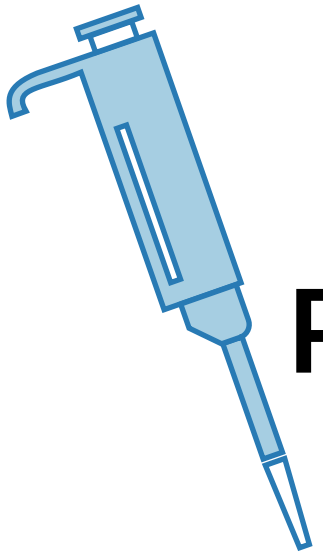


Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism

Quan Nguyen and Daniel Sanchez

Massachusetts Institute of Technology

MICRO-53

Live session: Session 4A: Microarchitecture II

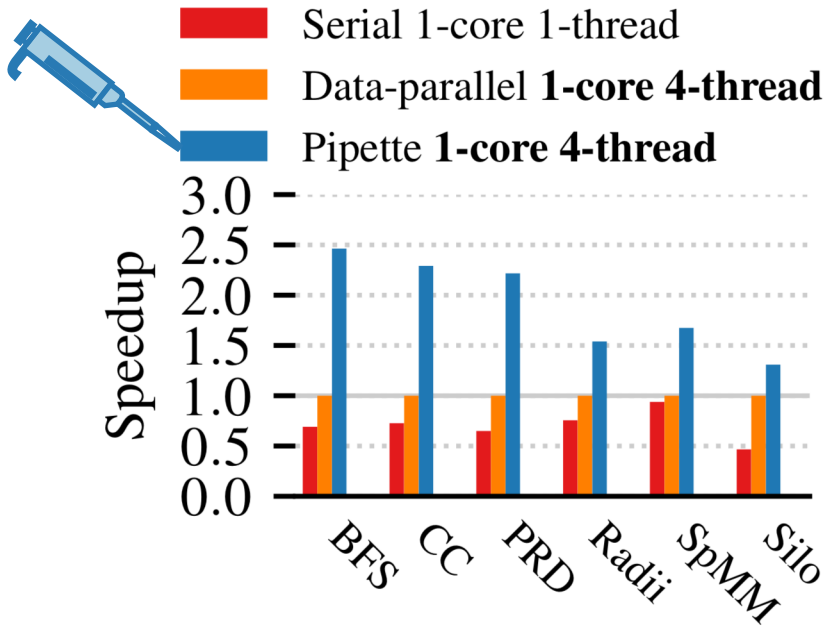
(October 20, 2020 at 2 PM EDT)

Irregular applications hamper core utilization

- Unpredictable memory accesses, control flow make poor use of OOO resources
- Decouple applications into pipeline-parallel stages for **latency tolerance**
- Time-multiplex stages for **load balance**

Reuse OOO core structures to achieve both!

- **Pipette leverages this insight** to implement fine-grain pipeline-parallel communication between threads of an SMT core
- Speedup of gmean 1.9×, up to 3.9× over SMT baseline in challenging irregular benchmarks



