# Hardware-Software Codesign for Efficient Computing

Magnus Själander

Associate Professor
Norwegian University of Science and Technology

# The Free Lunch is Over

- Technology and frequency scaling are at a standstill
- Most (all) designs are power limited
- Performance improvements require solutions that are more energy efficient, two examples:
  - Co-optimization of hardware and software
  - Specialization
- Both cases require strong compiler support to extract and represent program properties

1. Rethinking the HW-SW Interface

2. Software Programmable Accelerator

3. A New Intermediate Representation

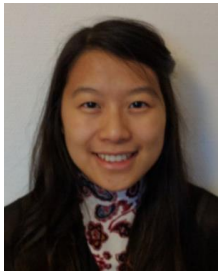
# Collaborators

NTNU

UPPSALA UNIVERSITET

NUS

XILINX



Kim-Anh Tran

Alexandra Jimborean

Trevor Carlson

Stefanos Kaxiras

Konstantinos Koukos

Yaman Umuroglu

Vasileios Spiliopoulos

Lahiru Rasnayake

Nico Reissmann

Jan Christian Meyer

# 1. Rethinking the HW-SW Interface

SWOOP: Software-Hardware Co-design for Non-speculative, Execute-Ahead, In-Order Cores

K. A. Tran, A. Jimborean, T. E. Carlsson, K. Koukus M. Själander, and S. Kaxiras

PLDI 2018

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Load value

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Load value

Use value

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

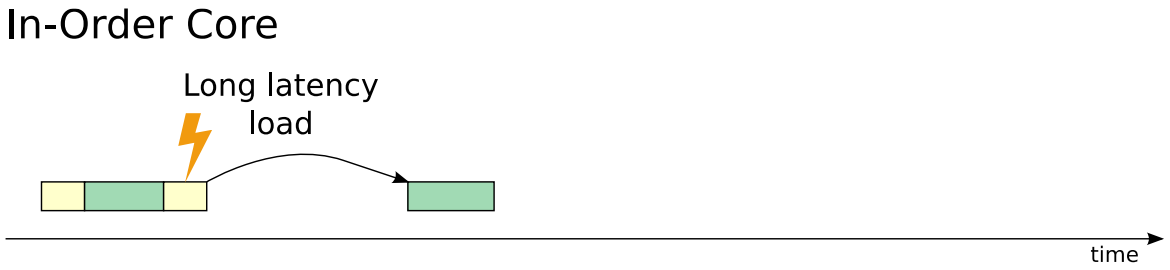
Long latency load

# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Long latency load

Stall on use

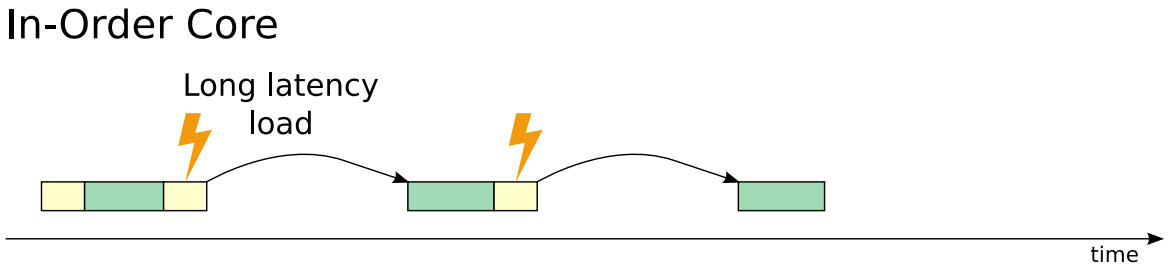


# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Long latency load

Stall on use

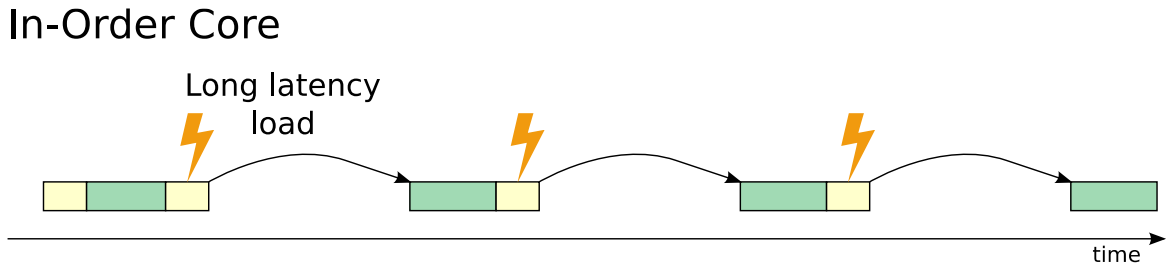


# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

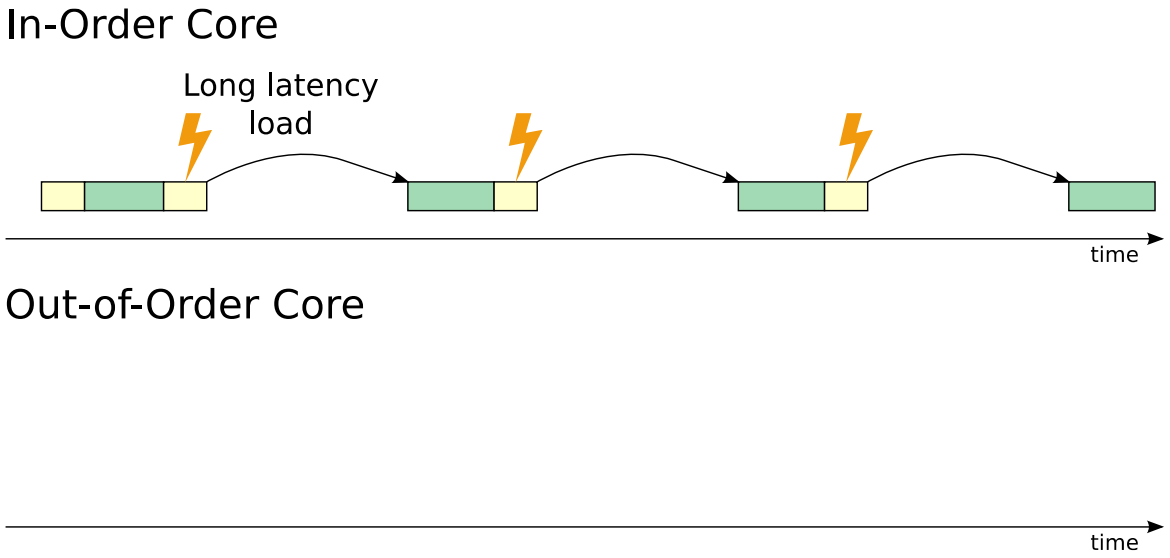
Long latency load

Stall on use



# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

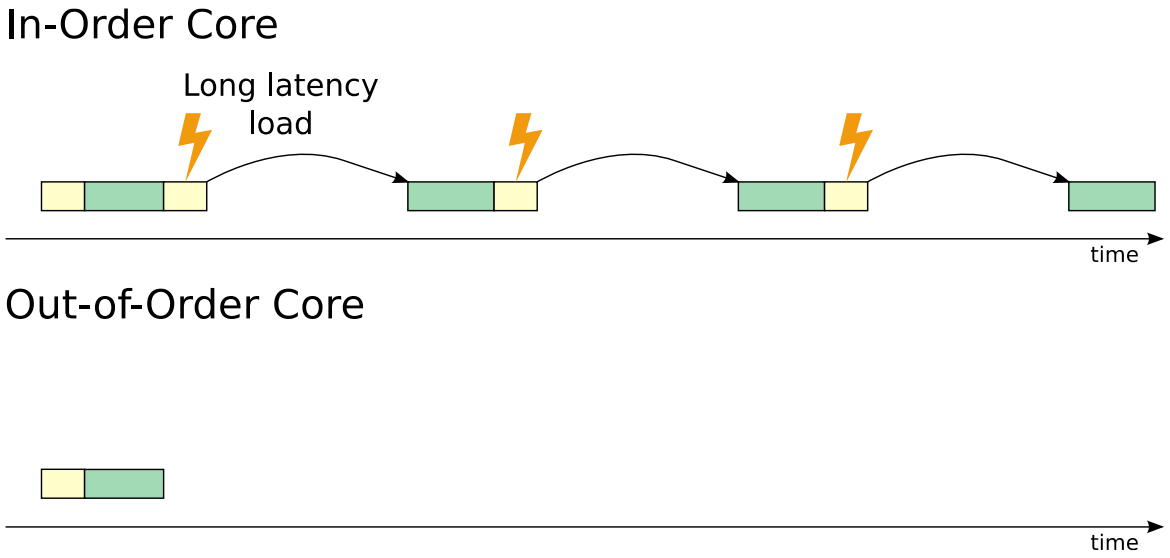


# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Load value

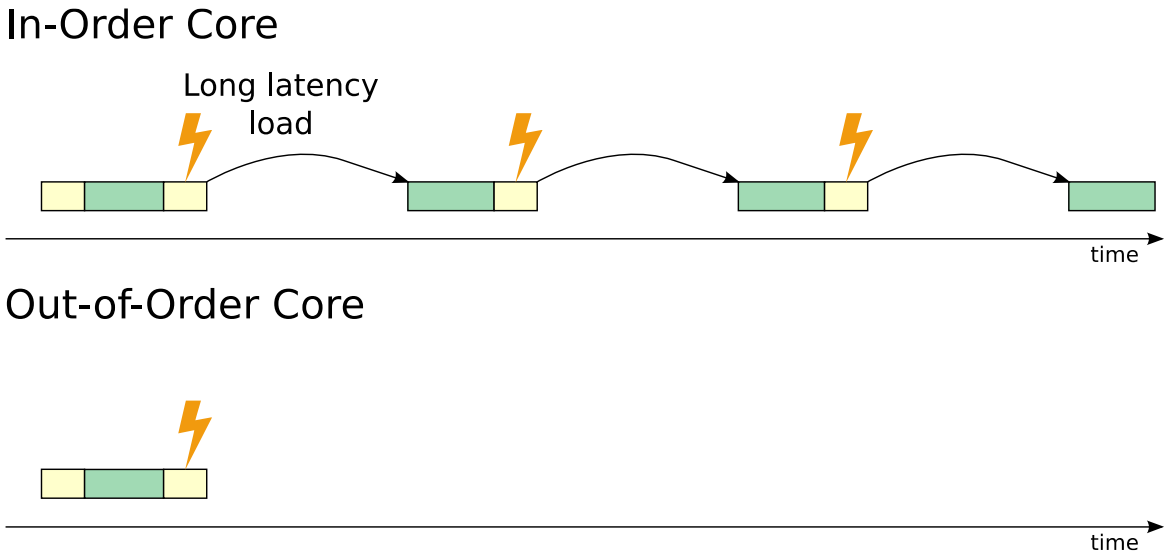
Use value



# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

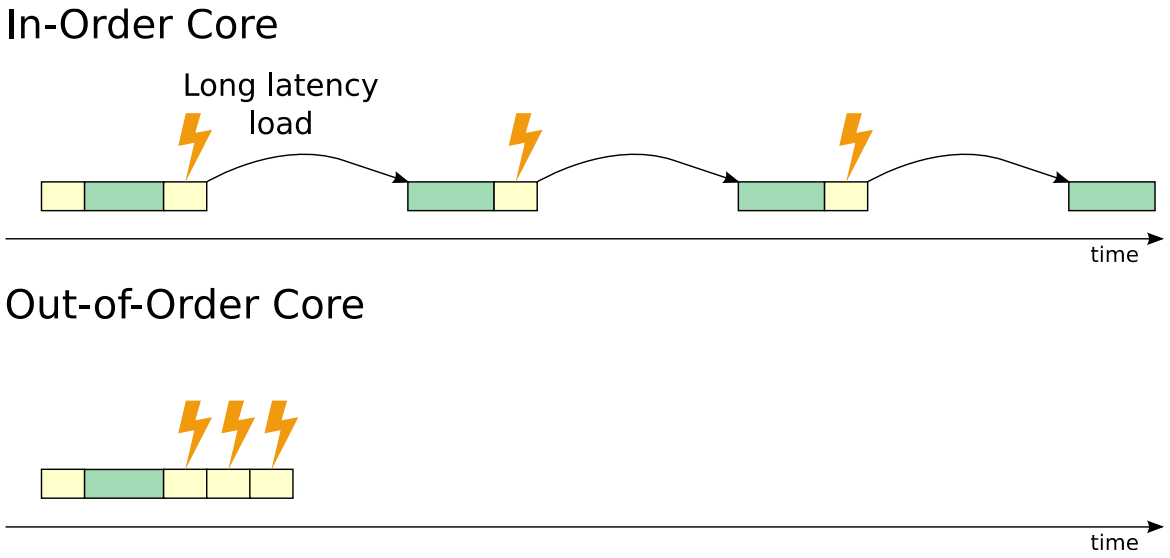
Long latency load



# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Long latency loads

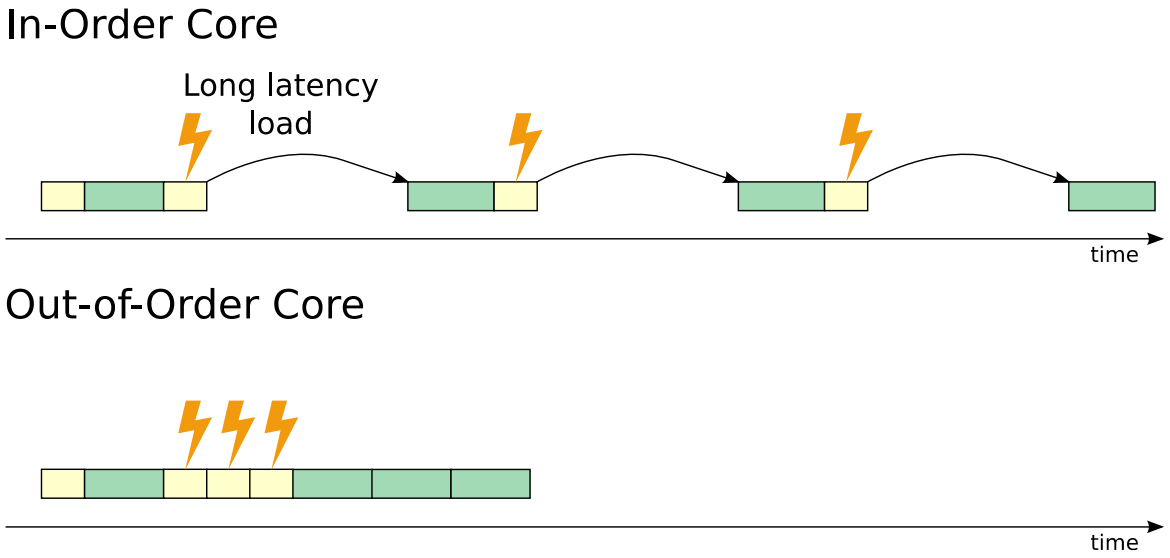


# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Long latency loads

Use values

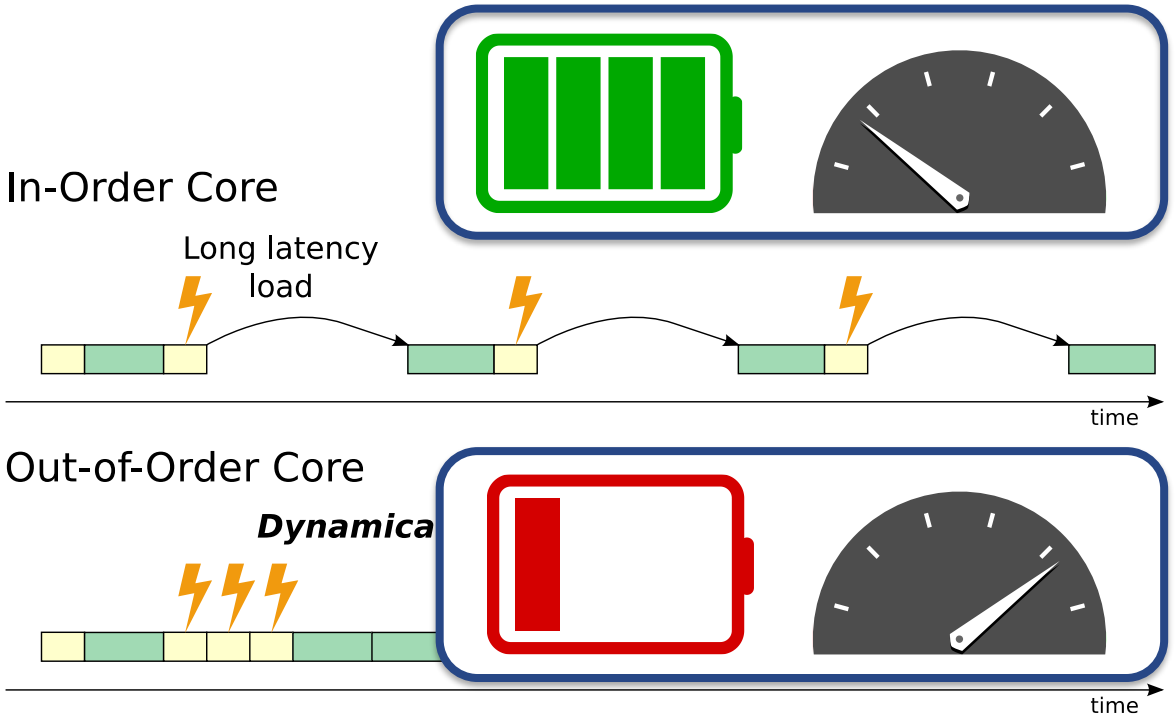


# Motivation

```
for (int i=0; i<N; i++) {
    tmp = x[i];
    if (tmp > 0)
        sum = tmp + sum;
}
```

Long latency loads

Use values

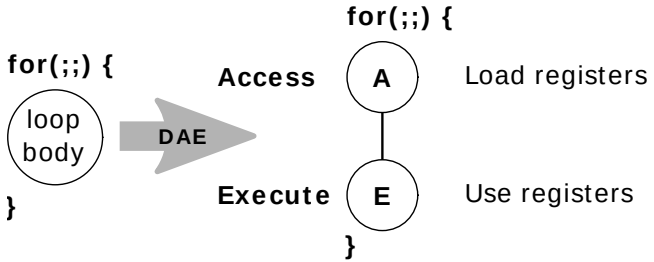


# SWOOP Software/Execution Model

- SWOOP targets loops

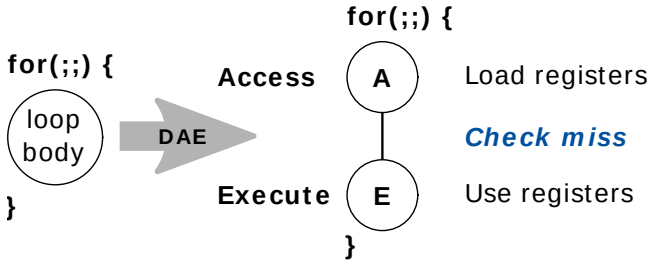
```
for(;;) {
  loop
  body
}
```

# SWOOP Software/Execution Model



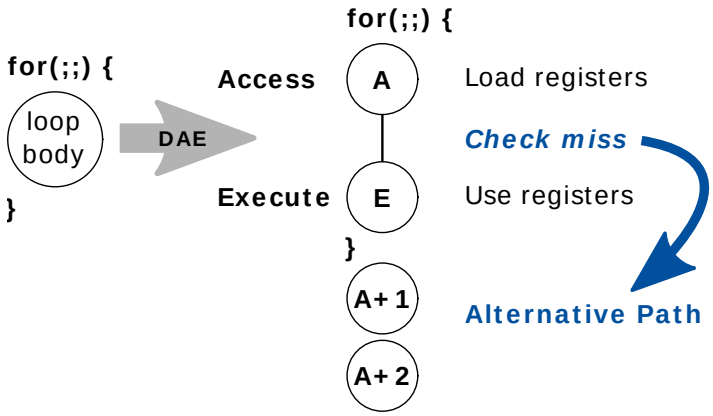

- SWOOP targets loops
- Amenable loops are split into an **Access** and an **Execute** phase (DAE)

# SWOOP Software/Execution Model



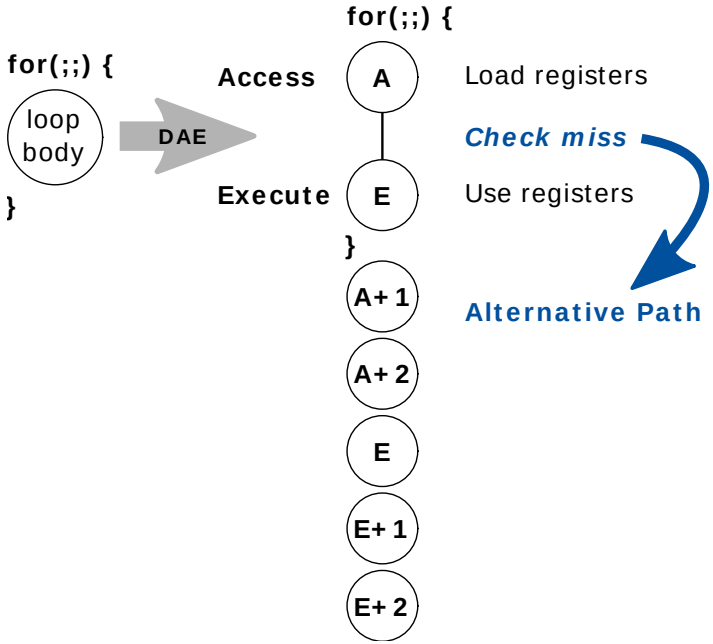

- SWOOP targets loops
- Amenable loops are split into an **Access** and an **Execute** phase (DAE)
- Check miss instruction

# SWOOP Software/Execution Model



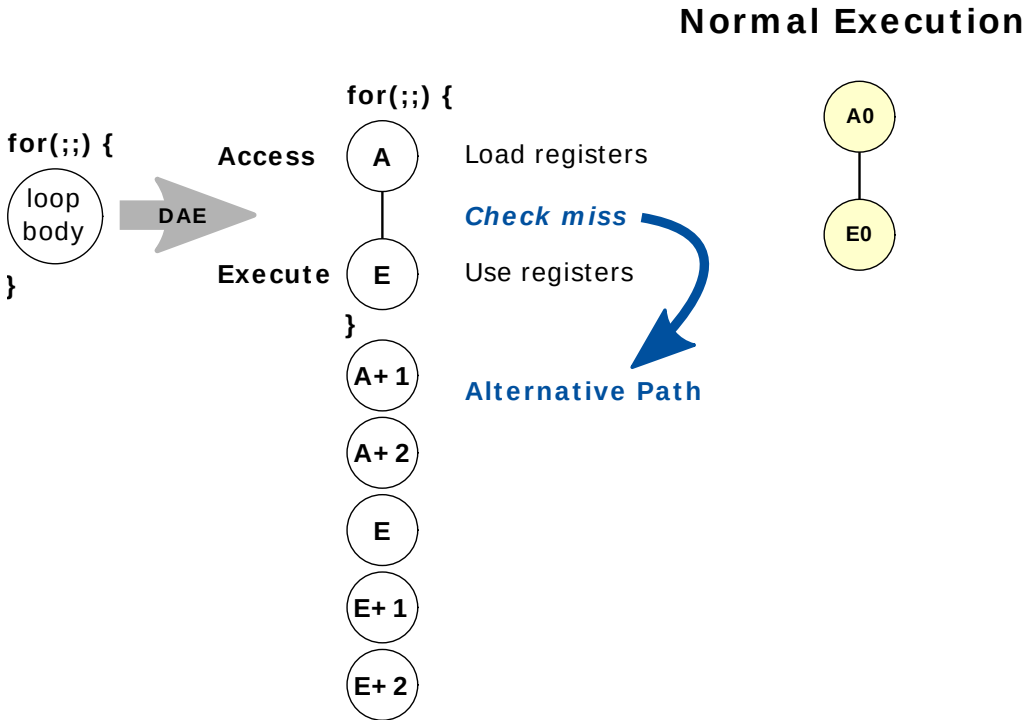

- SWOOP targets loops
- Amenable loops are split into an **Access** and an **Execute** phase (DAE)
- Check miss instruction
- Branch to alternative path on load miss

# SWOOP Software/Execution Model

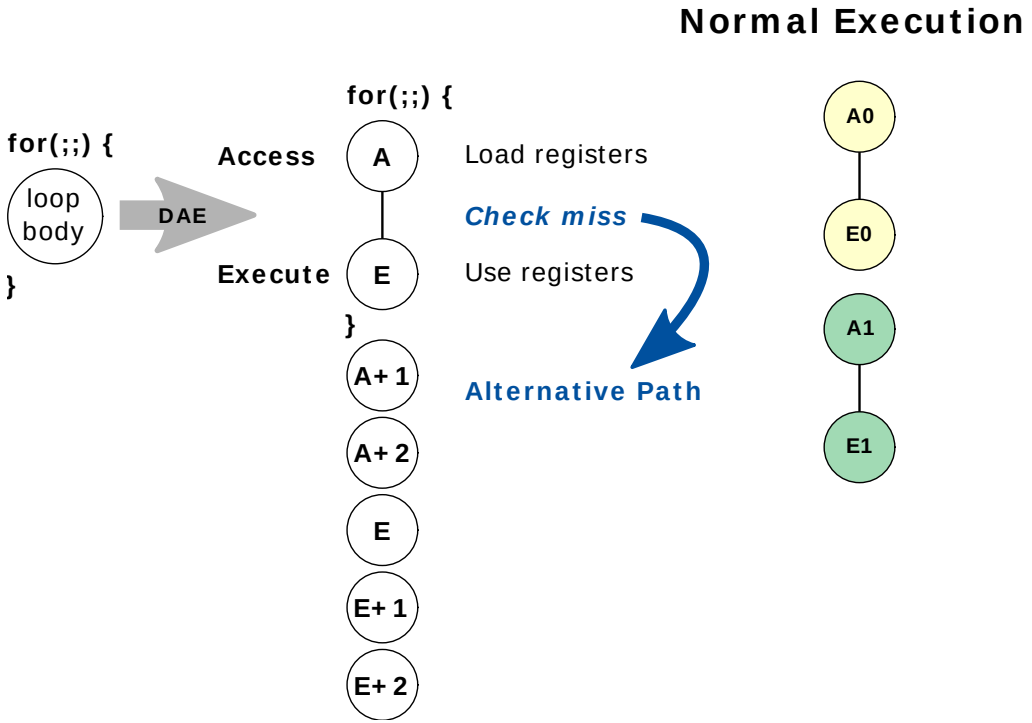



- SWOOP targets loops
- Amenable loops are split into an **Access** and an **Execute** phase (DAE)
- Check miss instruction
- Branch to alternative path on load miss

