Architectural Specialization for Inter-Iteration Loop Dependence Patterns

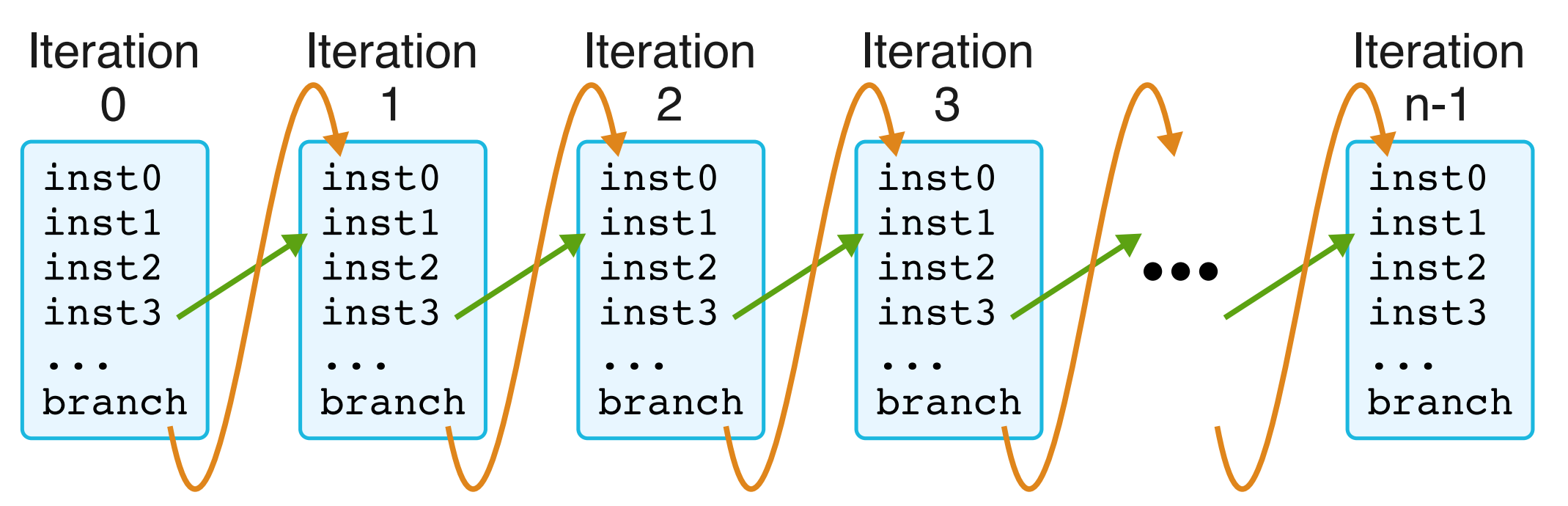
Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu
Zhiru Zhang, Christopher Batten
Computer Systems Laboratory
School of Electrical and Computer Engineering, Cornell University

1 Abstract

Hardware specialization is an increasingly common technique to improve performance and energy efficiency in spite of the diminished benefits of technology scaling. We are pursuing a single-ISA heterogeneous architecture called explicit loop specialization (XLOOPS) that transparently integrates general-purpose processors (GPPs) and specialized loop accelerators. XLOOPS supports a variety of inter-iteration data- and control-dependence patterns for both single and nested loops. The XLOOPS hardware/software abstraction requires only lightweight changes to a general-purpose compiler to generate XLOOPS binaries and enables executing these binaries on: (1) **traditional microarchitectures** with minimal performance impact, (2) **specialized microarchitectures** to improve performance and/or energy efficiency, and (3) **adaptive microarchitectures** that can seamlessly migrate loops between traditional and specialized execution. We evaluate XLOOPS using a vertically integrated research methodology and show compelling performance and energy efficiency improvements compared to both simple and comple GPPs.

2 Motivation

Computer architects have long realized the importance of focusing on the key loops that often dominate application performance. This has led to a diverse array of specialized hardware for exploiting loop dependence patterns. In this work, we focus on architectural specialization for inter-iteration loop dependence patterns.



Inter-iteration data-dependence patterns include:

- ▷ Loops with no inter-iteration dependences
- ▷ Loops with inter-iteration dependences encoded through registers and/or memory
- ▷ Loops that can execute in any order as long as updates to memory appear atomic

Inter-iteration control-dependence patterns include:

- ▷ Loops that terminate after comparing induction variable to loop-invariant bound
- ▷ Loops that terminate based on a data-dependent-exit condition
- ▷ Loops that can monotonically increase the loop bound during the loop's execution

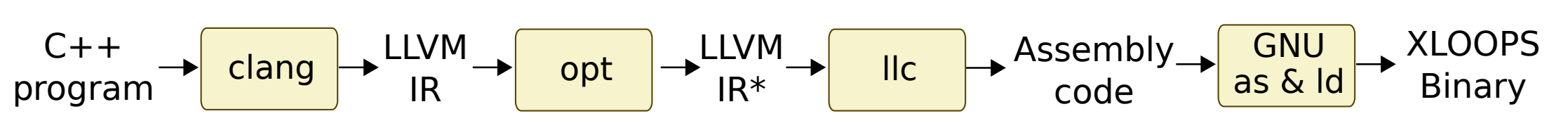
3 XLOOPS Compiler

We implemented an LLVM-based compiler framework that can compile `pragma`-annotated application kernels drawn from several benchmark suites and our own custom benchmarks.

