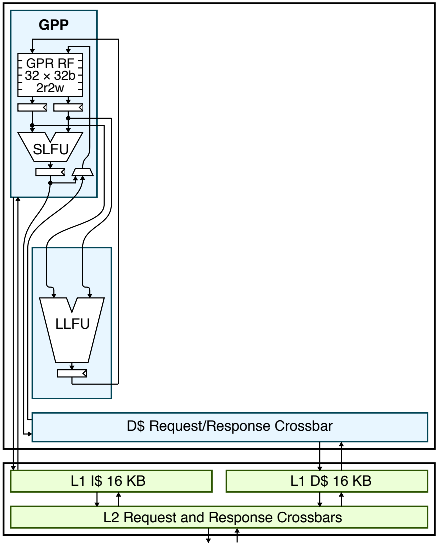
Minimal changes to a general-purpose processor (GPP)

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

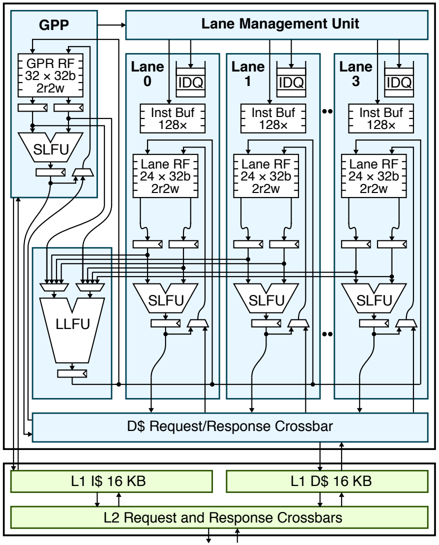
Efficient traditional execution

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

Specialized Execution – `xloop.uc`


GPP
GPR RF
32 × 32b
2r2w
SLFU
LLFU
D$ Request/Response Crossbar
L1 I$ 16 KB
L1 D$ 16 KB
L2 Request and Response Crossbars