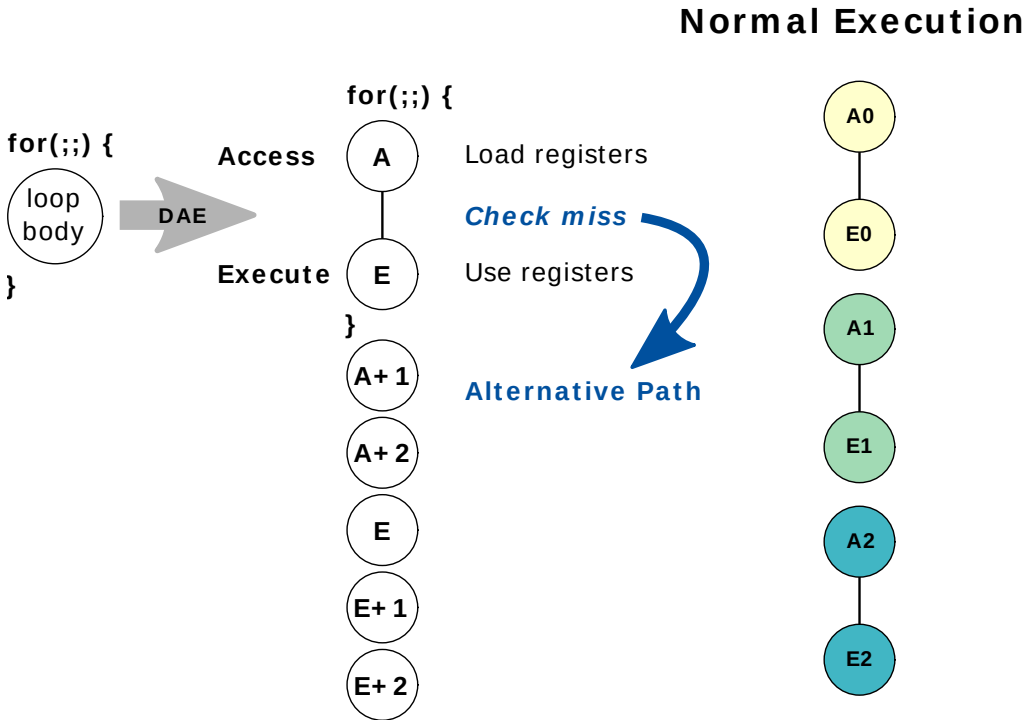
# SWOOP Software/Execution Model



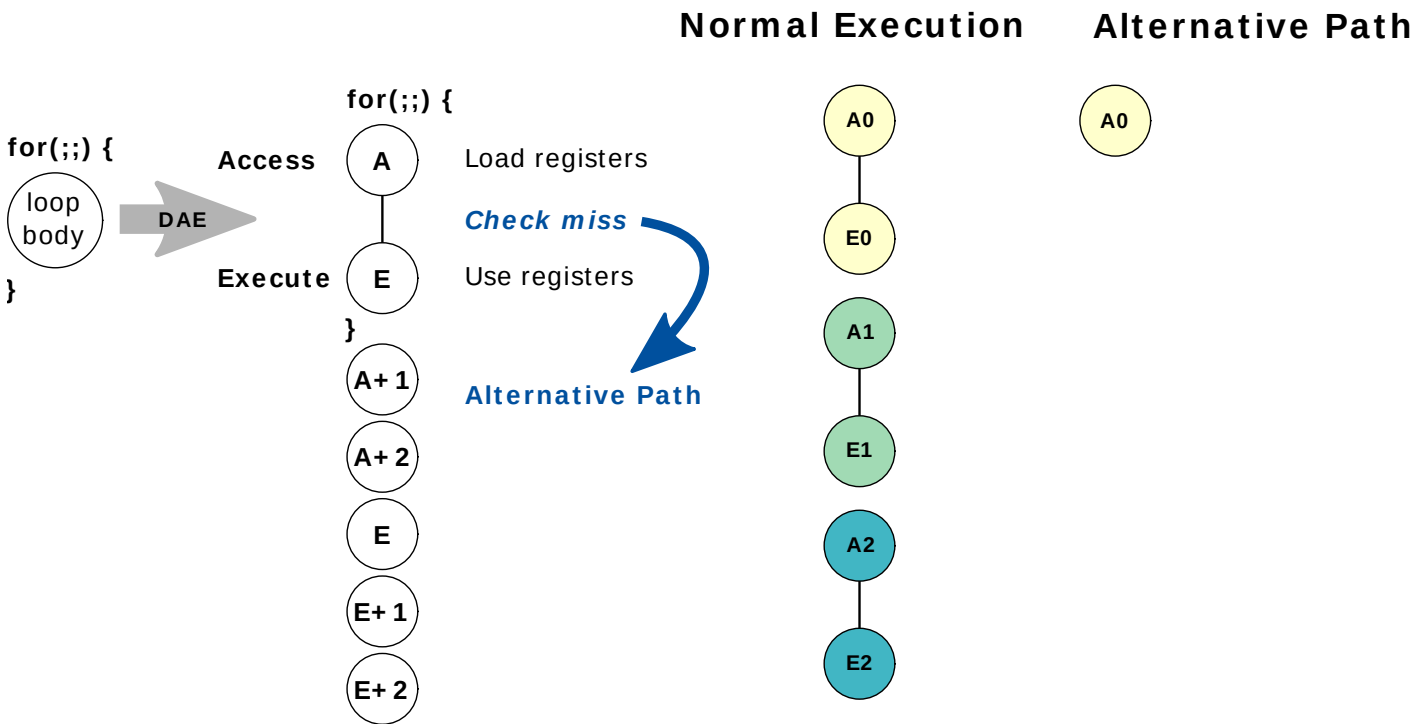
# SWOOP Software/Execution Model



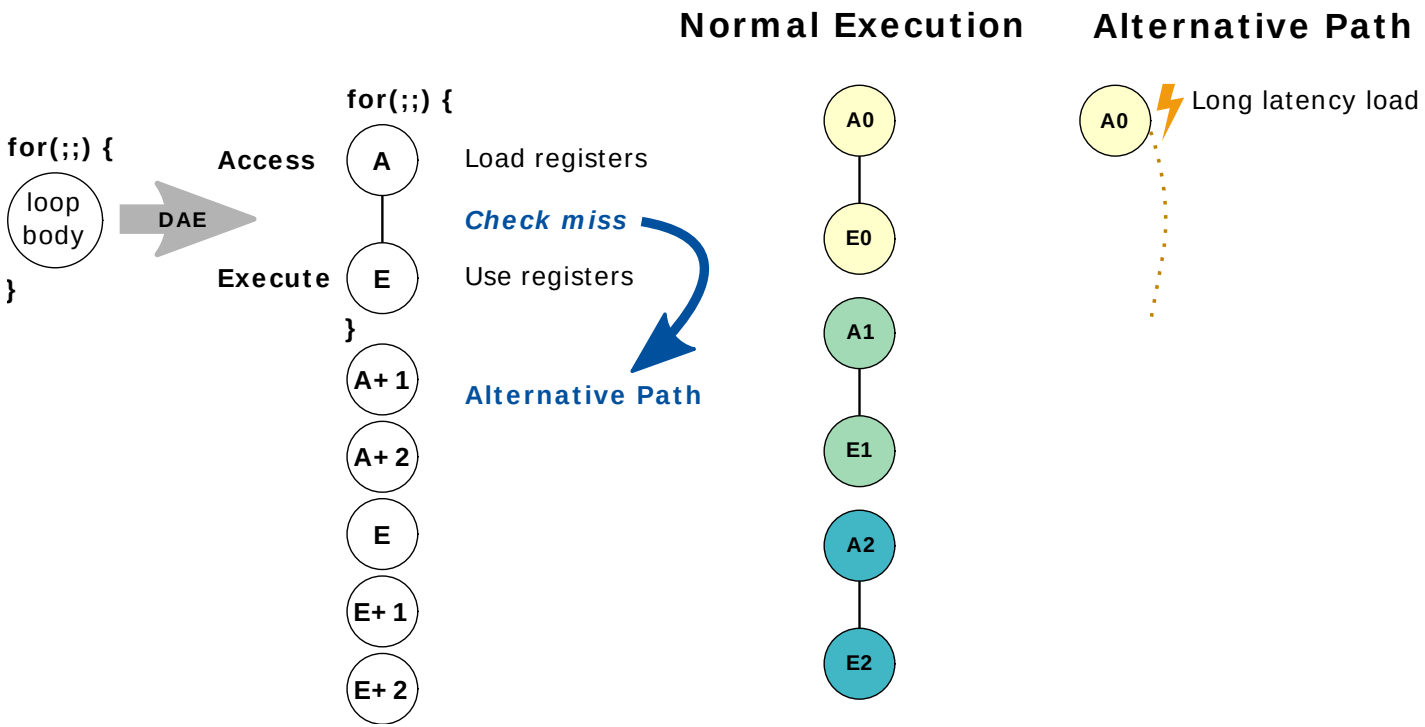
# SWOOP Software/Execution Model



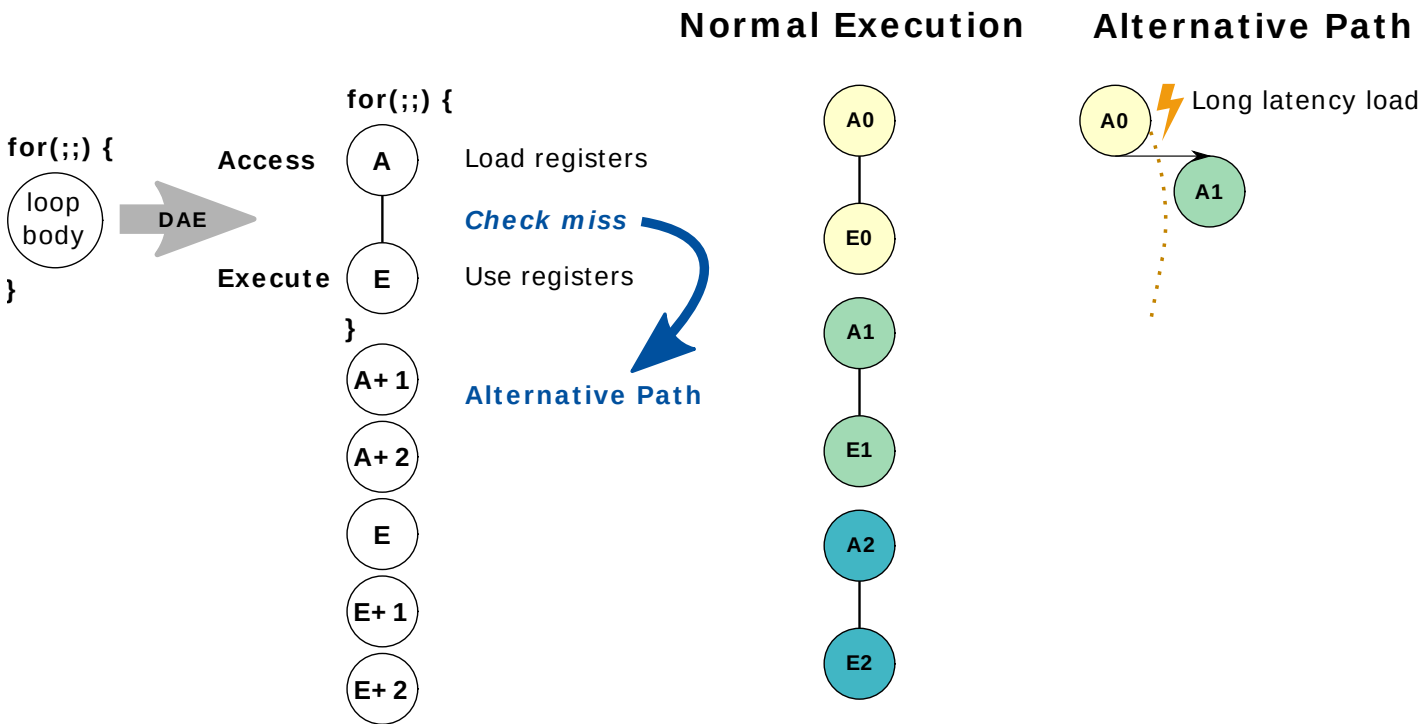
# SWOOP Software/Execution Model



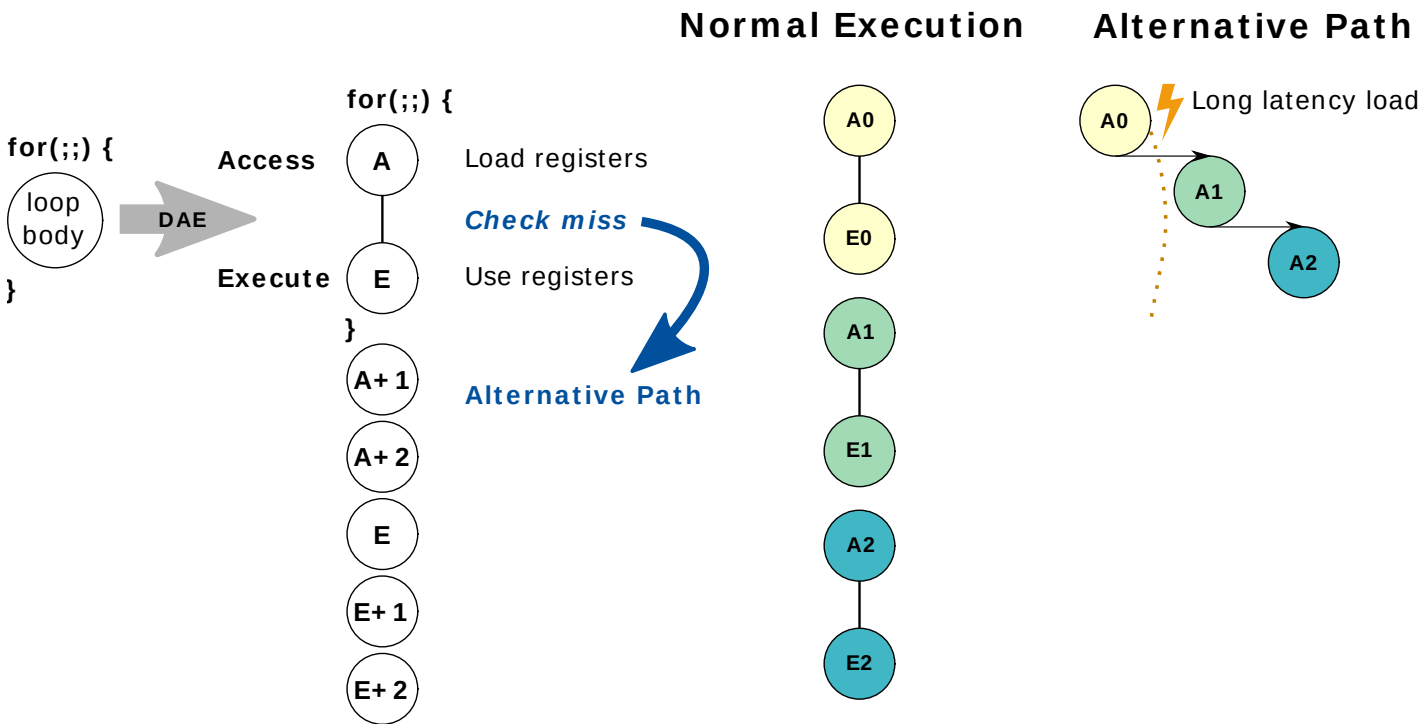
# SWOOP Software/Execution Model



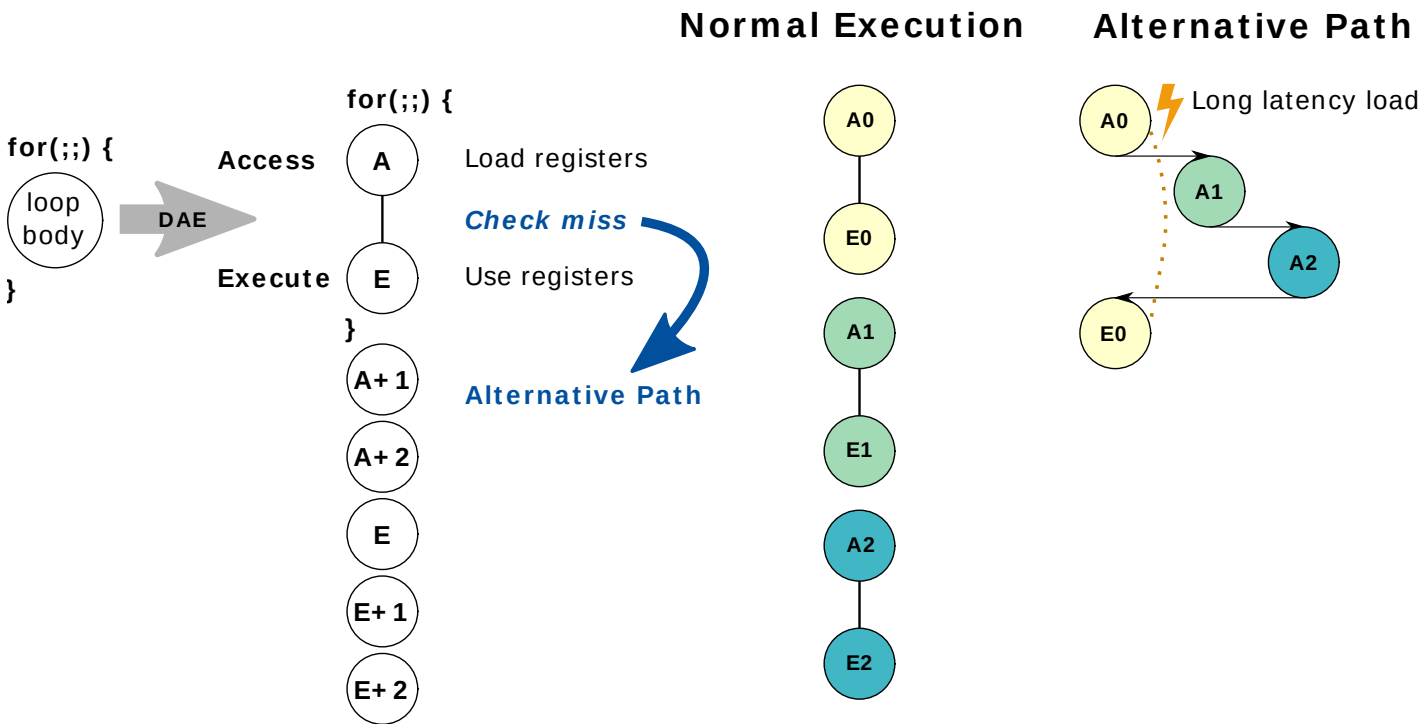
# SWOOP Software/Execution Model



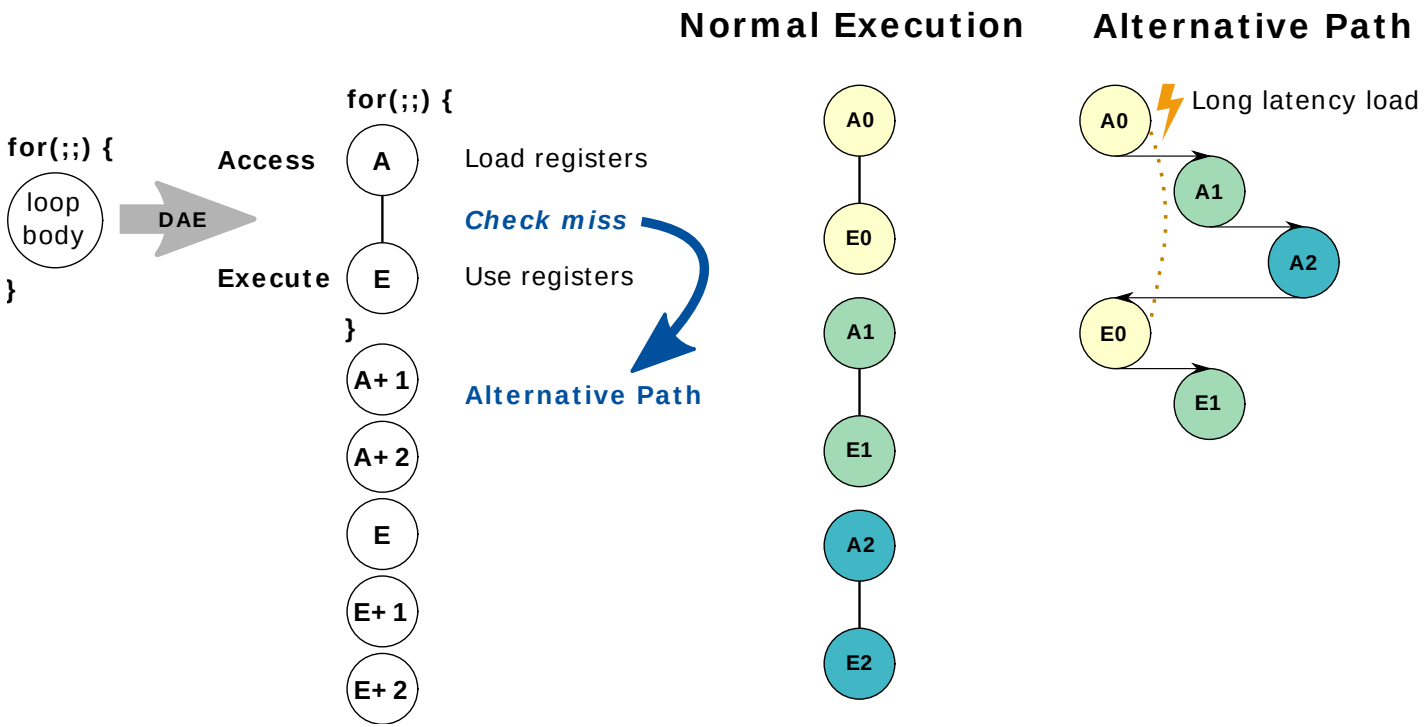
# SWOOP Software/Execution Model



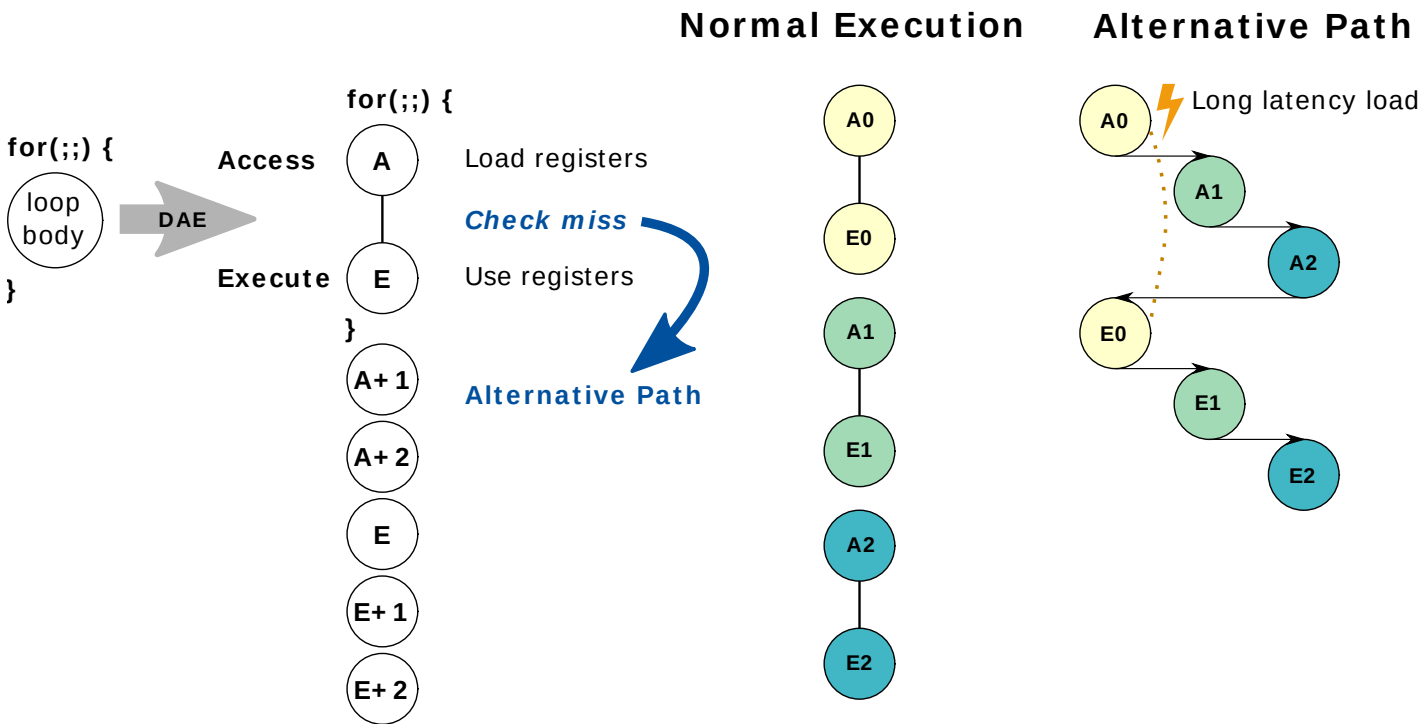
# SWOOP Software/Execution Model



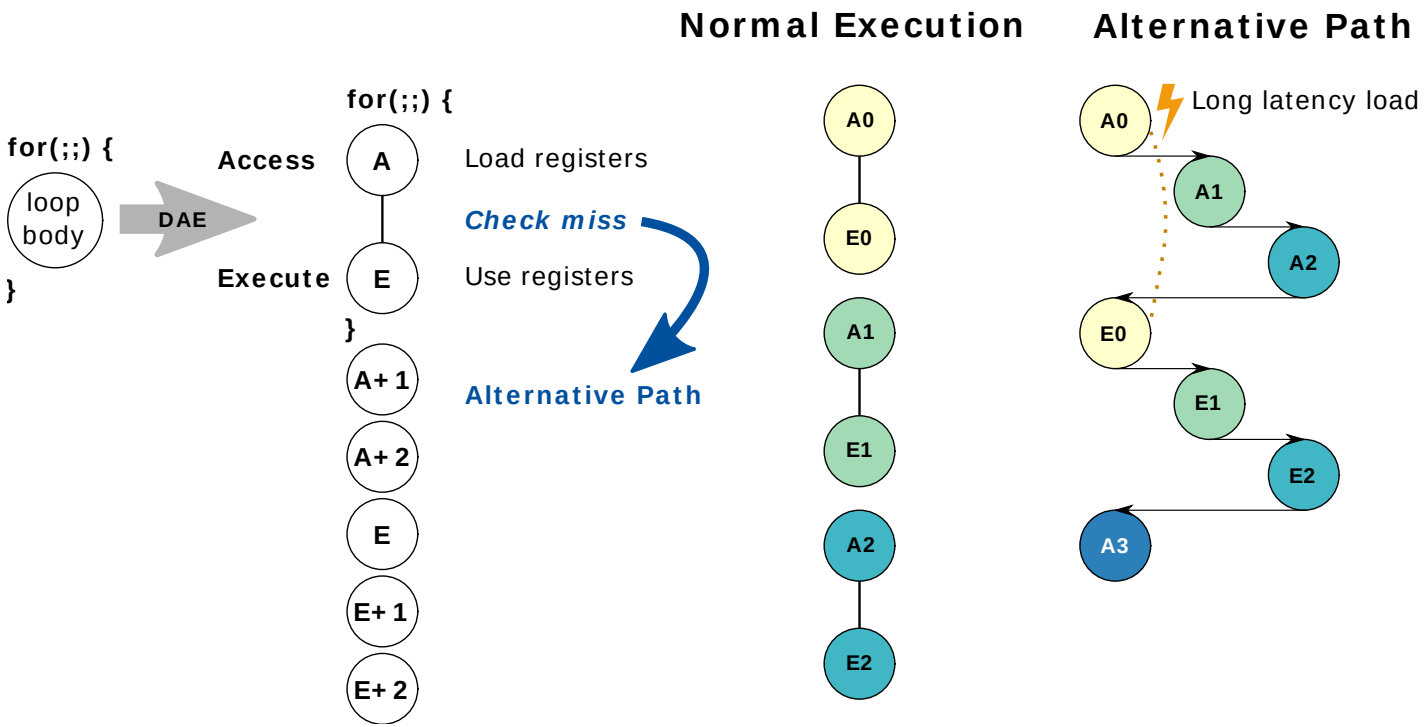