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] );
```

XLOOPS Compilation Flow



The XLOOPS compiler includes analysis passes to determine the type of inter-iteration data-dependence and control-dependence patterns. Register-dependence testing is implemented by analyzing the use-definition chains through the PHI nodes, and memory-dependence testing is implemented using well-known dependence techniques such as ZIV/SIV/MIV tests.

4 XLOOPS Instruction Set

The XLOOPS instruction set is carefully designed to enable efficient execution on both traditional general-purpose processors (GPPs) and specialized microarchitectures. The XLOOPS instructions encode the notion of a parallel loop body and the inter-iteration data- and control-dependence patterns as shown below.

`xloop.uc rI, rN, L`	unordered-concurrent
`xloop.ua rI, rN, L`	unordered-atomic
`xloop.or rI, rN, L`	ordered through registers
`xloop.om rI, rN, L`	ordered through memory
`xloop.*.db rI, rN, L`	dynamic-loop-bound
`addiu.xi X, imm`	encode mutual induction variables
`addu.xi rT`	encode mutual induction variables

Code and Assembly Examples

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

L:
  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, L
```

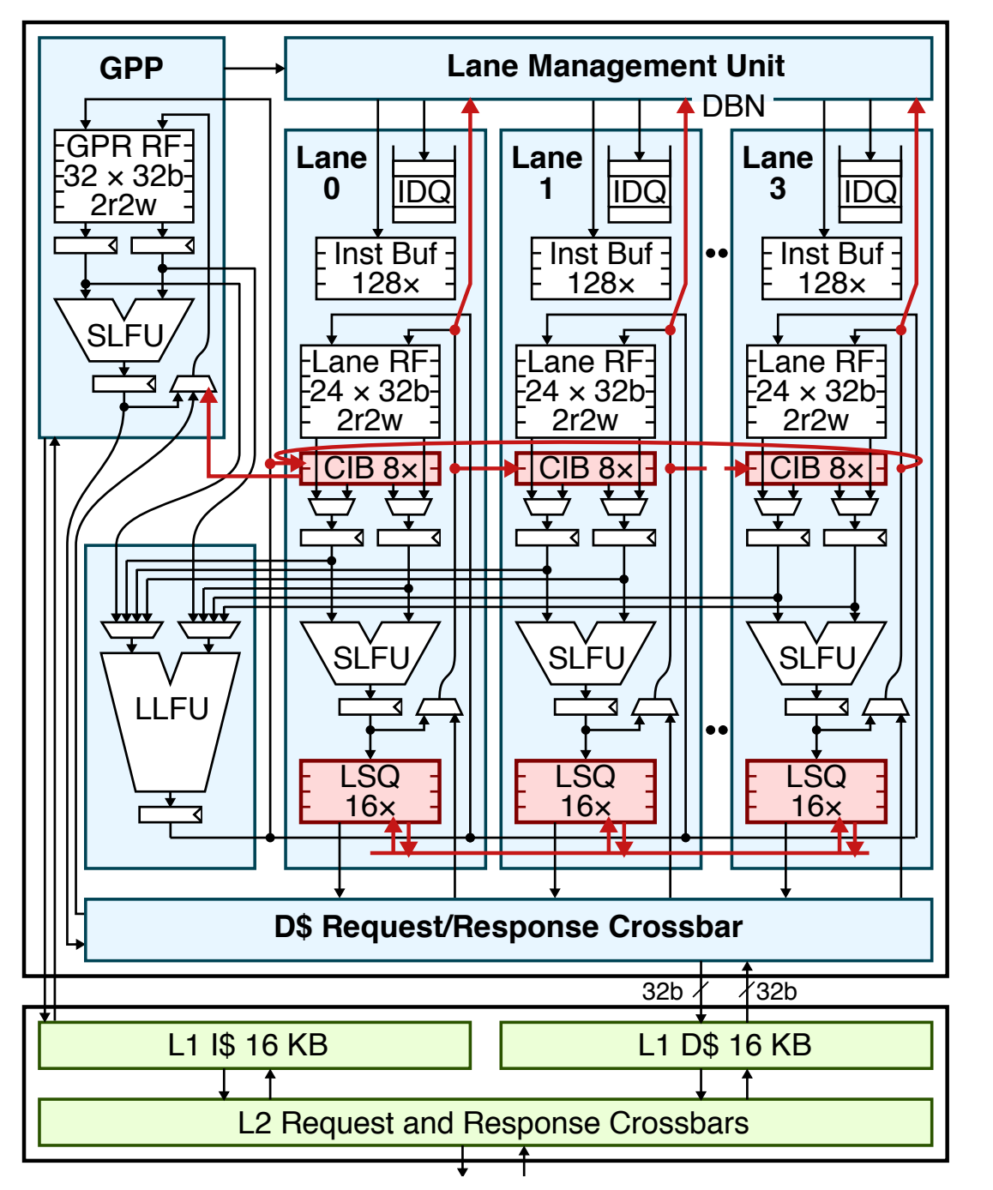
```
#pragma xloop ordered
for ( X=0, i=0; i<N; i++ )
  X += A[i]; B[i] = X

 L:
  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, L
```