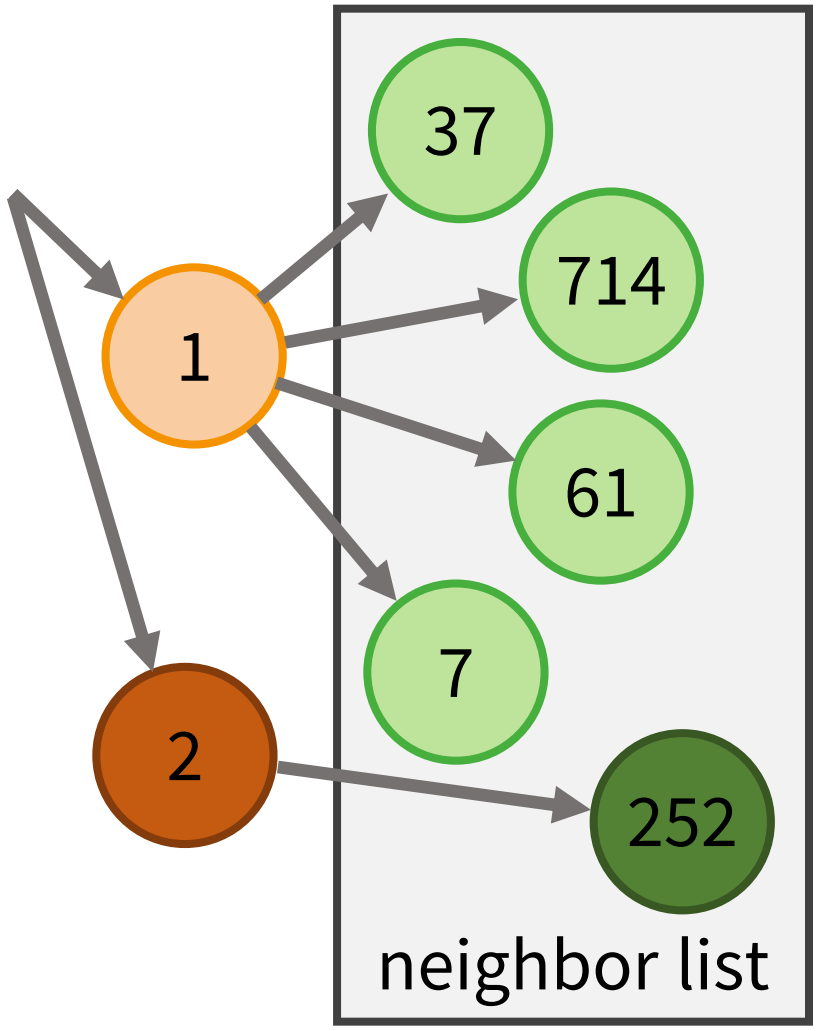
Agenda

Background

Pipette

Evaluation

Unpredictable accesses hurt performance



Example: run `func()` on `array` data
for each neighbor of vertices 1 and 2

```
…
for ngh in neighbor list:
    x = array[ngh]
    func(x)
…
```

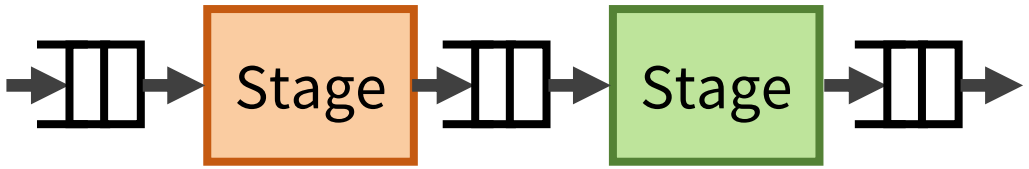
2-3 instrs?

10s, 100s instrs?

If `func()` many instructions, **reorder buffer fills**
Want to **decouple fetches from processing**