# SWOOP Software/Execution Model



# SWOOP Software/Execution Model



# SWOOP Software/Execution Model



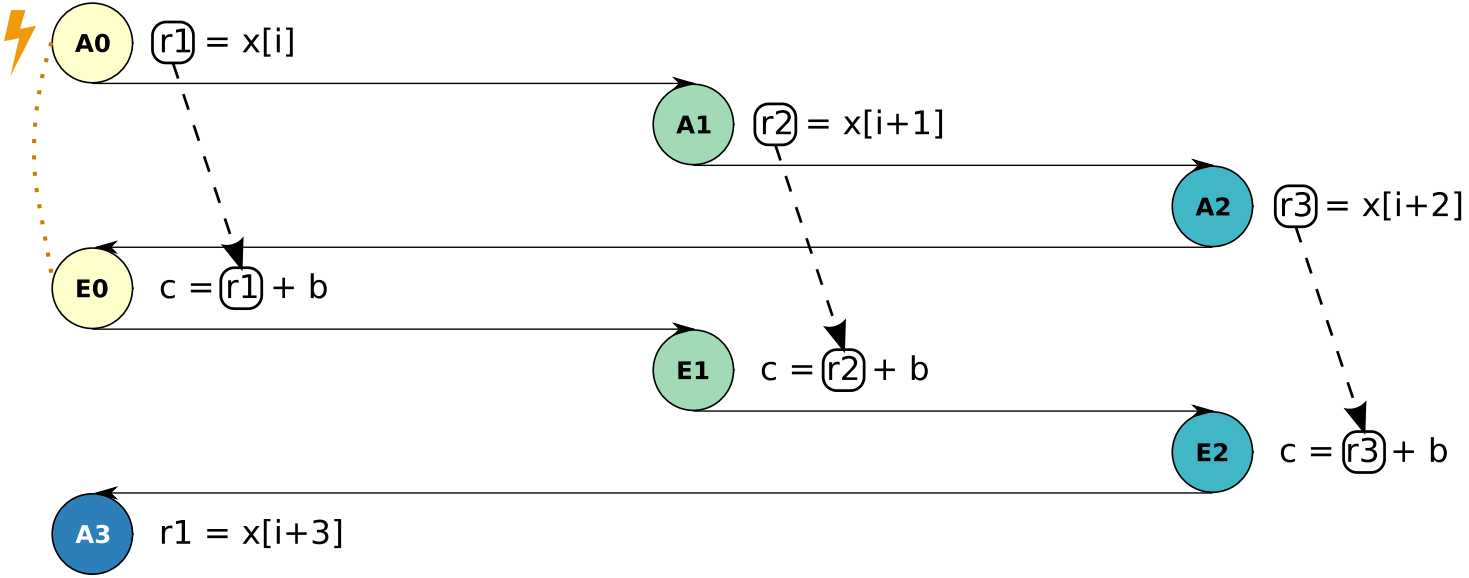
# Challenges

Creating efficient decoupled access execute phases

- memory dependencies
- complex control flow
- register pressure

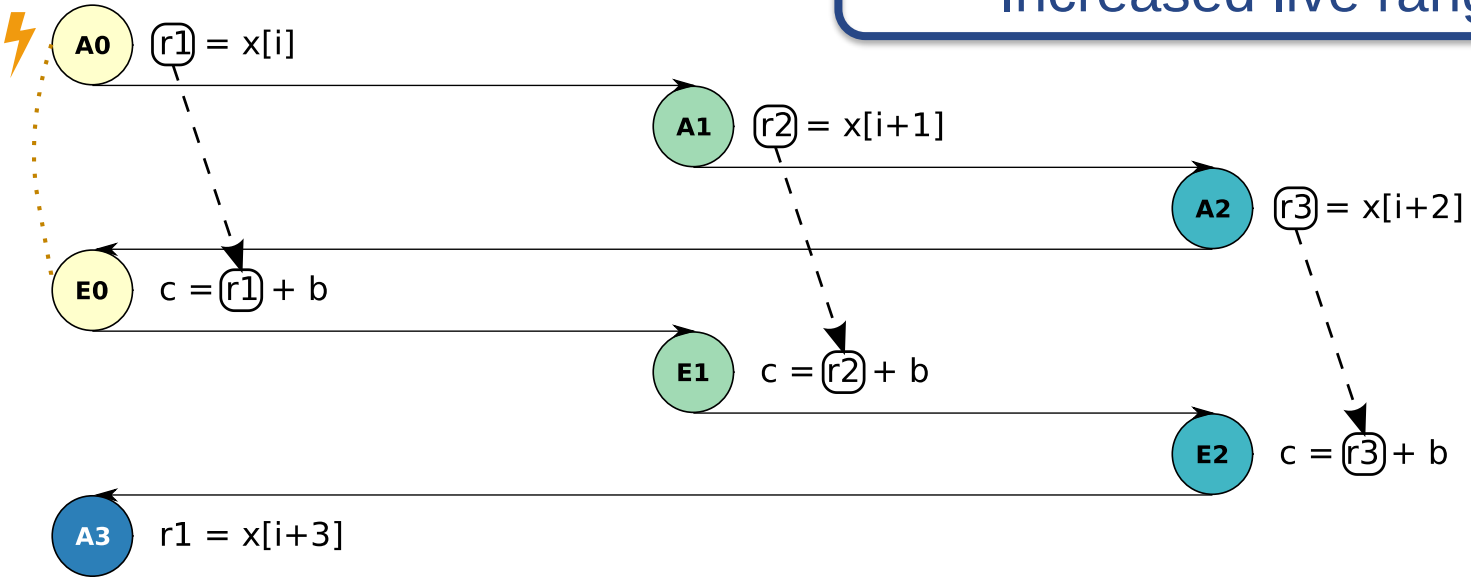
while being non-speculative and minimizing code duplication

# Increased Register Pressure



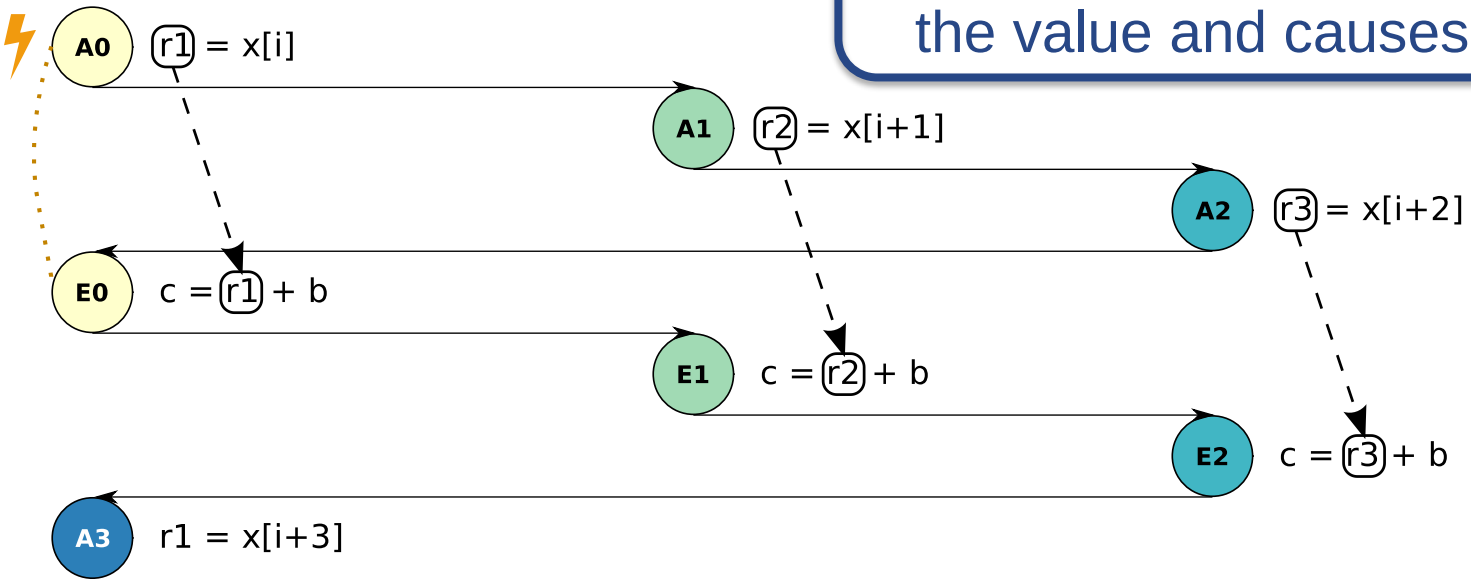
# Increased Register Pressure

Increased number of registers
Increased live ranges



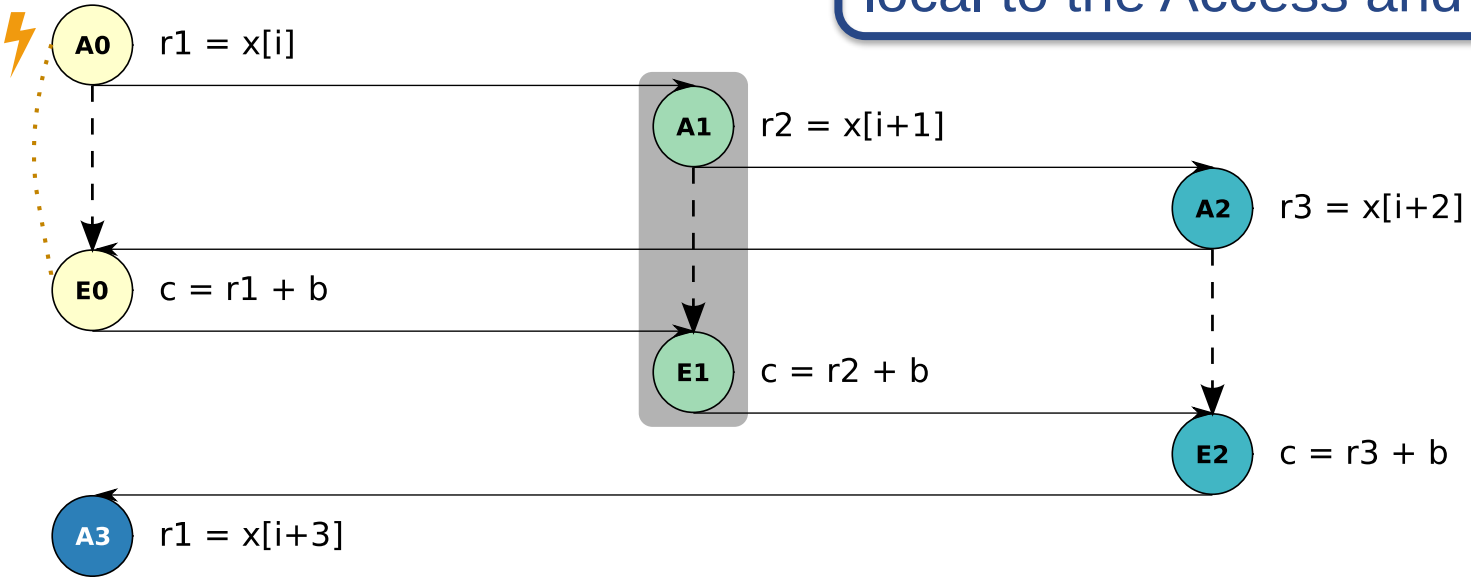
# Increased Register Pressure

A register spill stores the value to the stack, which is a use of the value and causes a stall