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

  move     r1, rK
  sll      r2, rK, 0x2
  addu     r3, rA, r2
L:
  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, L
```

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

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

5 XLOOPS Microarchitecture

A GPP augmented with a *loop-pattern specialization unit* (LPSU) that contains a lane management unit and a number of decoupled lanes for executing iterations in parallel. The GPP and the lanes in the LPSU share long-latency functional units (LLFUs) and data-memory ports.



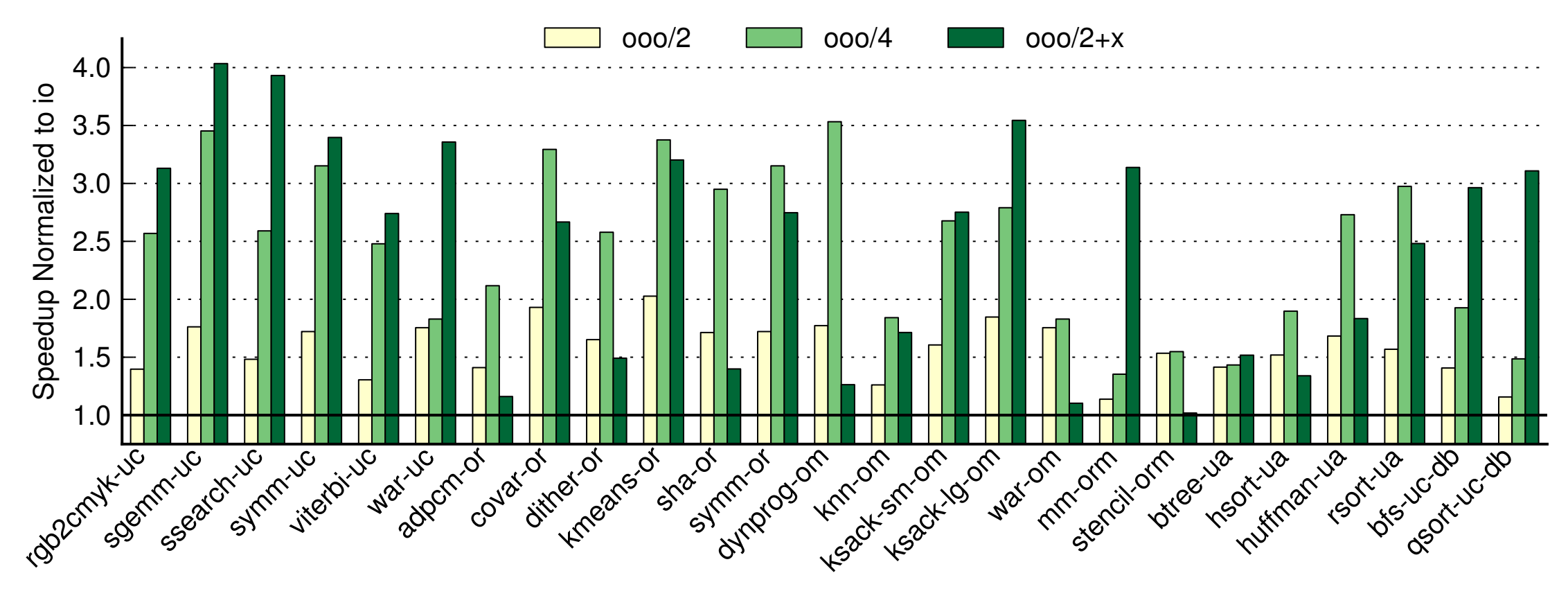
- ▷ **Traditional Execution –** An `xloop` instruction is executed as a conditional branch, and an `xi` instruction is executed as simple addition.
- ▷ **Specialized Execution –** Specialized execution occurs in two phases: *scan phase* where the GPP scans the `xloop` and configures the LPSU and *specialized execution phase* where the LPSU executes the iterations in parallel.