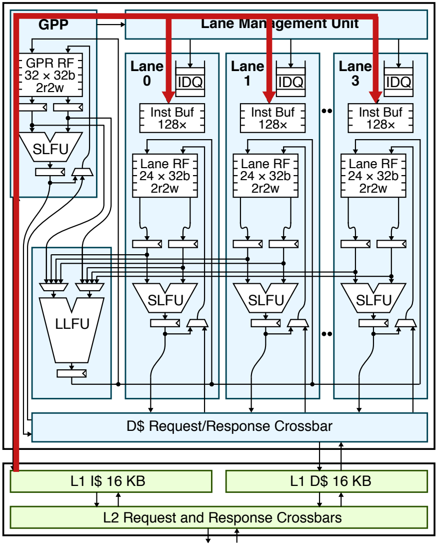
Specialized Execution – `xloop.uc`



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 – `xloop.uc`



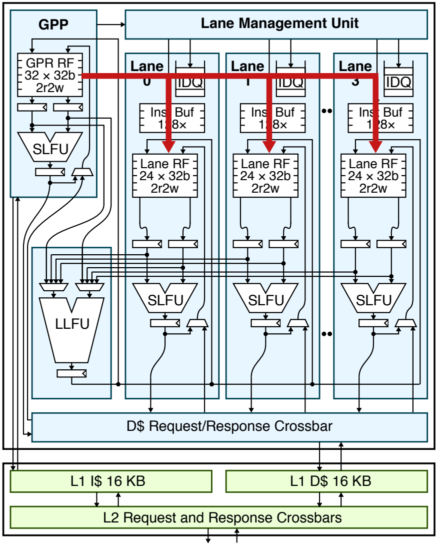
Loop Pattern Specialization Unit

- 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 – `xloop.uc`



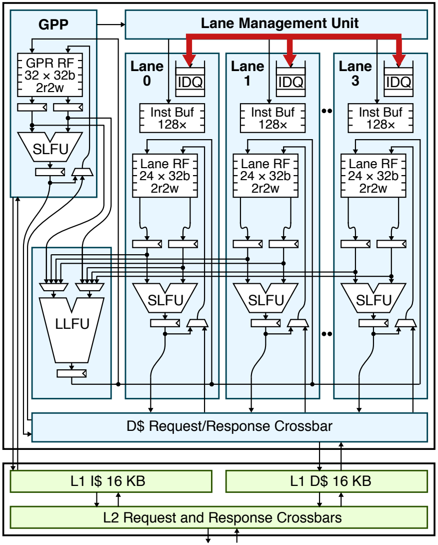
Loop Pattern Specialization Unit

- 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 – `xloop.uc`



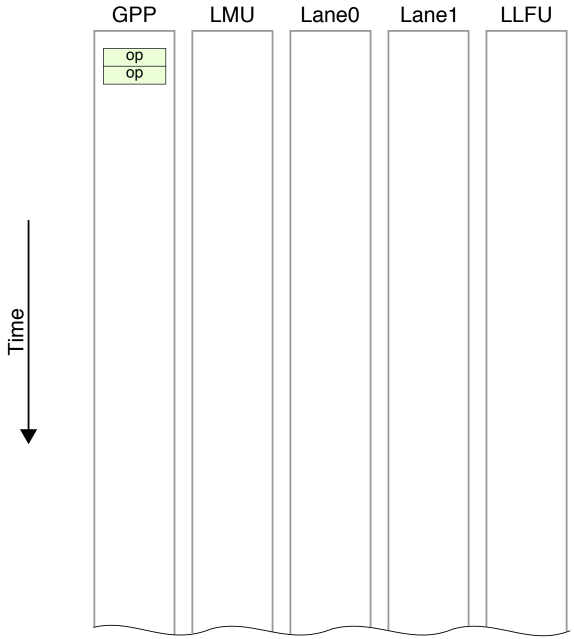
Loop Pattern Specialization Unit

- 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

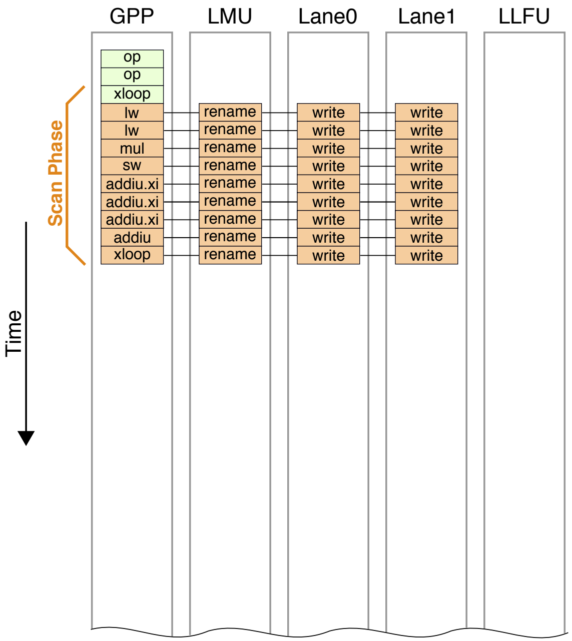
Specialized execution

- Scan phase
- Specialized execution phase

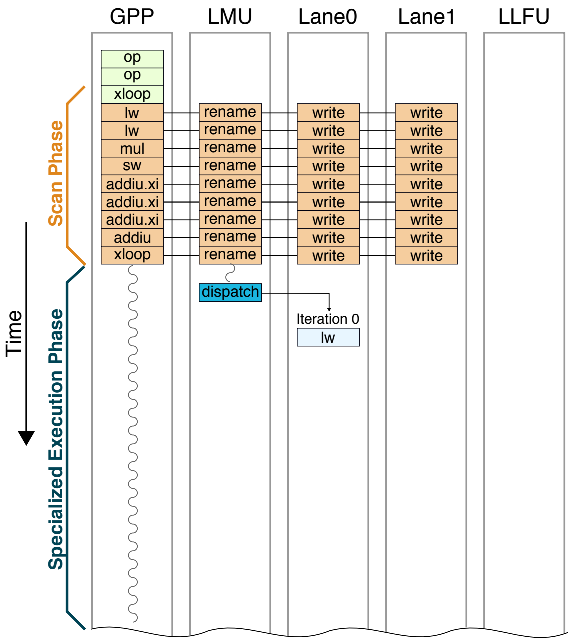
```
loop:
  lw         r2, 0(rA)
  lw         r3, 0(rB)
  mul        r4, r2, r3
  sw         r4, 0(rC)
  addiu.xi   rA, 4
  addiu.xi   rB, 4
  addiu.xi   rC, 4
  addiu      r1, r1, 1
  xloop.uc   r1, rN, loop
```



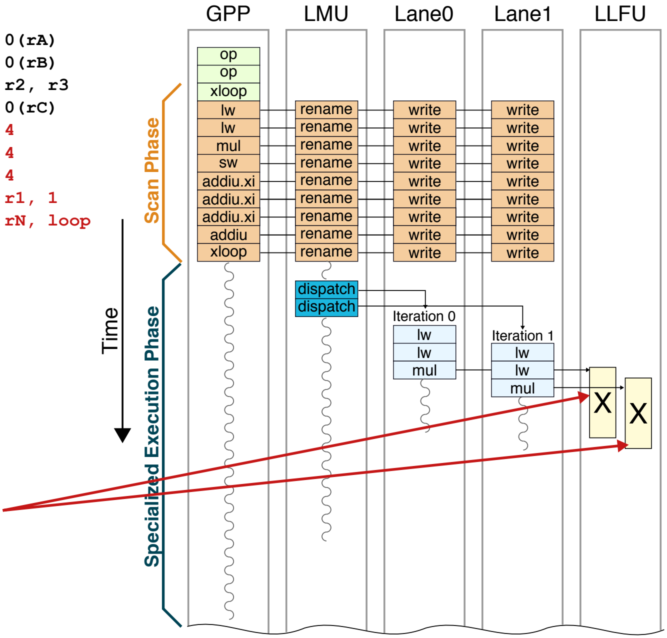
```
loop:
  lw          r2, 0(rA)
  lw          r3, 0(rB)
  mul         r4, r2, r3
  sw          r4, 0(rC)
  addiu.xi    rA, 4
  addiu.xi    rB, 4
  addiu.xi    rC, 4
  addiu       r1, r1, 1
  xloop.uc    r1, rN, loop
```