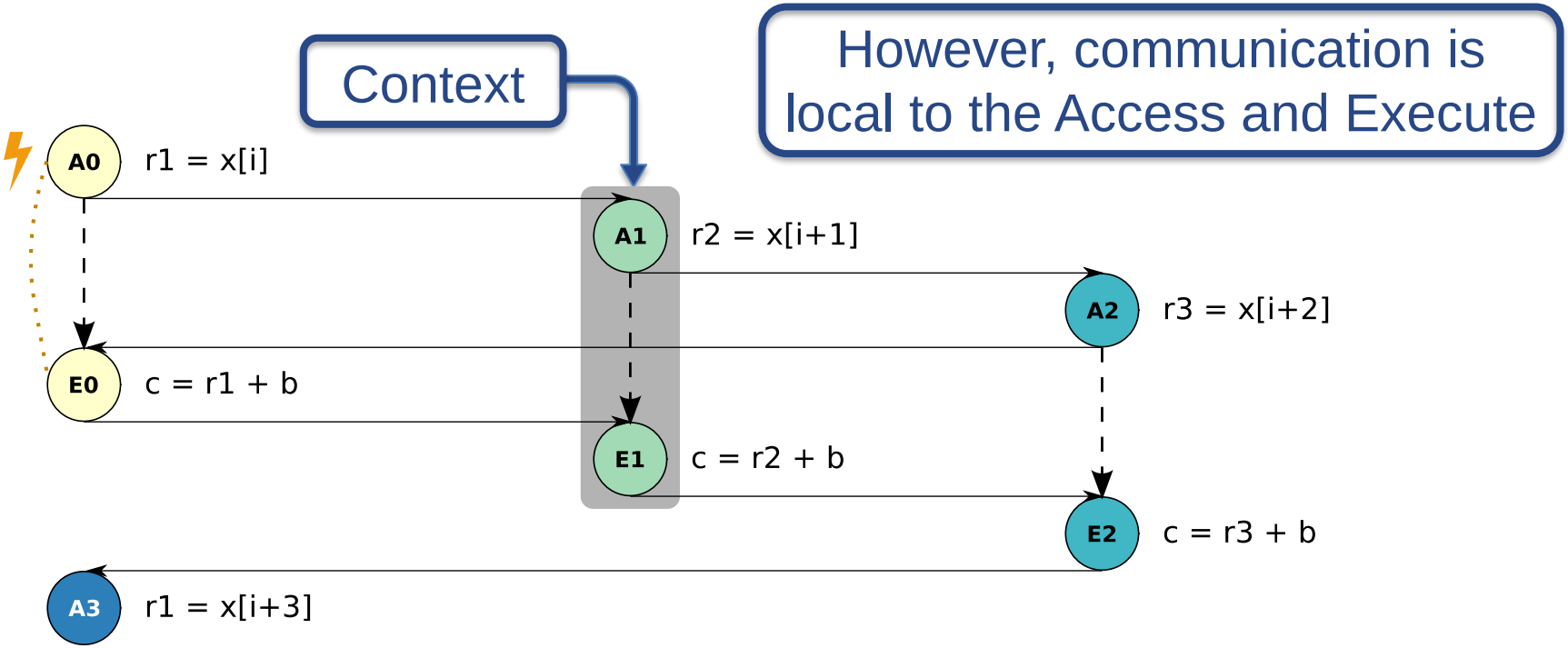


# Increased Register Pressure

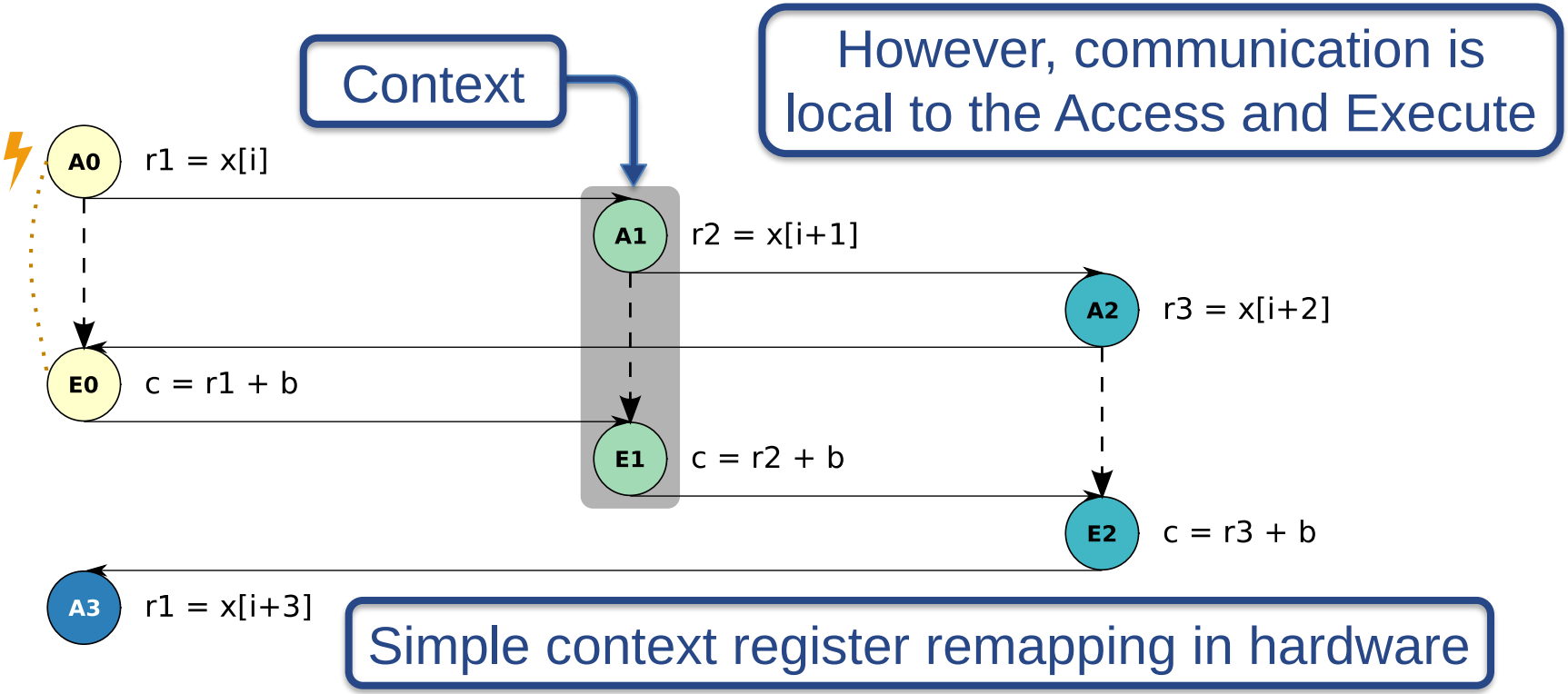
However, communication is local to the Access and Execute



# Increased Register Pressure



# Increased Register Pressure



# Context Register Remapping

- A hardware-software solution to reduce register pressure
- Only registers written in an Access phase are remapped
- A register is remapped at most once per Access phase
- Contexts are software controlled

- Avoids increasing the register pressure
- Significant reduction of remappings compared to OoO

# Evaluation Setup

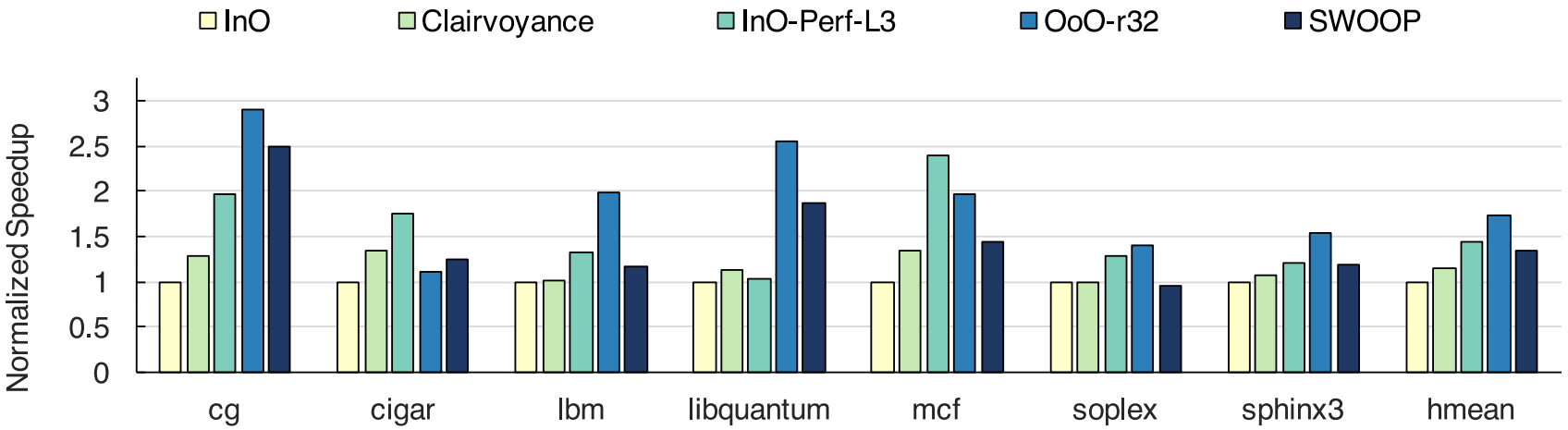
- SWOOP compiler based on LLVM
- Sniper Multi-Core Simulator + McPAT

- SWOOP
  - Based on an in-order core similar to an ARM Cortex A7
  - One extra pipeline stage to model context remapping

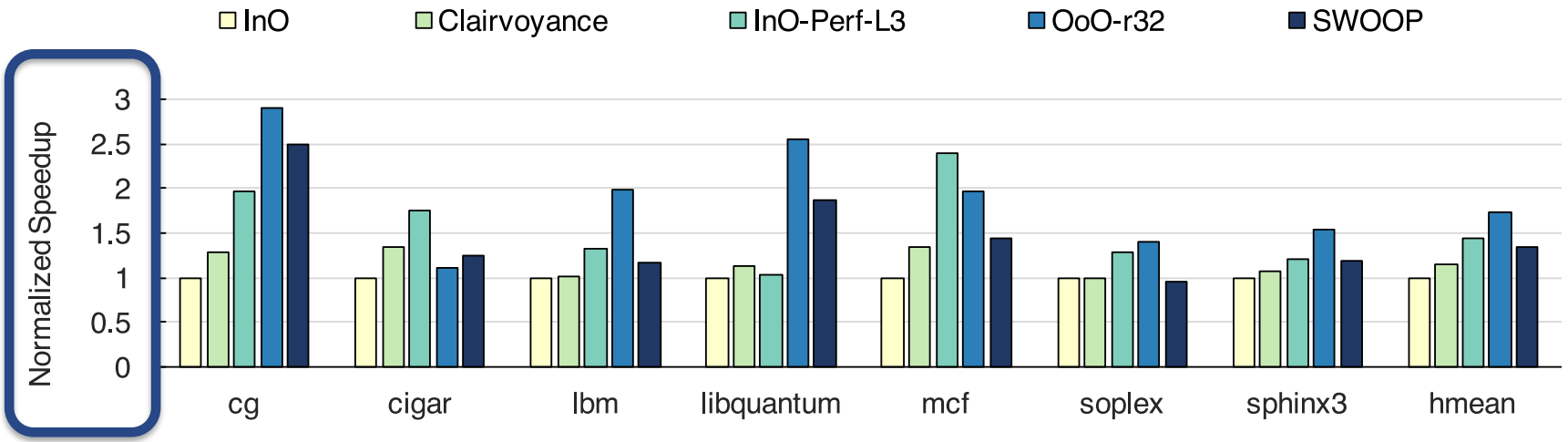
- Compared against
  - In-Order (similar to a Cortex A7)
  - In-Order with perfect L3 cache (no main memory accesses)
  - Out-of-Order (similar to a Cortex A15)
  - Clairvoyance (software-only solution — CGO 2017)

- SPEC CPU 2006, CIGAR, and NAS benchmark suites (high MPKI)