- ▷ **Adaptive Execution –** Adaptive execution mechanism that adds two phases, *GPP profiling phase* and *LPSU profiling phase* to determine the best performing microarchitecture and adaptively migrates the loop execution.

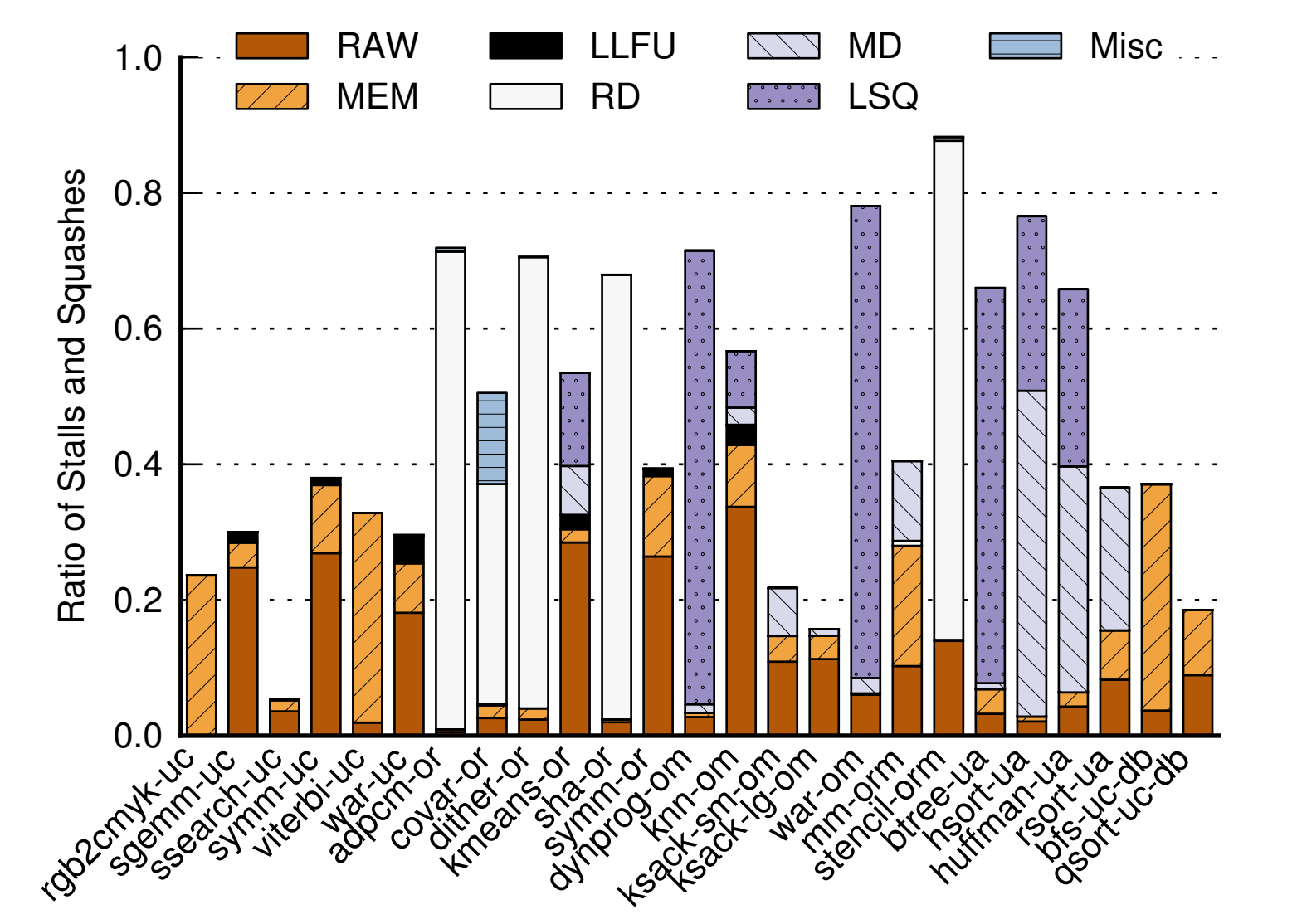
6 Cycle-Level Evaluation

We modified a gem5+McPAT-1.0 simulation framework to model both in-order and out-of-order processors augmented with an LPSU. We compare XLOOPS to three baseline GPPs: a simple single-issue in-order processor (*io*), a moderate 2-way out-of-order superscalar processor (*ooo/2*), and an aggressive 4-way out-of-order superscalar processor (*ooo/4*). We augmented each baseline GPP with an LPSU to create three XLOOPS configurations: *io+x*, *ooo/2+x*, and *ooo/4+x*.

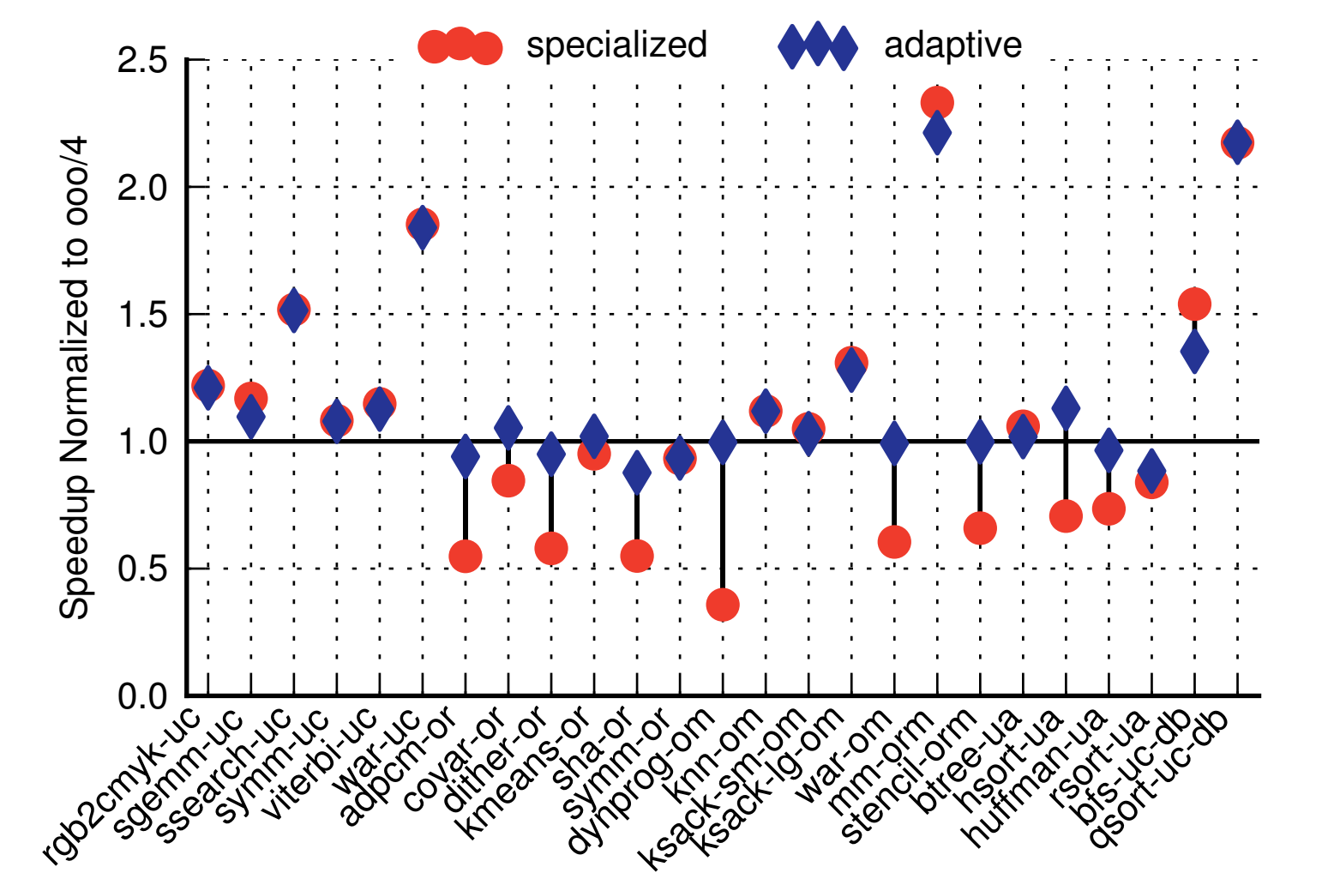
Performance Results

We observe that specialized execution always benefits the in-order processor. For a total of 25 application kernels, specialized execution performs better for 18 kernels compared to *ooo/2*, and performs better for 12 kernels compared to *ooo/4*.