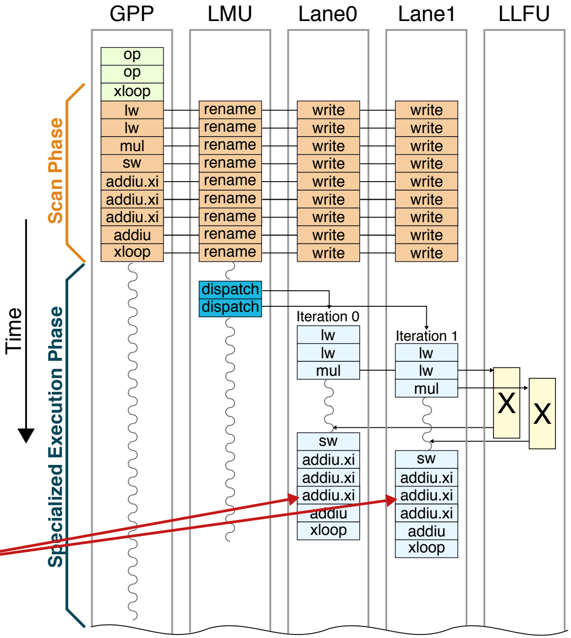
```
loop:
  lw        r2, 0(rA)
  lw        r3, 0(rB)
  mul       r4, r2, r3
  sw        r4, 0(rC)
  addiu.xi  rA, 4
  addiu.xi  rB, 4
  addiu.xi  rC, 4
  addiu     r1, r1, 1
  xloop.uc  r1, rN, loop
```



```
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
```

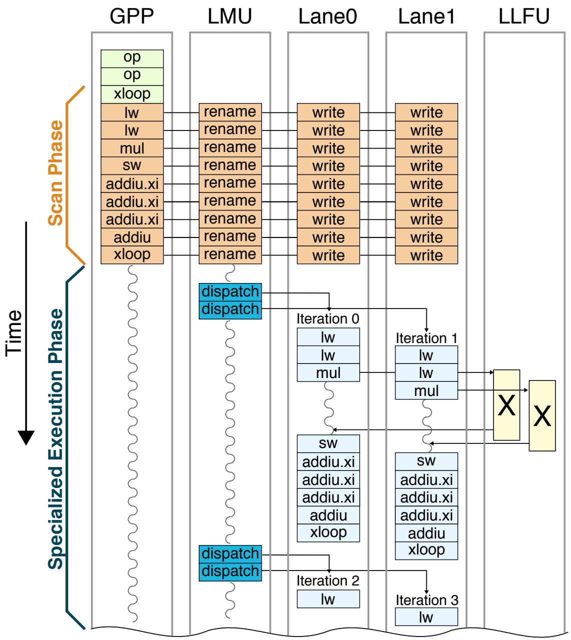


```
loop:
  lw        r2, 0(rA)
  lw        r3, 0(rB)
  mul       r4, r2, r3
  sw        r4, 0(rC)
  addiu.xi  rA, 4
  addiu.xi  rB, 4
  addiu.xi  rC, 4
  addiu     r1, r1, 1
  xloop.uc  r1, rN, loop
```

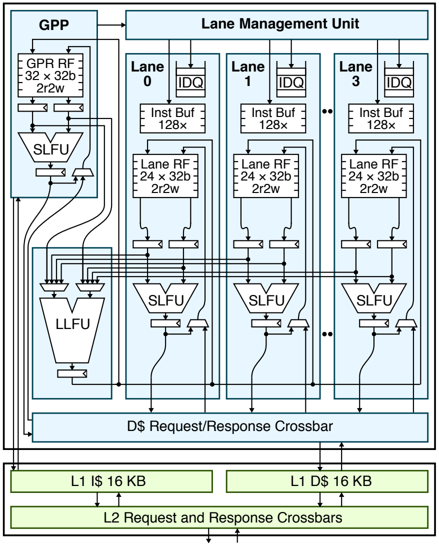


Specialized logic

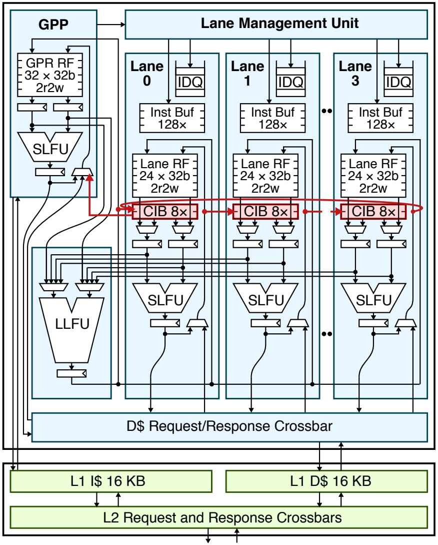
```
loop:
  lw          r2, 0(rA)
  lw          r3, 0(rB)
  mul         r4, r2, r3
  sw          r4, 0(rC)
  addiu.xi    rA, 4
  addiu.xi    rB, 4
  addiu.xi    rC, 4
  addiu       r1, r1, 1
  xloop.uc    r1, rN, loop
```