# Performance

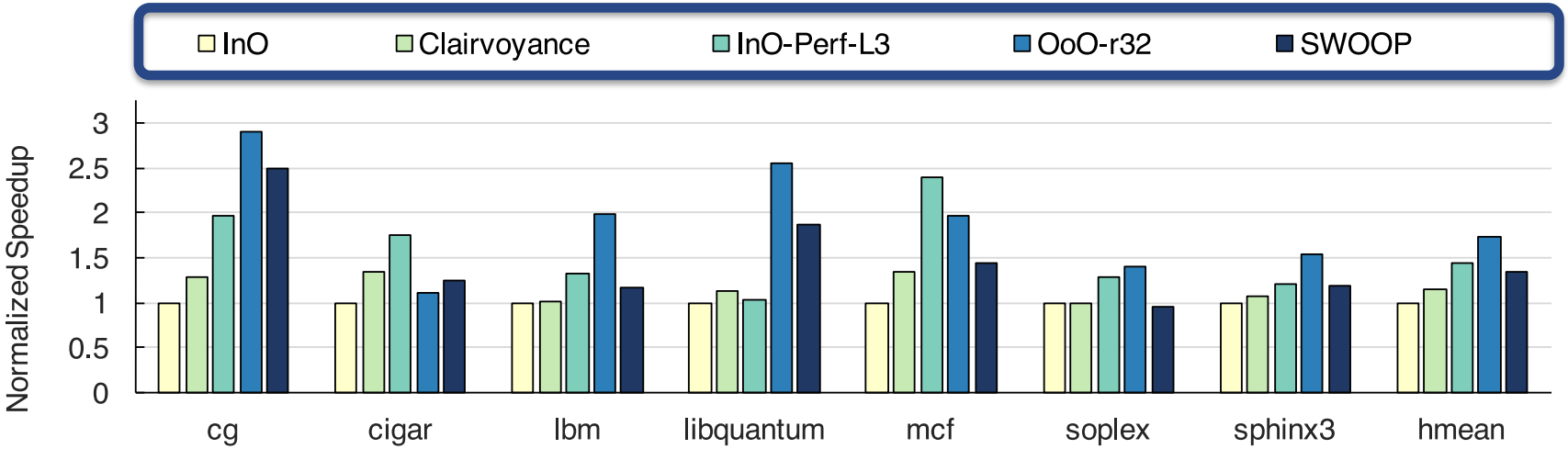


# Performance

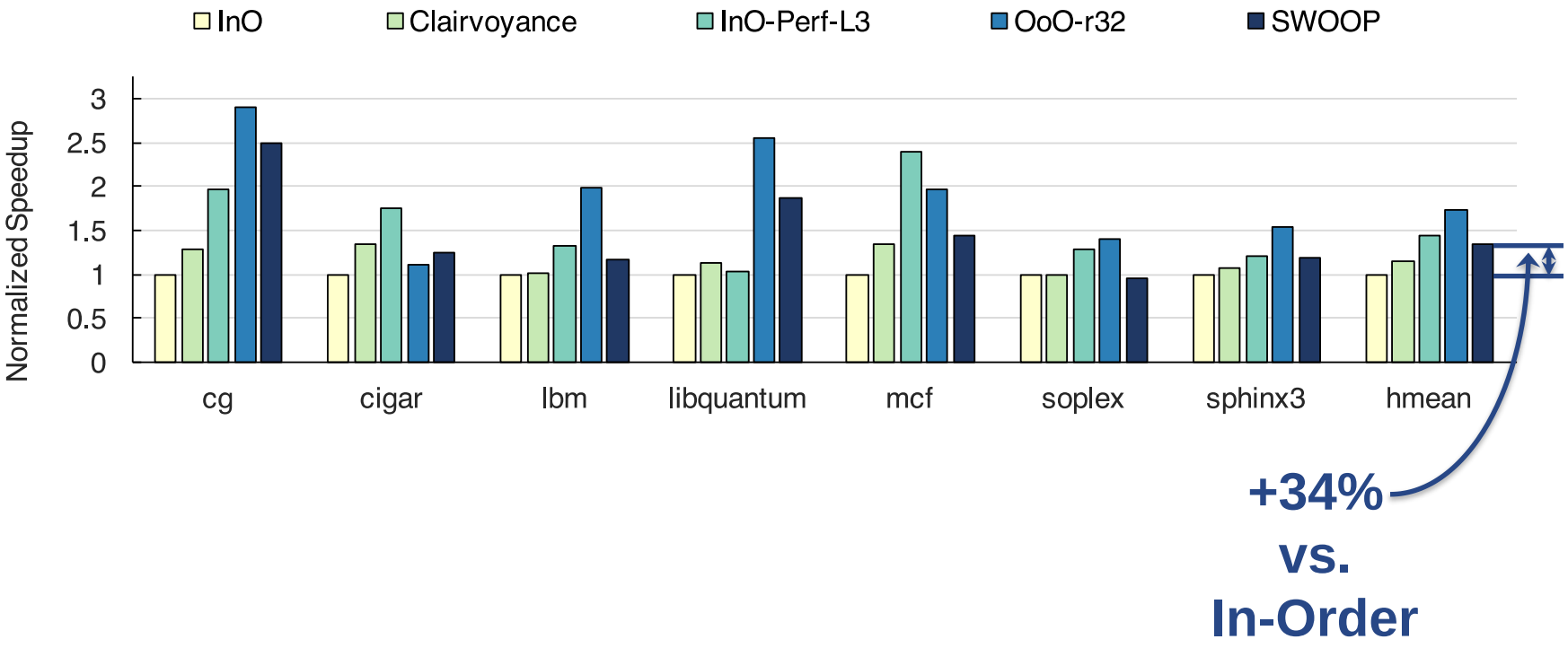



**Higher is better**

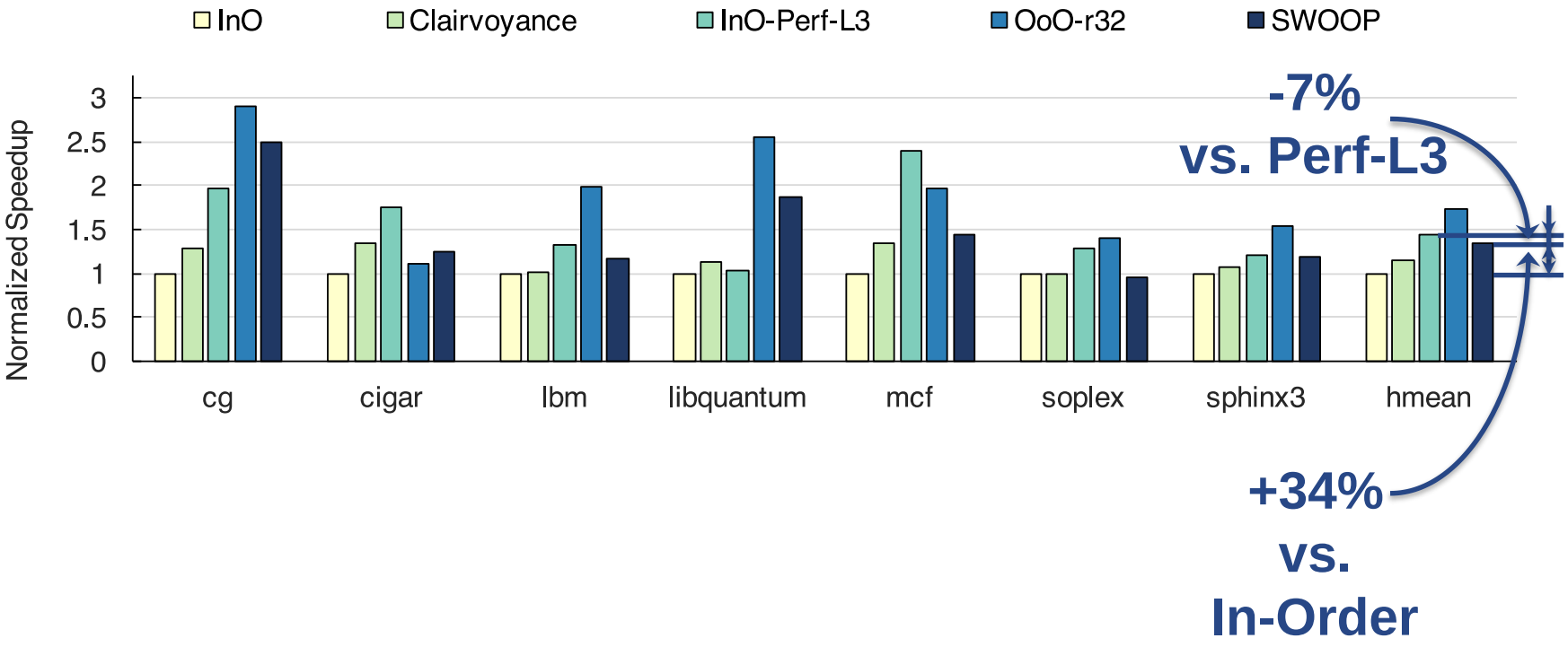
# Performance



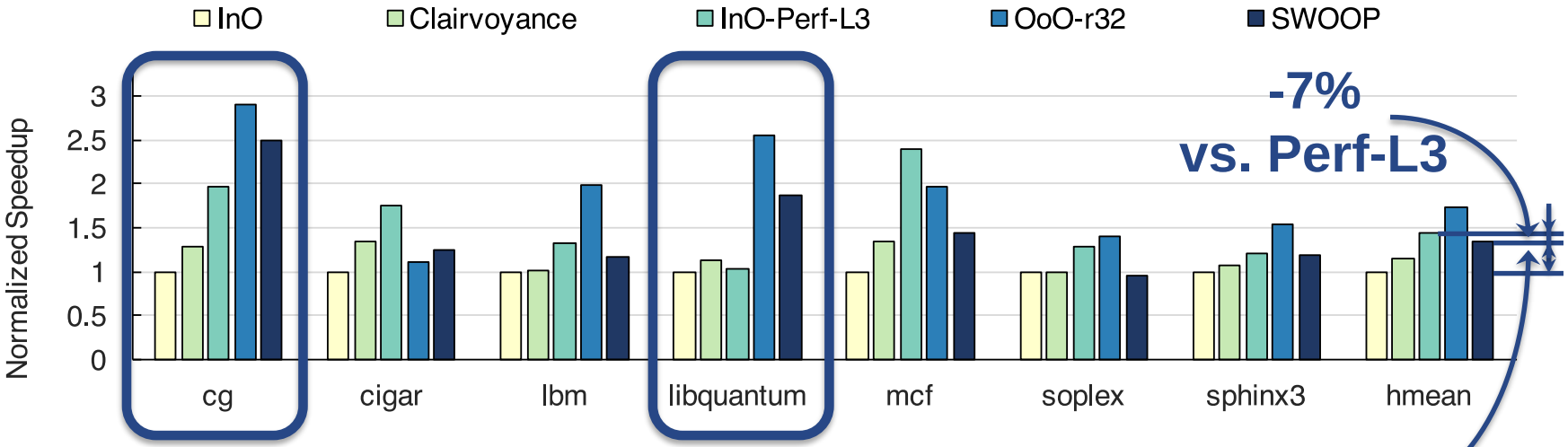
# Performance



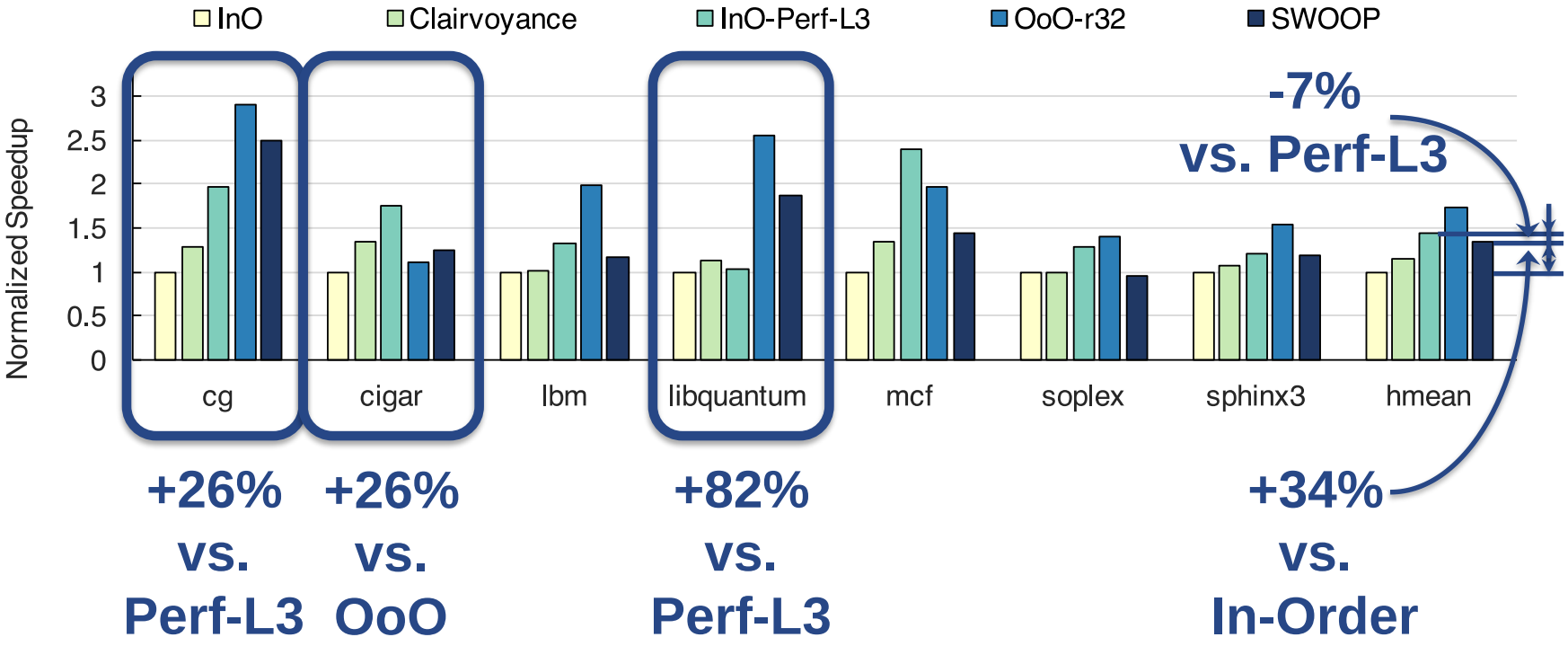
# Performance



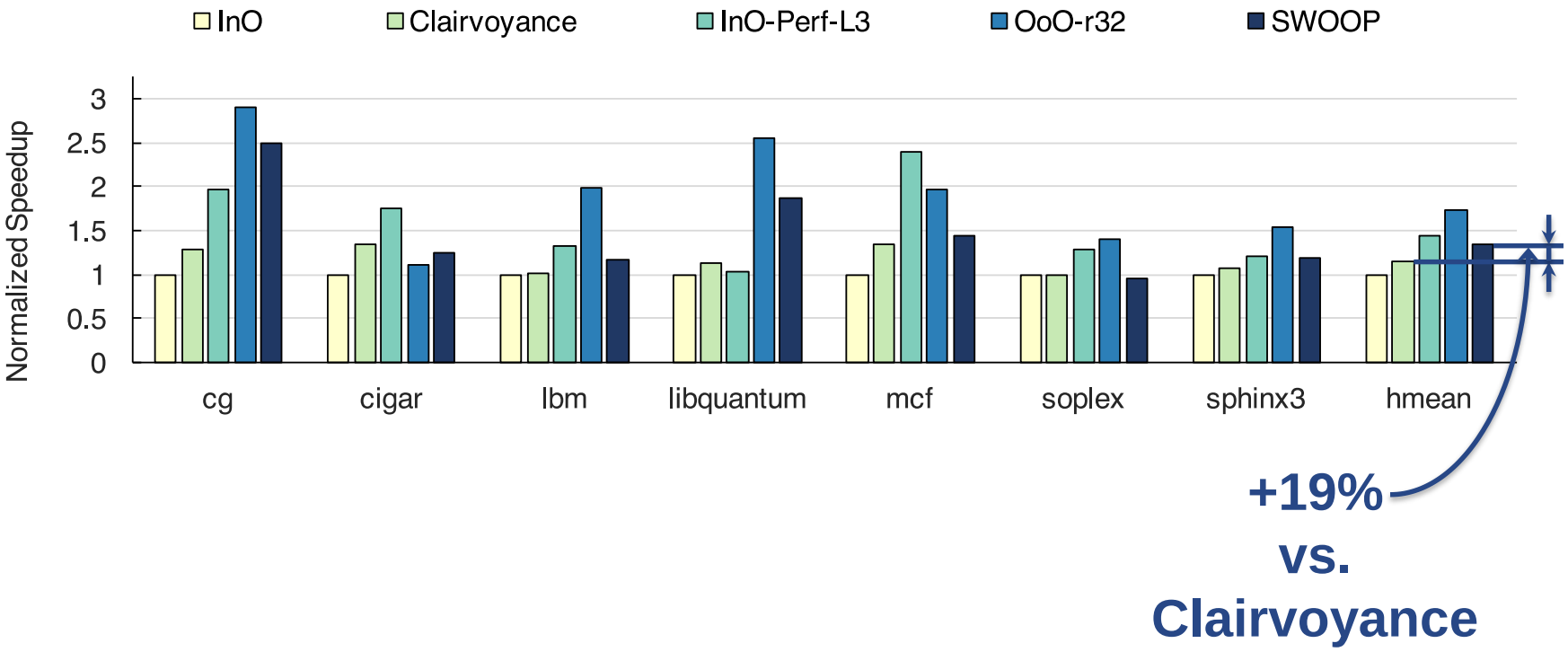
# Performance



# Performance

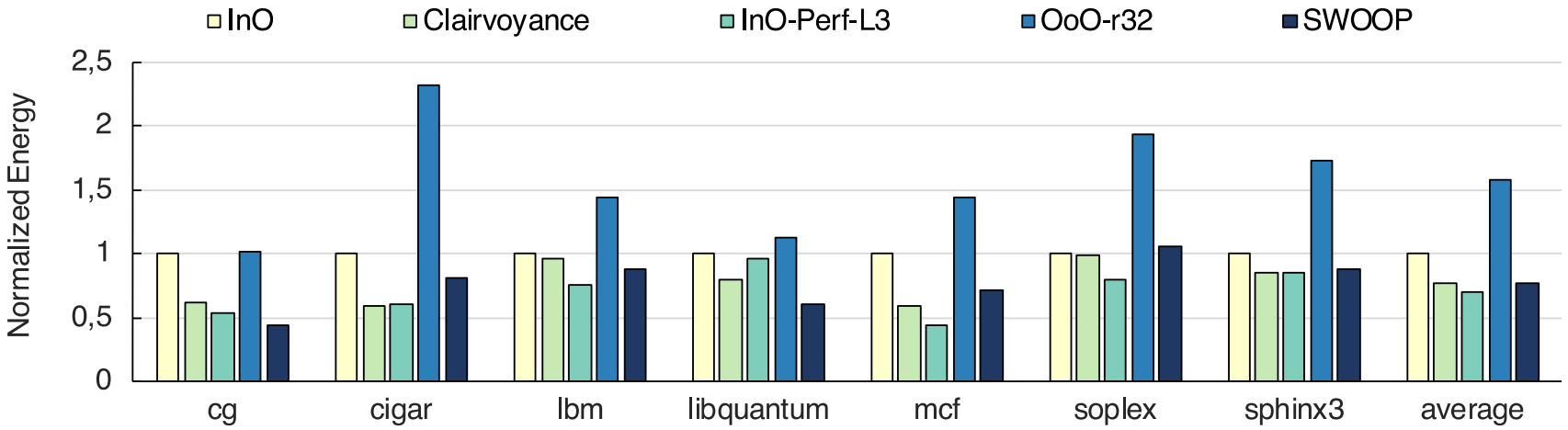


# Performance

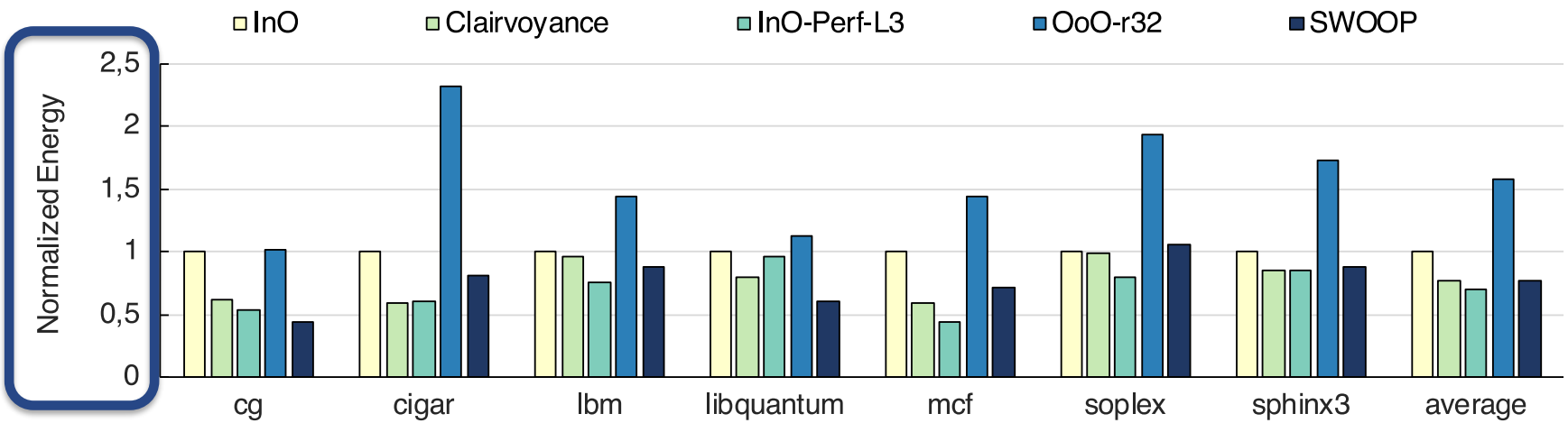



[1] K.-A. Tran et al., Clairvoyance: Look-ahead compile-time scheduling, CGO 2017

# Energy Efficiency

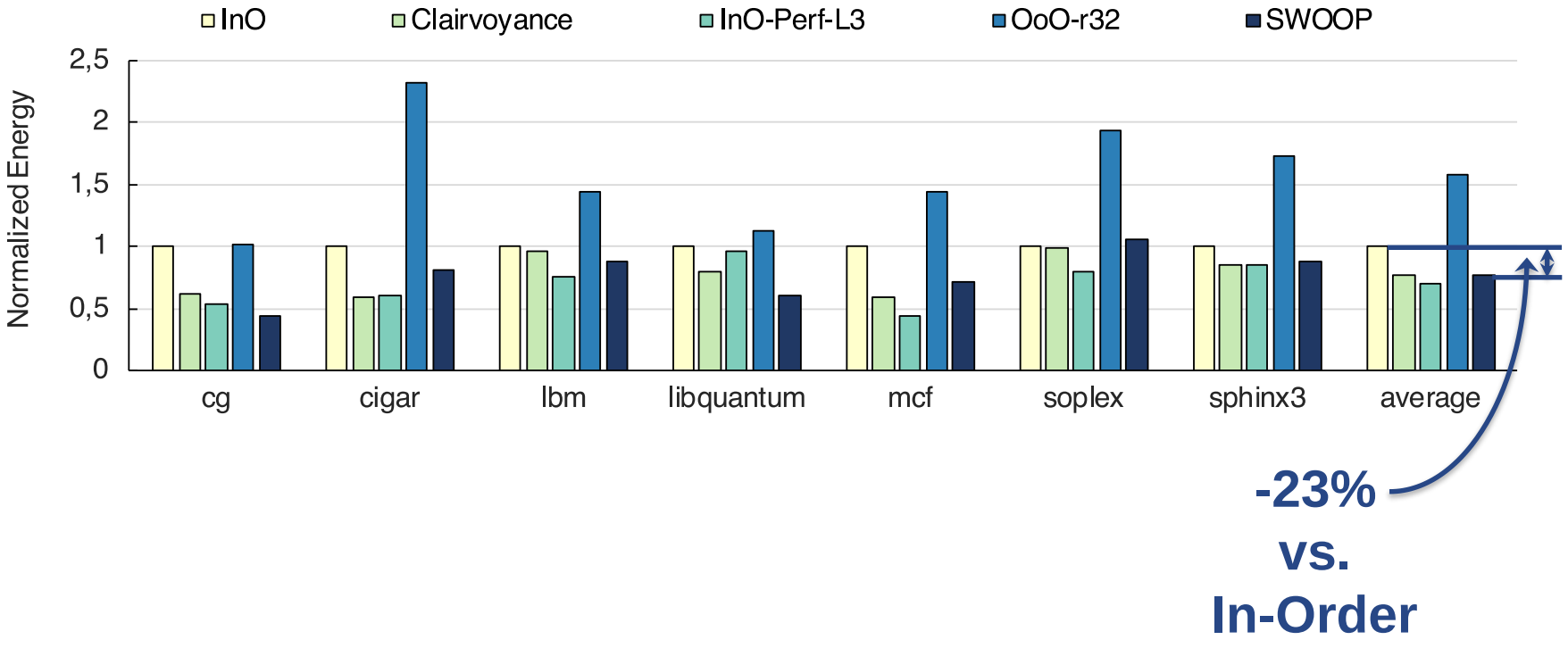


# Energy Efficiency

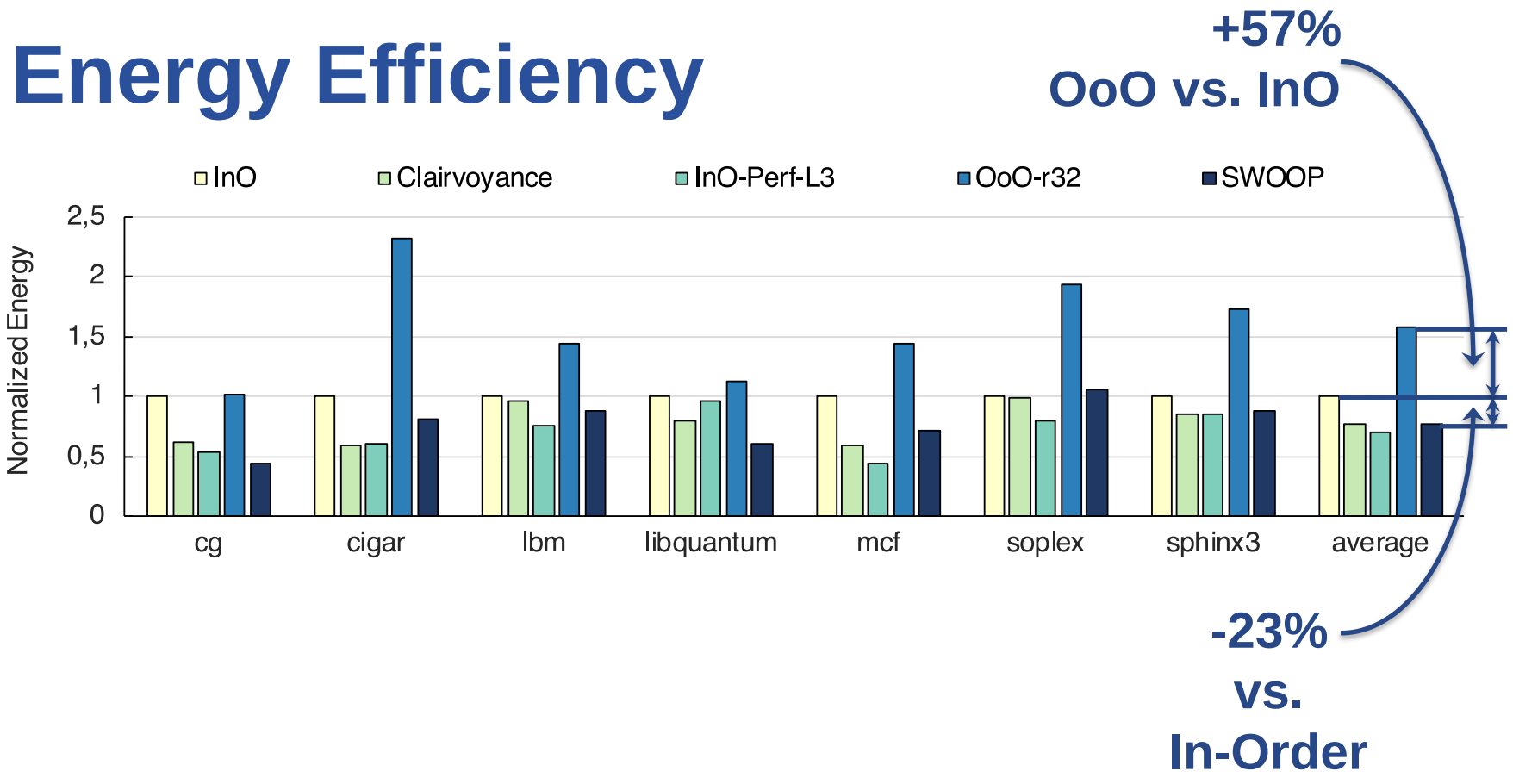



**Lower is better**

# Energy Efficiency



# Energy Efficiency



# Summary

- SWOOP is a non-speculative hardware-software design
- SWOOP jumps ahead to independent regions of code
- 34% performance improvement over an In-Order core
- 23% energy reduction over an In-Order core

# 2. Software Programmable Accelerator

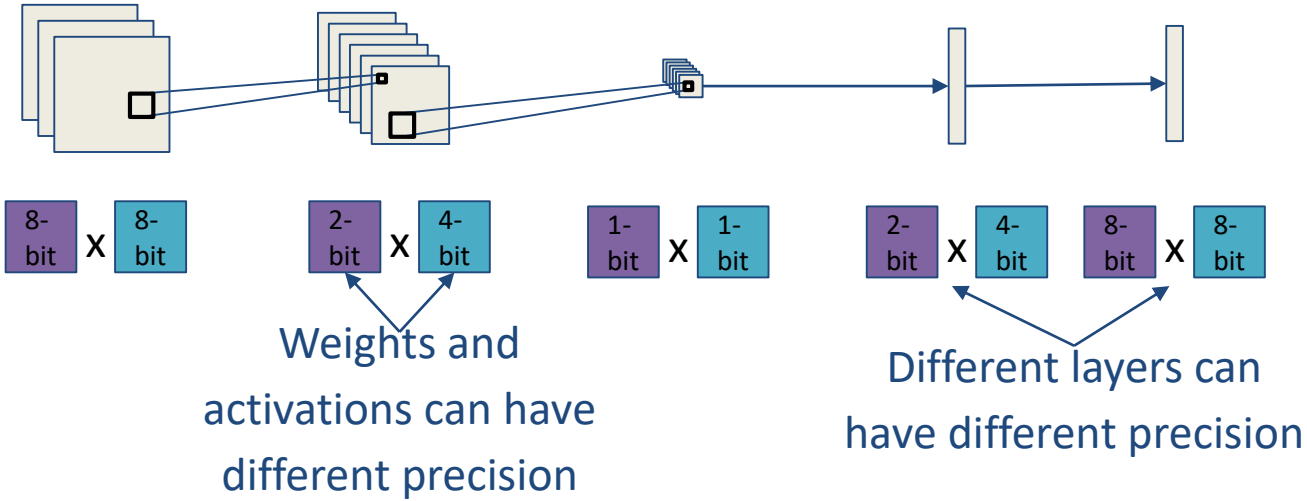
BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing

Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander

FPL 2018

# The Need for Variable-Precision Matrix Multiply

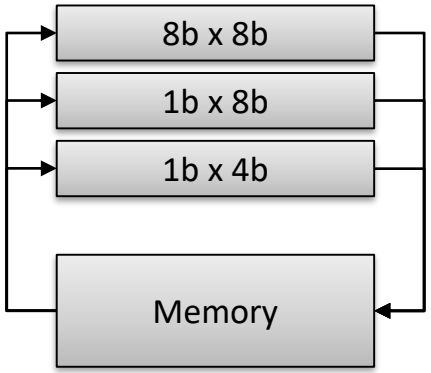
- Not all applications need the same arithmetic precision
- Example: mixed-precision in Quantized Neural Networks (QNNs)



# Hardware for Variable Precision

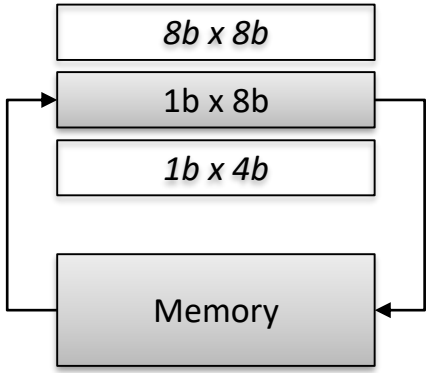