Decouple for latency tolerance

- Separate application into many stages, allowing producers to run ahead

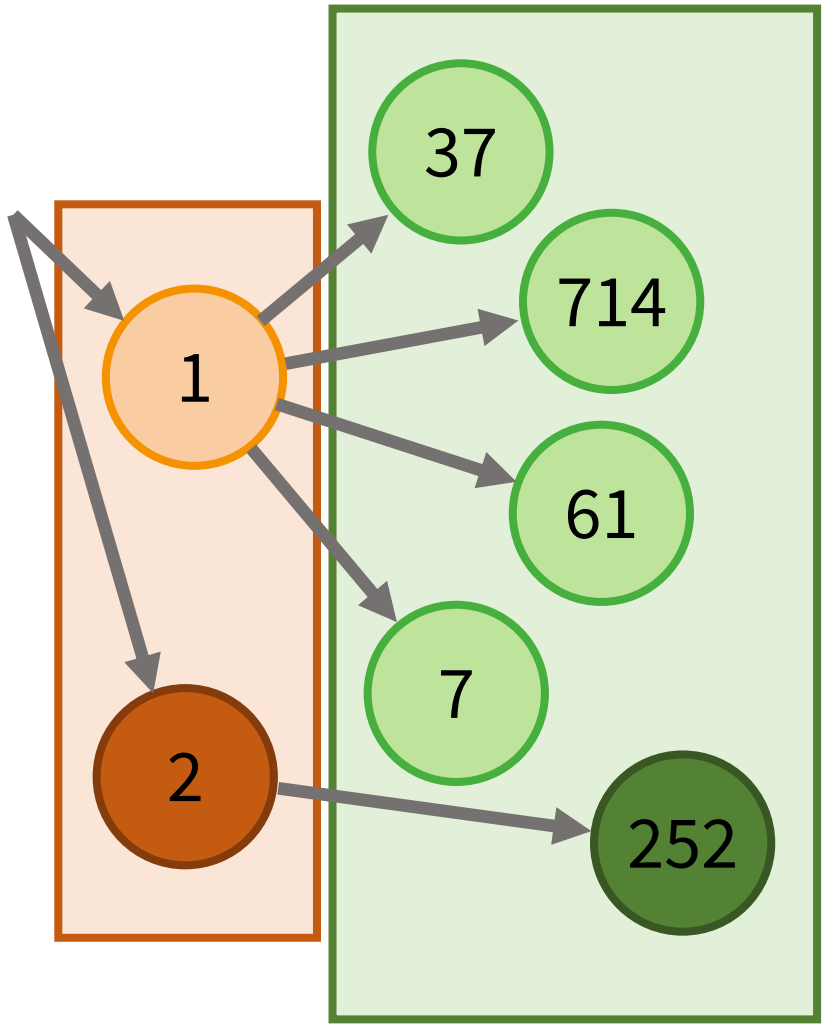


- Split on **long-latency operations**

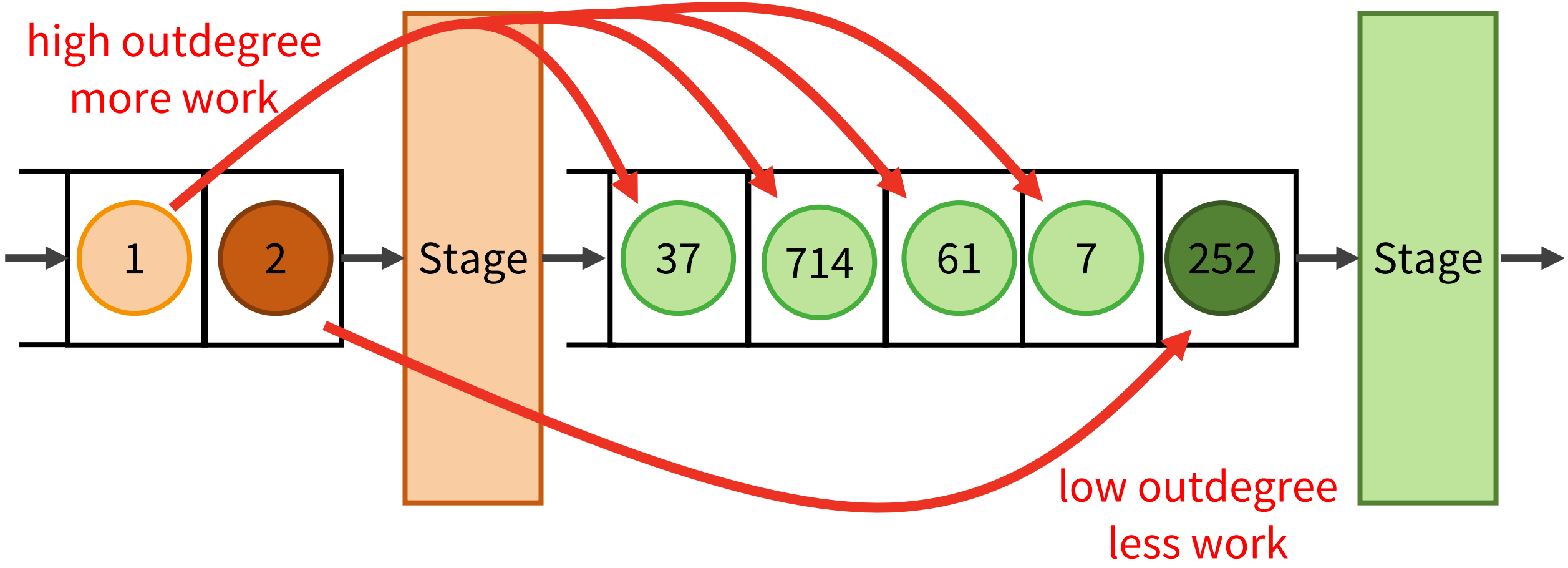
```
…
for ngh in neighbor list:
    x = array[ngh]
    func(x)
…
```

- Pipeline parallelism a natural fit
  - *Fine-grain*: queue operations **every few instructions**
  - Overheads for software techniques too high
  - Prior hardware techniques only for *regular* applications