Specialized Execution – `xloop.or`

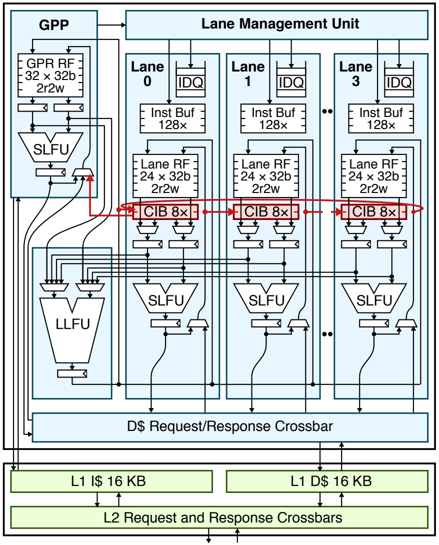


Specialized Execution – `xloop.or`

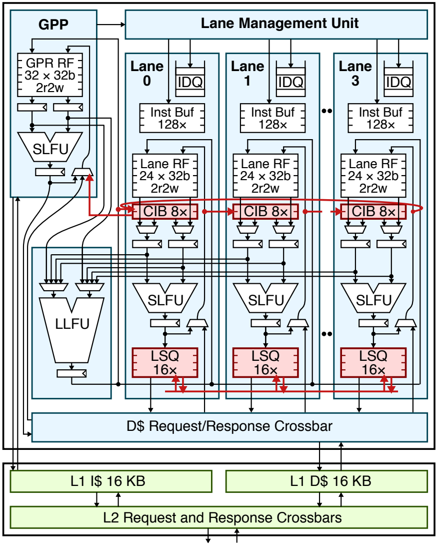


- **Cross-iteration buffers** (CIBs) forward register-dependences
- More details in the paper!

Specialized Execution – `xloop.om`

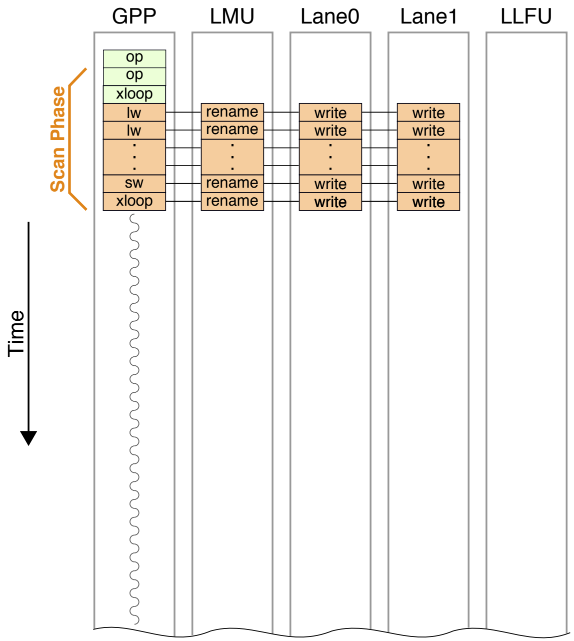


Specialized Execution – `xloop.om`

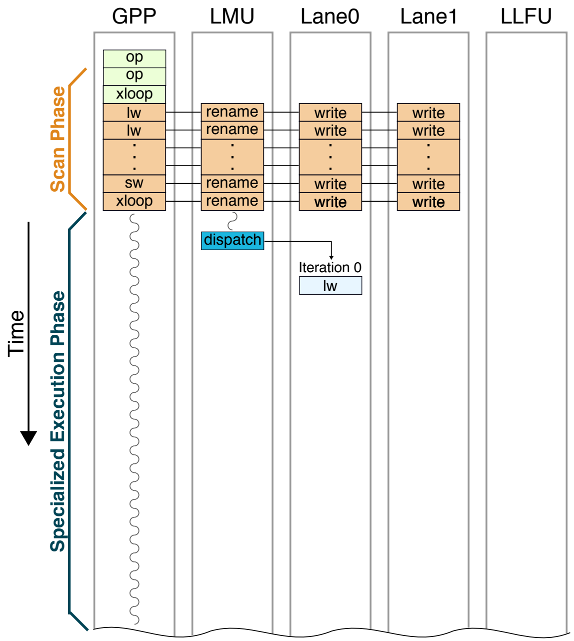


- **LSQ** to support hardware memory disambiguation
- **LMU control logic**
  - Track non-speculative vs. speculative lanes
  - Promote lanes to be non-speculative
- **Lane control logic**
  - Handle structural hazards
  - Handle dependence violations

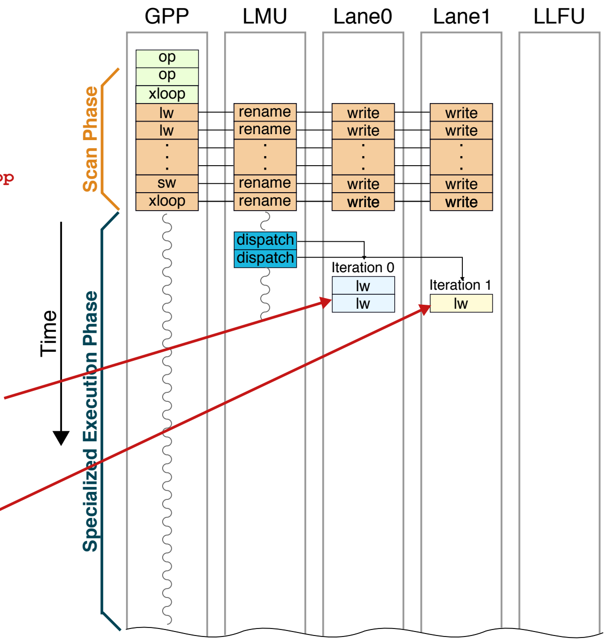
```
loop:
 lw        r4, 0(r3)
 lw        r5, 0(rA)
 ...
 ...
 sw        r6, 0(r7)
 addiu     r1, r1, 1
 xloop.om  r1, rN, loop
```