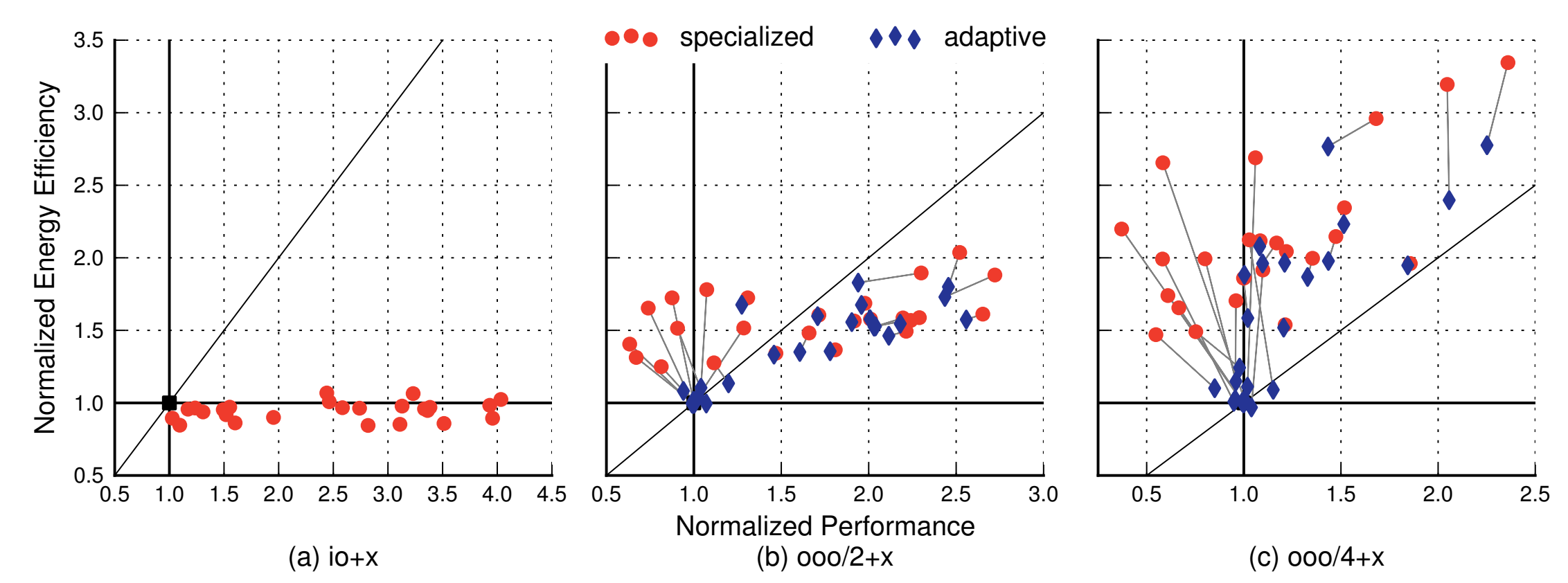


- ▷ Sharing the LLFUs does not significantly hurt specialized execution.
- ▷ Sharing the memory port could improve the performance for some kernels.
- ▷ Inter-iteration critical path restricts performance of `xloop.or` kernels.
- ▷ Input datasets influence performance of `xloop.{om,orm,ua}` kernels.



- ▷ Overheads of profiling and work migration result in only minimal performance degradation.
- ▷ `xloop.or` kernels benefit from traditional execution when there is high intra-iteration ILP.
- ▷ Adaptive work migration helps `xloop.{om,orm,ua}` kernels when there is limited MLP.

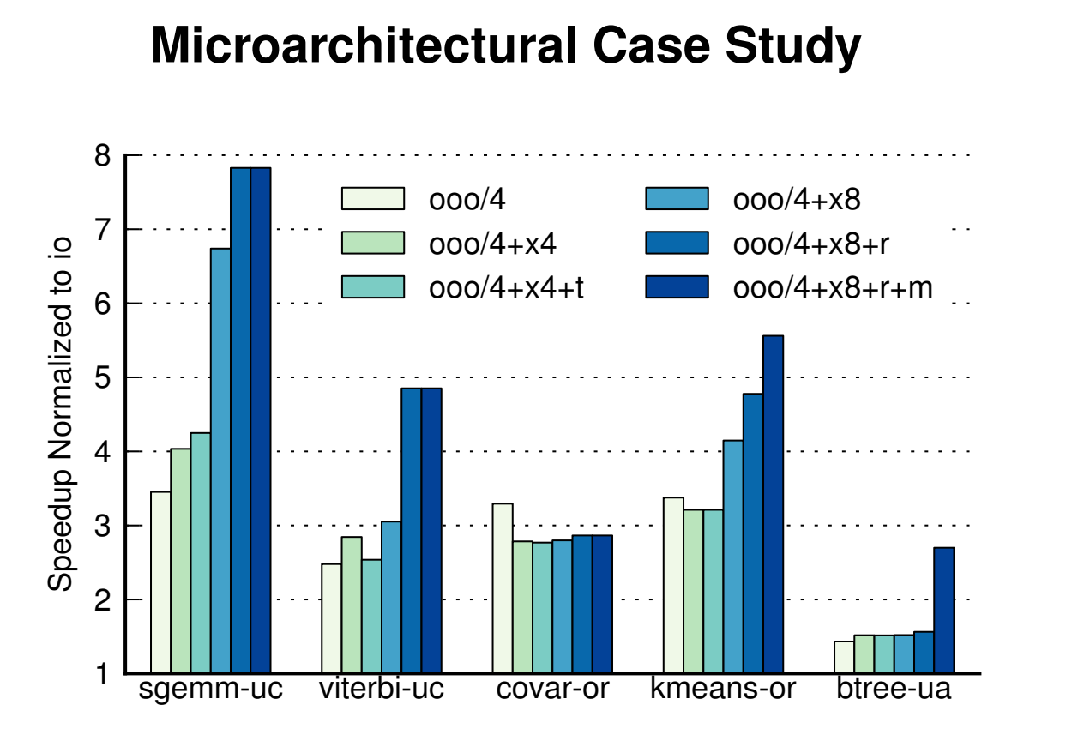
Energy Efficiency vs. Performance Results



Specialized execution adds minimal energy overhead and results in increased performance for on *io+x*. Specialized execution is more energy efficient for *ooo/2+x* and *ooo/4+x*.