- HW typically **fixed precision**, SW benefits from **variable precision**
- Mismatch causes power, performance, and area overheads
- Potential solutions to the problem

**Cover-All Datapath**



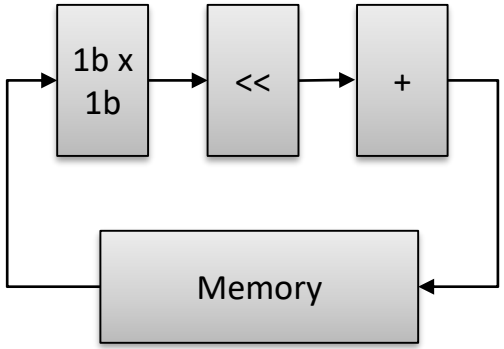

Under-utilization
Need to pick the precisions

**Dynamic Reconfiguration**




Reconfiguration overhead

**Bit-Serial Datapath**




One datapath that can do all precisions

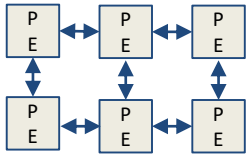
# Approach: Bit-Serial Matrix Multiplication

$$L = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 10 & 00 \\ 00 & 11 \end{bmatrix} = 2^1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = (2^1 L^{[1]} + 2^0 L^{[0]})$$

**1) Integer matrix = weighted sum of binary matrices**

$$L \cdot R = (2^1 L^{[1]} + 2^0 L^{[0]}) \cdot (2^1 R^{[1]} + 2^0 R^{[0]})$$

**2) Integer matrix multiply = weighted sum of binary matrix multiplications**



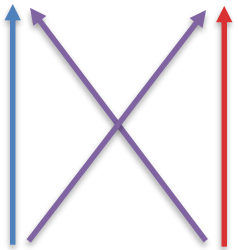

**3) Parallelize inside each binary matrix multiplication for throughput**

# Example: Bit-Serial Matrix Multiplication

$$L = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 10 & 00 \\ 00 & 11 \end{bmatrix} = 2^1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

$2^1 L^{[1]}$ $2^0 L^{[0]}$

**2-bit integer matrix = weighted sum of 2 binary matrices**



$$R = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 00 & 01 \\ 01 & 10 \end{bmatrix} = 2^1 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