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



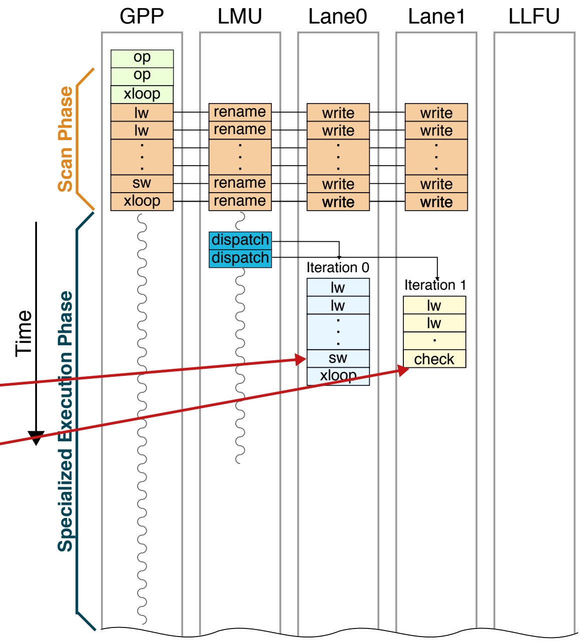
```
loop:
 lw        r4, 0(r3)
 lw        r5, 0(rA)
 ...
 ...
 sw        r6, 0(r7)
 addiu     r1, r1, 1
 xloop.om  r1, rN, loop
```



Non-Speculative Lane

Speculative Lane

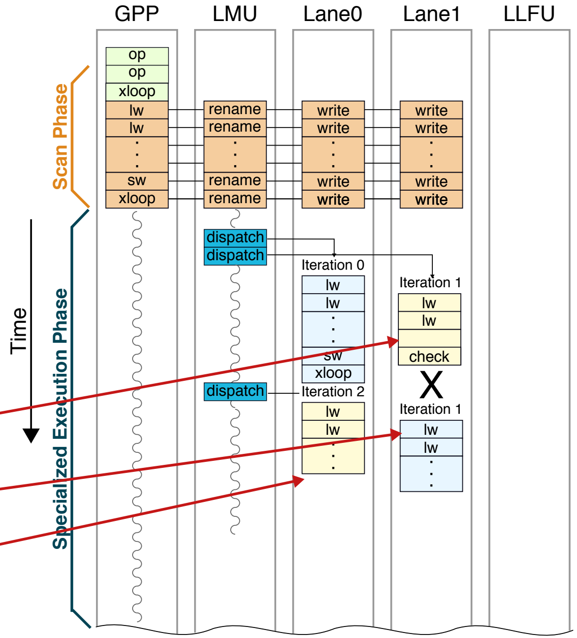
```
loop:
 lw        r4, 0(r3)
 lw        r5, 0(rA)
 ...
 ...
 sw        r6, 0(r7)
 addiu     r1, r1, 1
 xloop.om  r1, rN, loop
```



Broadcast Store

Dependence Check

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



Wasted work

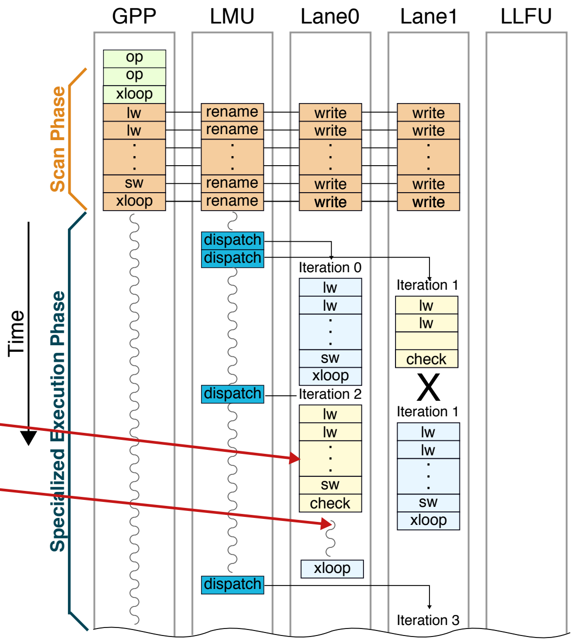
Non-Speculative Lane

Speculative Lane

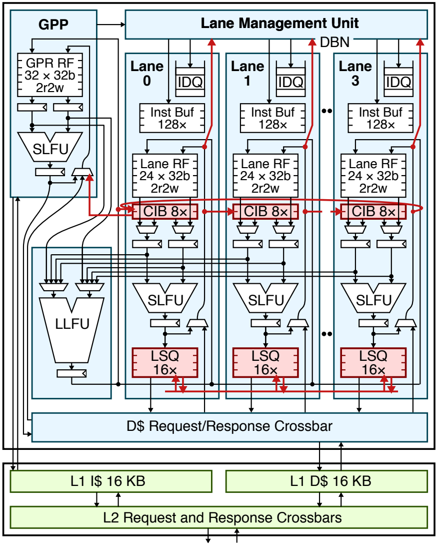
```
loop:
 lw       r4, 0(r3)
 lw       r5, 0(rA)
 ...
 ...
 sw       r6, 0(r7)
 addiu    r1, r1, 1
 xloop.om r1, rN, loop
```