7 Case Studies

We explored the microarchitectural design space by adding limited vertical multi-threading, scaling the number of lanes, scaling shared resources, and increasing the per-lane LSQ entries.



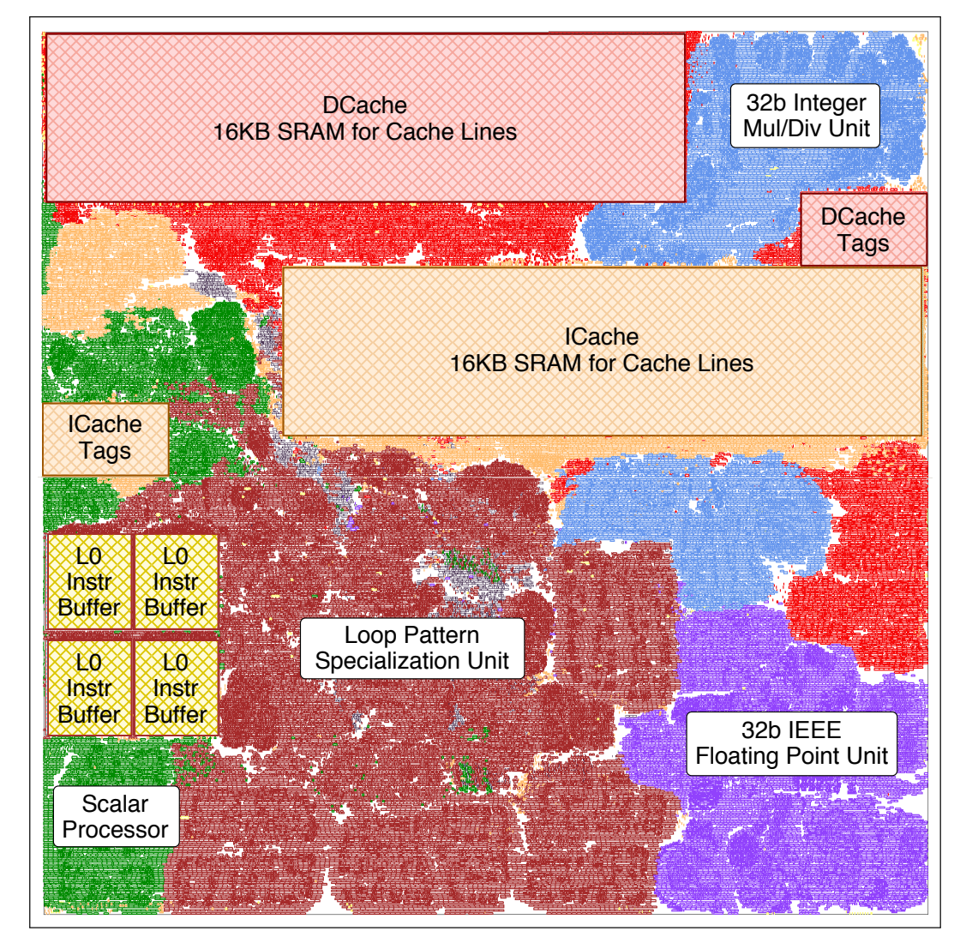
- ▷ Limited vertical multi-threading and increased lanes only helps for few kernels.
- ▷ Doubling shared resources helps to reduce memory contention and LLFU structural hazards.
- ▷ Scaling resources does not help overcome inter-iteration register dependences.

Application Case Studies

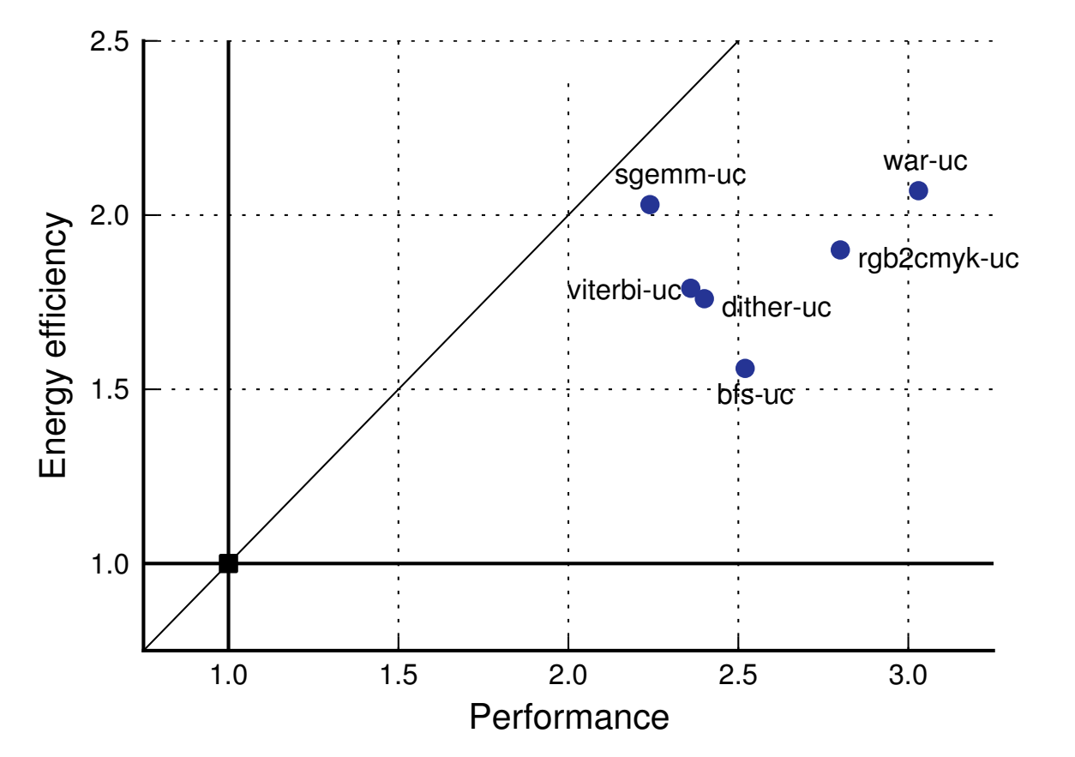
- ▷ Hand-optimizing select `xloop.or` kernels to reduce the cross-iteration iteration critical path improves performance by 50–70%.
- ▷ Simply annotating serial versions of the kernels often performs better than code with significant loop transformations which shows that XLOOPS allows ease-of-programmability without sacrificing performance.

8 RTL/VLSI Evaluation