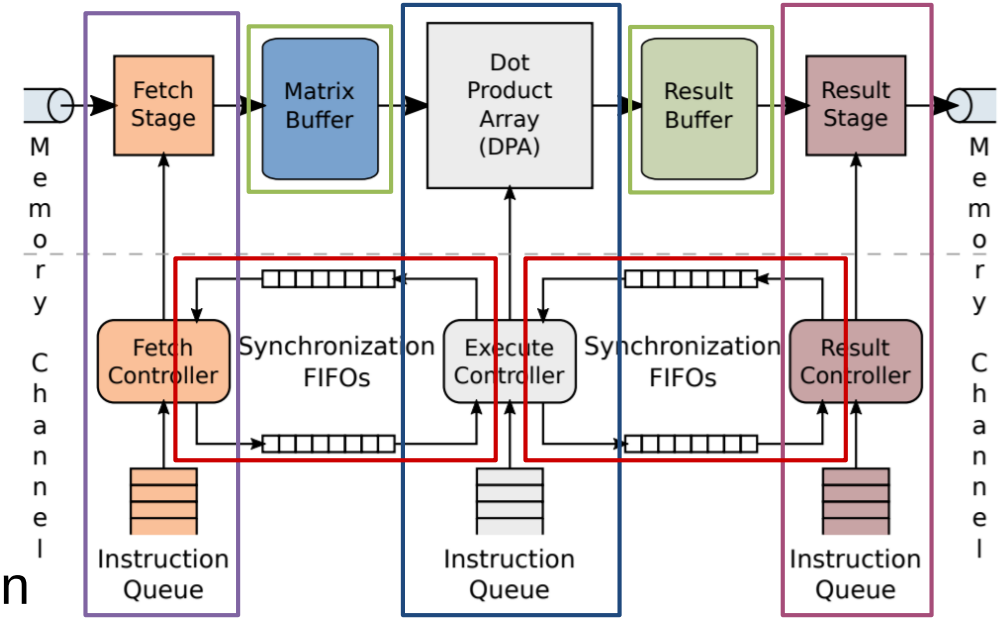
$2^1 R^{[1]}$ $2^0 R^{[0]}$

$$\begin{aligned} P &= L \cdot R \\ &= (2^1 L^{[1]} + 2^0 L^{[0]}) \cdot (2^1 R^{[1]} + 2^0 R^{[0]}) \\ &= 2^2 L^{[1]} \cdot R^{[1]} + \\ &\quad 2^1 L^{[1]} \cdot R^{[0]} + \\ &\quad 2^1 L^{[0]} \cdot R^{[1]} + \\ &\quad 2^0 L^{[0]} \cdot R^{[0]} \end{aligned}$$

**2-bit integer matrix multiply = weighted sum of 4 binary matrix multiplications**

# BISMO Hardware Architecture

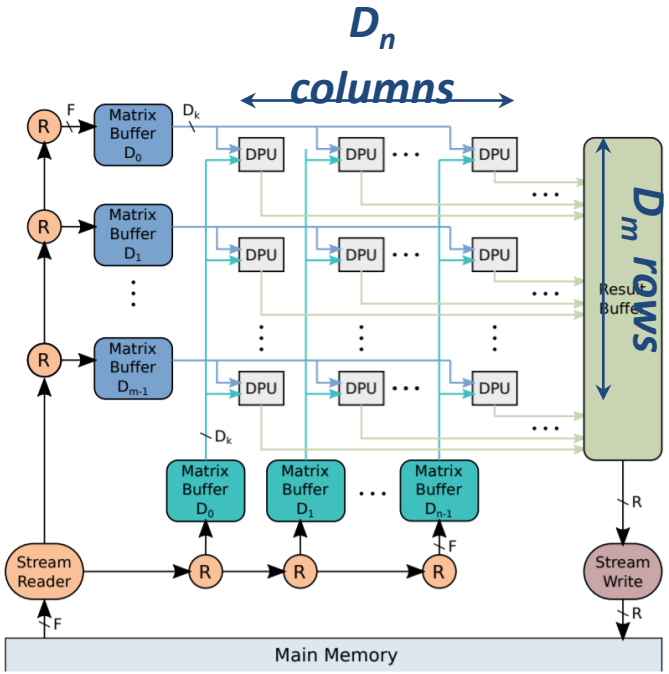
- 3-stage pipeline
  - Fetch: 2D DMA read
  - Execute: Bit-serial mat.mul
  - Result: 2D DMA write
  - All running in parallel
- BRAM matrix buffers
  - Fill/empty via 2D DMA
- Explicit stage synchronization with token FIFOs



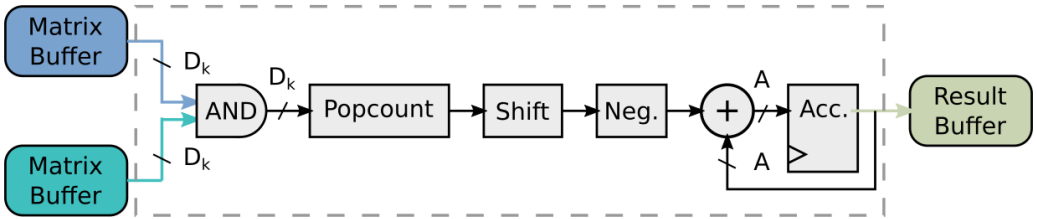
# Execute Stage: Dot Product Array (DPA)

- Array of Dot Product Units (DPU)
  - Parallelizes a single binary matrix multiply
  - 2D broadcast interconnect
  - Configurable array dimensions $D_m$ and $D_n$

- Array fed by matrix buffers
  - Address sequencer reads out matrix data as specified by instruction
  - Matrix buffers filled by 2D DMA



# The Heart of BISMO: Dot Product Unit (DPU)




- Performs weighted binary dot product
  - Power-of-two weight via left shift
  - Optional negation for signed values
- Configurable width $D_k$ for parallel operation
  - $D_k$ binary MACs every cycle

***x*-bit by *y*-bit, *N-element* dot product = *Nxy* binary MACs = *Nxy* / $D_k$ cycles**

# Instruction Set and Programming Model

Send/receive tokens representing shared buffer regions

2D DMA descriptors to move data between DRAM ⇔ BRAM

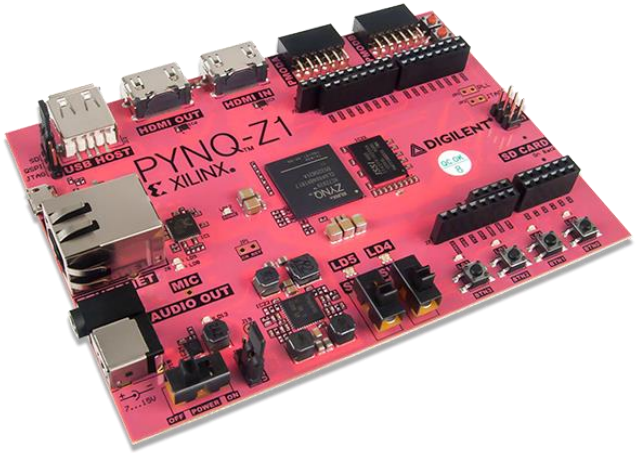
| Instruction type | Fields |
|---|---|
| `Wait & Signal` | Associated FIFO:<br>Fetch stage: Execute<br>Execute stage: Fetch or Result<br>Result stage: Execute |
| `RunFetch` | Source (main memory) parameters:<br>Base address<br>Block size (bytes)<br>Block offset (bytes)<br>Number of blocks to fetch<br>Destination (matrix buffer) parameters:<br>Matrix buffer offset<br>Starting matrix buffer<br>Range of matrix buffers<br>Consecutive words per matrix buffer |
| `RunExecute` | Matrix buffer offset<br>Weight<br>Accumulator reset |
| `RunResult` | Result base address in main memory<br>Address offset |

Run weighted binary dot product on BRAM data and accumulate

- 5-instruction vector ISA
- Supports tiling, bit position skipping, latency hiding ...

# Evaluation

- Prototype implementation
  - Hardware generator in Chisel
  - Minimal software stack in C++
  - Vivado 2017.4 for synthesis
- Hardware platform
  - PYNQ-Z1 board (Zynq Z-7020)
  - USB power meter for the entire PYNQ board
- Evaluated on
  - Resource cost (synthesis results)
  - Throughput, efficiency, power (execution results)



# Performance & Efficiency

| # | $D_m$ | $D_k$ | $D_n$ | LUT | BRAM | GOPS |
|---|---|---|---|---|---|---|
| 1 | 8 | 64 | 8 | 19545 (37%) | 121 (86%) | 1638.4 |
| 2 | 8 | 128 | 8 | 27740 (52%) | 129 (92%) | 3276.8 |
| 3 | 8 | 256 | 8 | 45573 (86%) | 129 (92%) | 6553.6 |
| 4 | 4 | 256 | 4 | 13352 (25%) | 129 (92%) | 1638.4 |
| 5 | 8 | 256 | 4 | 24202 (45%) | 129 (92%) | 3276.8 |
| 6 | 4 | 512 | 4 | 21755 (41%) | 129 (92%) | 3276.8 |

$F = R = 64$ and $F_{\text{clk}} = 200$ MHz unless otherwise stated.

Largest design: 8x256x8 array, 45 kLUTs, **6.5 binary TOPS @200 MHz**

- w-bit by a-bit performance = 6.5 TOPS / (w × a)

# Power

| Configuration (Instance, $F_{clk}$) | Power (W) Idle | Exec | F & R | Full | Binary GOPS | Binary GOPS/W |
|---|---|---|---|---|---|---|
| (#1, 200 MHz) | 2.53 | +0.33 | +1.09 | 4.07 | 1638 | 402.16 |
| (#2, 100 MHz) | 2.10 | +0.19 | +0.87 | 3.11 | 1638 | 527.51 |
| (#3, 50 MHz) | 1.76 | +0.30 | +0.63 | 2.53 | 1638 | 646.39 |
| (#4, 200 MHz) | 2.53 | +0.34 | +1.09 | 3.86 | 1638 | 424.98 |
| (#5, 100 MHz) | 2.05 | +0.24 | +0.92 | 3.06 | 1638 | 536.02 |
| (#3, 200 MHz) | 2.87 | +0.71 | +1.19 | 4.64 | 6554 | 1413.39 |

- Up to **1.4 binary TOPS/W** in power efficiency, including DRAM
- Average power breakdown: 65% idle, 25% fetch/result, 10% execute

# Summary

- Bit-serial matrix multiplication overlay with configurable dimensions
- Up to 6.5 binary TOPS on a PYNQ-Z1
- Runtime proportional to product of matrix precisions
- Software-programmable with a 5-instruction ISA

GitHub

https://github.com/EECS-NTNU/bismo

# 3. A New Intermediate Representation

RVSDG: An Intermediate Representation for Optimizing Compilers

Nico Reissmann, Jan Christian Meyer, and Magnus Själander

Manuscript

# Motivation

- The key to performance in modern architectures is exploitation of parallelism
- This is needed for programming
  - CMPs and GPUs
  - but also for specialization (HLS)
- Thus, we need to be able to easily extract parallelism from programs in order to further improve performance

# Current State

- However, currently used program representations are mostly sequential in nature (e.g., CFG)
- They are a remnant of a time when sequential programming was sufficient and parallelization was done dynamically by the processor

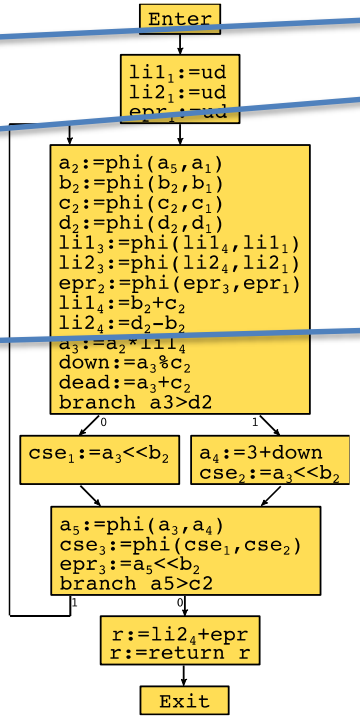
**We need to rethink our models**

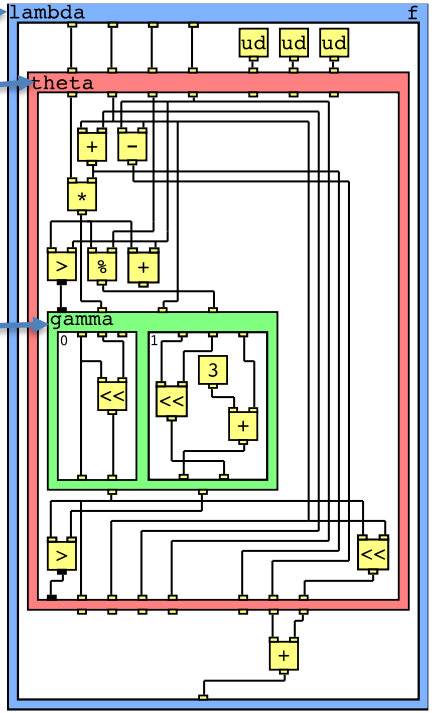
# RVSDG: Regionalized Value State Dependence Graph

- Single unified IR that normalizes program representation
- Exposes hierarchical structure of the program
  - Functions
  - Loops
  - Conditionals
- Explicitly exposes parallelism
  - Not just ILP
  - Functions and loops

# Example of CFG vs. RVSDG

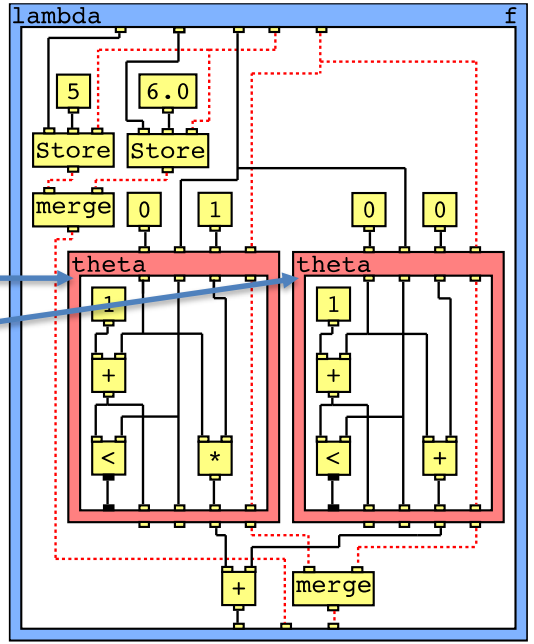
```
int
f(int a, int b, int c, int d)
{
  int li1, li2;
  int cse, epr;
  do {
    li1 = b+c;
    li2 = d-b;
    a = a*li1;
    int down = a%c;
    int dead = a+d;
    if(a > d) {
      int acopy = a;
      a = 3+down;
      cse = acopy<<b;
    } else {
      cse = a<<b;
    }
    epr = a<<b;
  } while(a > cse);
  return li2+epr;
}
```



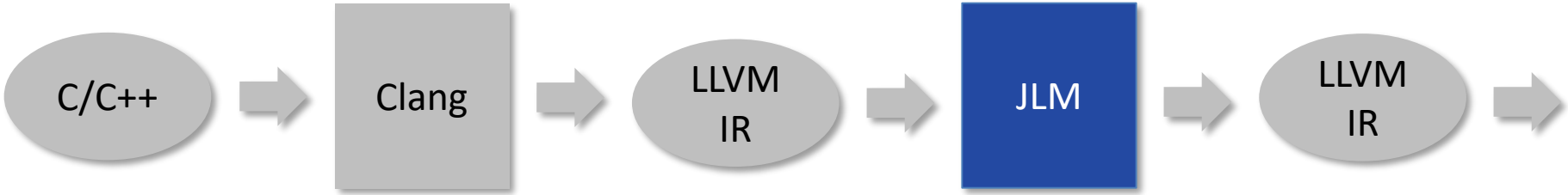
# Parallelism Explicitly Encoded

```
int
f(int* x, float* y, int k)
{
  *x = 5;
  *y = 6.0;
  int i=0, f=1;
  int sum=0, fac=1;
  do {
   sum += i;
   i++;
  } while(i < k);
  do {
    fac *= f;
    f++;
  } while(f < k);
  return fac+sum;
}
```



# The JLM Compiler

- Prototype compiler that uses RVSDG for optimizations
- Plugs into the LLVM compilation flow




- https://github.com/EECS-NTNU/jlm

# Conclusion

# High Efficiency Computing

- Today's computing systems are increasingly power constrained
- Challenges established assumptions and force us to rethink how we construct programs and build systems
- We need a holistic approach, where combined hardware and software optimizations are performed across the complete system stack

# We are hiring

Associate Professor in Compiler Design

PhD graduates and Postdocs as well as more senior researchers are encouraged to apply