Useful work

Buffered Stores

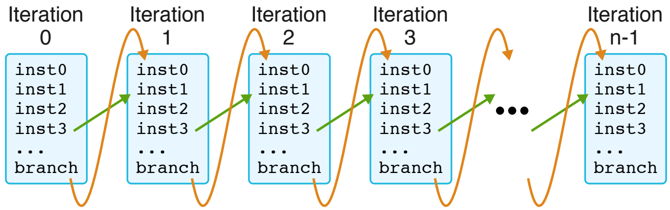


Supporting other patterns



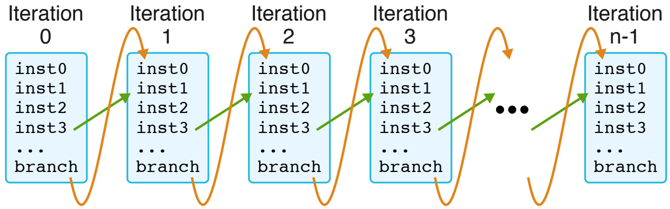
- `xloop.ua` – Using `xloop.om` mechanisms
- `xloop.orm` – Combine `xloop.or` and `xloop.om` mechanisms
- `xloop.*.db`
  - Lanes communicate updates to loop bound
  - LMU tracks maximum bound and generates additional work

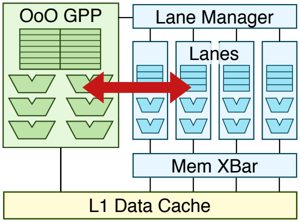
Adaptive Execution



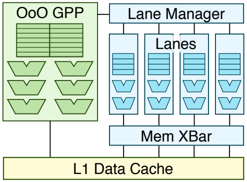
- Significant intra-iteration and limited inter-iteration parallelism
- Specialized execution not beneficial using simple in-order lanes

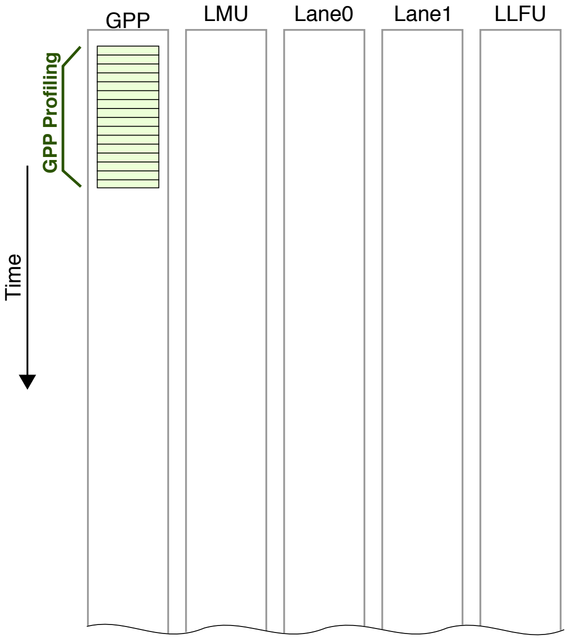
Adaptive Execution

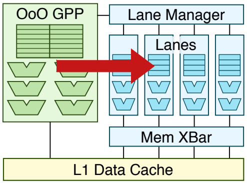


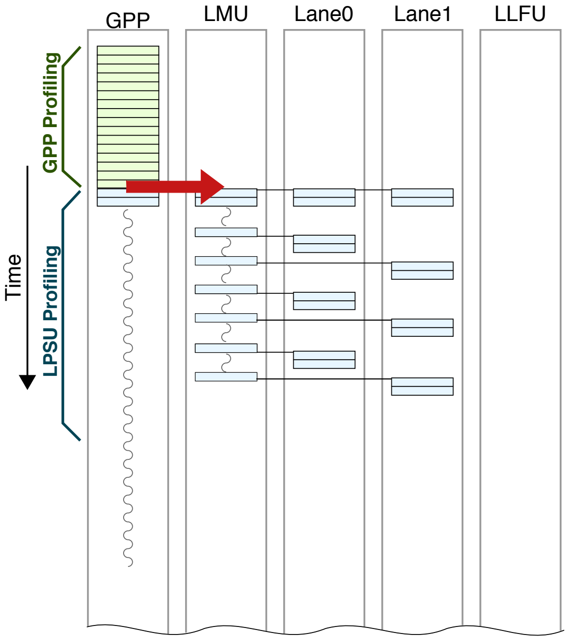


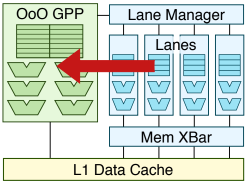
- Significant intra-iteration and limited inter-iteration parallelism
- Specialized execution not beneficial using simple in-order lanes
- **Adaptively migrate** to complex OoO cores