We implemented a register-transfer-level (RTL) model for a basic LPSU that supports `xloop.uc` instructions. We target a 40 nm TSMC process using a Synopsys ASIC CAD toolflow: VCS for RTL simulation, DesignCompiler for synthesis, IC Compiler for place-and-route, and PrimeTime for power analysis.


DCache 16KB SRAM for Cache Lines
32b Integer Mul/Div Unit
DCache Tags
ICache 16KB SRAM for Cache Lines
ICache Tags
L0 Instr Buffer
Loop Pattern Specialization Unit
32b IEEE Floating Point Unit
Scalar Processor

- ▷ Total area of the LPSU design is 0.36 mm^2 which is only 43% larger than the in-order GPP (0.25 mm^2).
- ▷ Sharing the LLFUs and data-memory port is a key design decision that results in incurring minimal area overheads.
- ▷ Scaling experiments show that area overhead of a given LPSU design roughly increases linearly with the number of lanes (≈10%).



- ▷ Specialized execution improves performance by 2.4–4× and energy efficiency by 1.6–2.1×.
- ▷ Accessing instruction buffer is cheaper by a factor of ten compared to accessing the instruction cache.
- ▷ McPAT results are relatively conservative which motivates RTL implementation of other patterns.

9 Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, NSF XPS Award #1337240, a DARPA Young Faculty Award, and donations from Intel Corporation, Synopsys, Inc, and Xilinx, Inc. The authors acknowledge and thank Pol Rosello and Paul Jackson for their help in writing XLOOPS application kernels, Christopher Torng for his help bringing up the gem5 simulation framework, Aadeetya Shreedhar for his help in bringing up traditional execution on gem5, and Derek Lockhart for developing the Python hardware modeling framework used in the cycle-level model of the LPSU.

Publication: Appears in the Proceedings of the 47th Int'l Symp. on Microarchitecture (MICRO-47), Dec. 2014. **URL:** `http://www.csl.cornell.edu/~cbatten/pdfs/srinath-xloops-micro2014.pdf` **Contact Author:** Shreesha Srinath, 364 Upson Hall, Ithaca, NY 14853, `ss2783@cornell.edu`