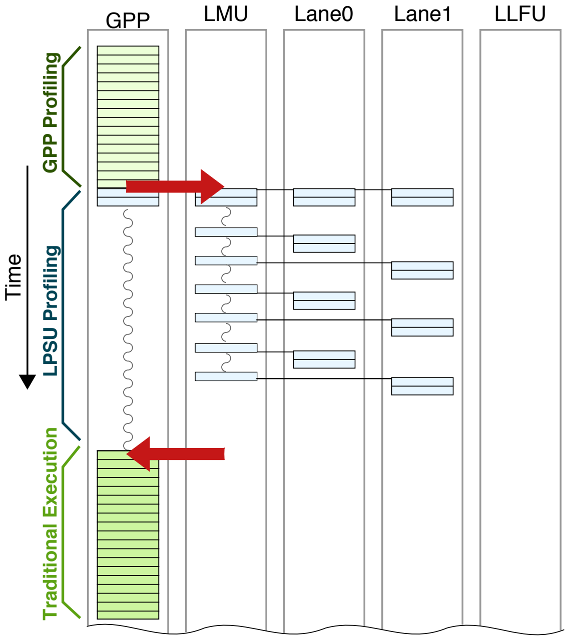

OoO GPP
Lane Manager
Lanes
Mem XBar
L1 Data Cache


GPP
LMU
Lane0
Lane1
LLFU
GPP Profiling
Time


OoO GPP
Lane Manager
Lanes
Mem XBar
L1 Data Cache


GPP
LMU
Lane0
Lane1
LLFU
GPP Profiling
LPSU Profiling
Time


OoO GPP
Lane Manager
Lanes
Mem XBar
L1 Data Cache


GPP
LMU
Lane0
Lane1
LLFU
Time
GPP Profiling
LPSU Profiling
Traditional Execution

1. XLOOPS ISA

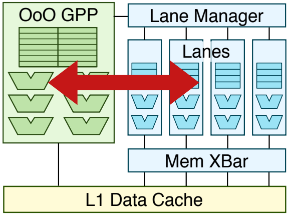
```
loop:
 lw        r2, 0(rA)
 lw        r3, 0(rB)
 ...
 ...
 addiu.xi rA, 4
 addiu.xi rB, 4
 addiu    r1, r1, 1
 xloop.uc r1, rN, loop
```

2. XLOOPS Compiler

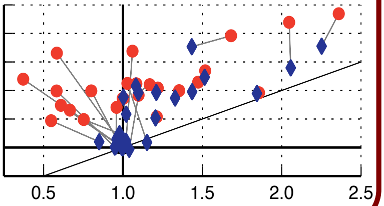
```
#pragma xloops ordered
for(i = 0; i < N i++)
 A[i] = A[i] * A[i-K];

#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

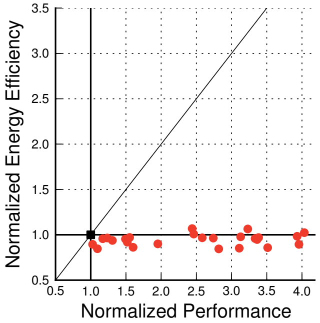
25 Kernels from MiBench, PolyBench, PBBS, and Custom

Cycle-Level Methodology

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

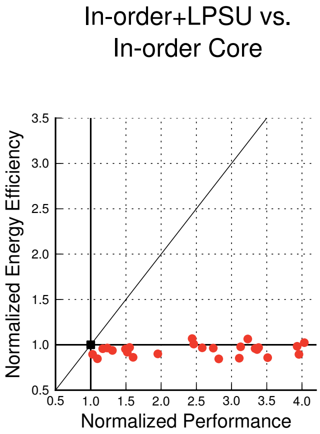
Energy-Efficiency vs. Performance Results

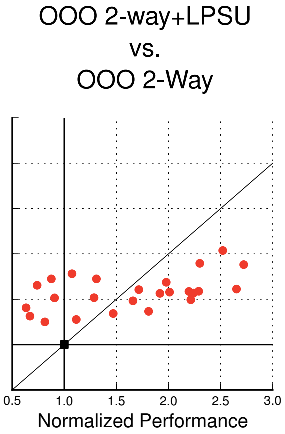
In-order+LPSU vs.
In-order Core



- Competitive energy efficiency
- Higher dynamic power
- Always higher performance

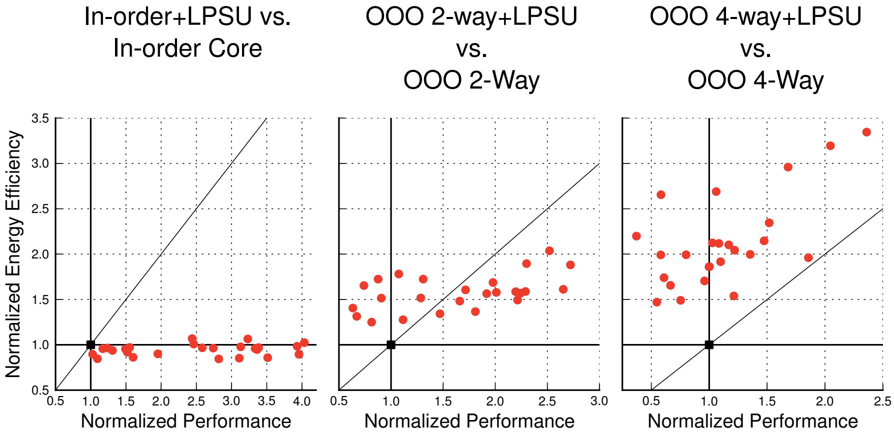
Energy-Efficiency vs. Performance Results





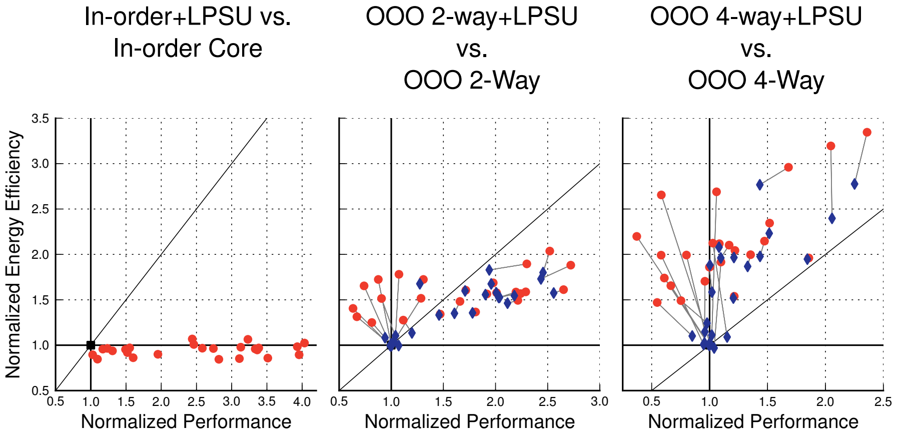
- Always more energy efficient
- Mixed dynamic power
- Competitive or higher performance (`uc/or/om/ua/db`)

Energy-Efficiency vs. Performance Results



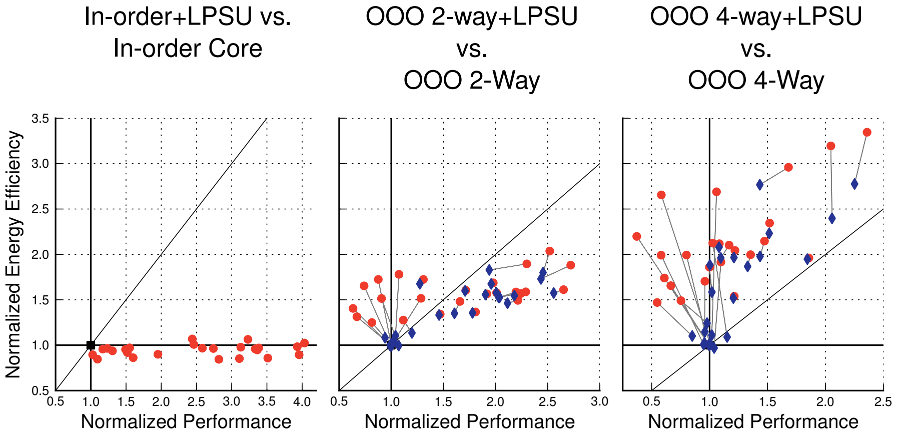
- Always more energy efficient
- Always lower dynamic power
- Mixed performance (`uc/om/ua/db`)

Energy-Efficiency vs. Performance Results

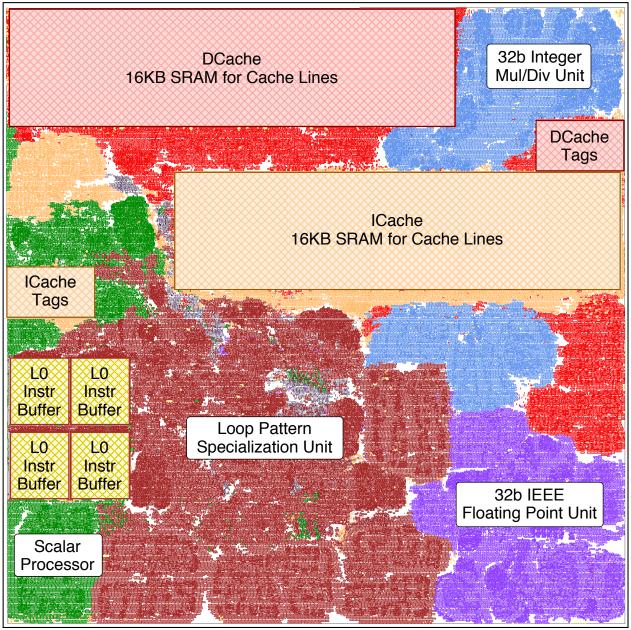


- Trade energy efficiency for performance for slower kernels
- Profiling and migration cause minimal performance degradtion

Energy-Efficiency vs. Performance Results



More results in the paper!



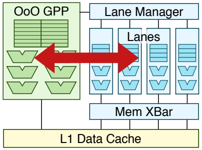
VLSI Implementation

- 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] ]++;
```



Take-Away Points

- Elegant new abstraction that enables performance-portable execution of loops
- 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.