Challenge #1: Irregular application pipelines have load imbalance

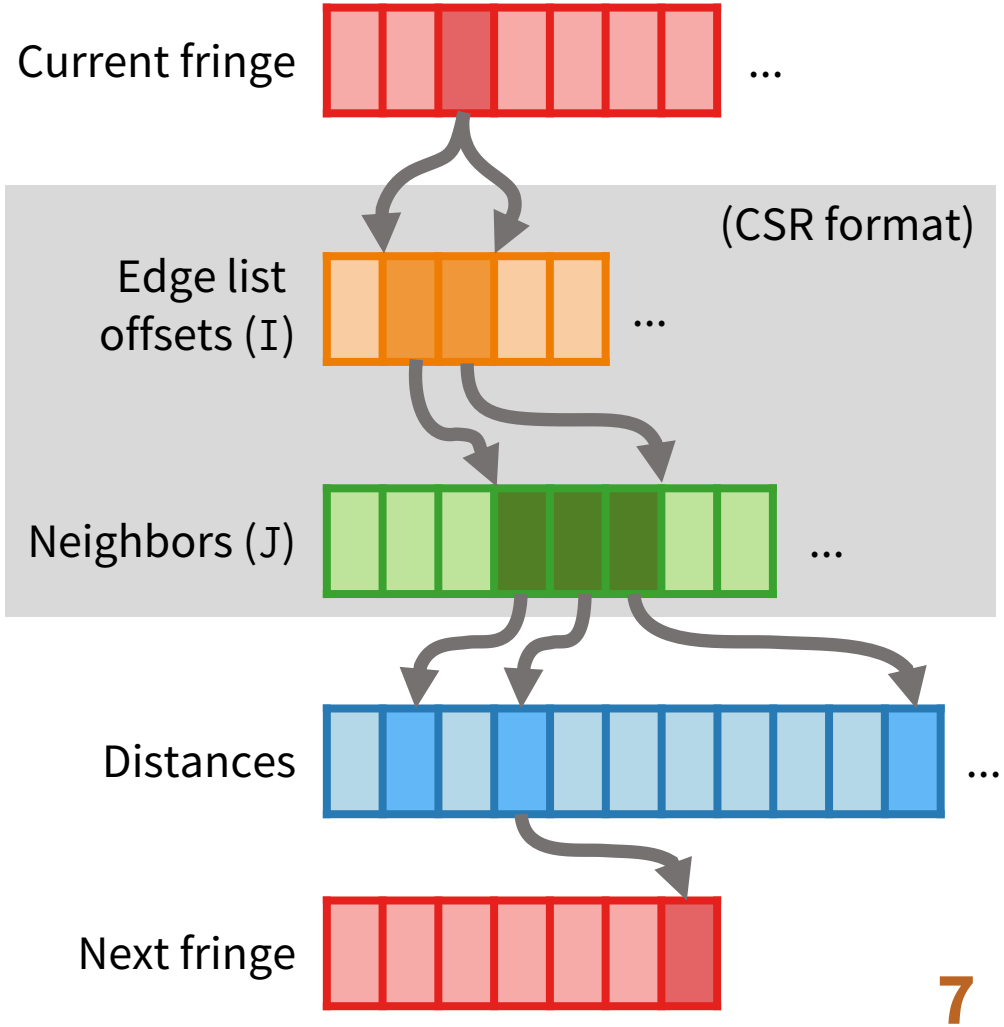


Challenge #1: Irregular application pipelines have load imbalance



Challenge #2: Full decoupling requires several stages

```
def bfs(src):
  …
  for v in current fringe:
    start, end = offsets[v], offsets[v+1]
    for ngh in neighbors[start:end]:
      dist = distances[ngh]
      if dist is not set:
        set distance; add to next fringe
  …
```



Challenge #2: Full decoupling requires several stages

```
def bfs(src):
  …
  for v in current fringe:
    start, end = offsets[v], offsets[v+1]
    for ngh in neighbors[start:end]:
      dist = distances[ngh]
      if dist is not set:
        set distance; add to next fringe
  …
```

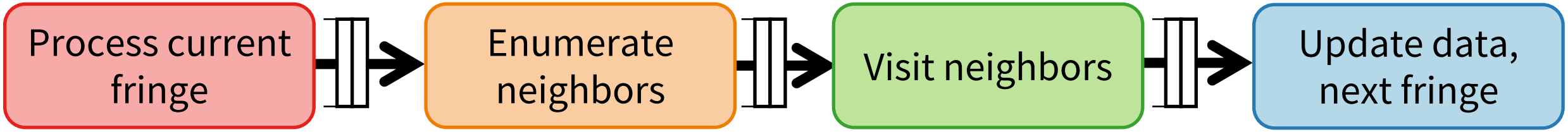
Process current fringe

Enumerate neighbors

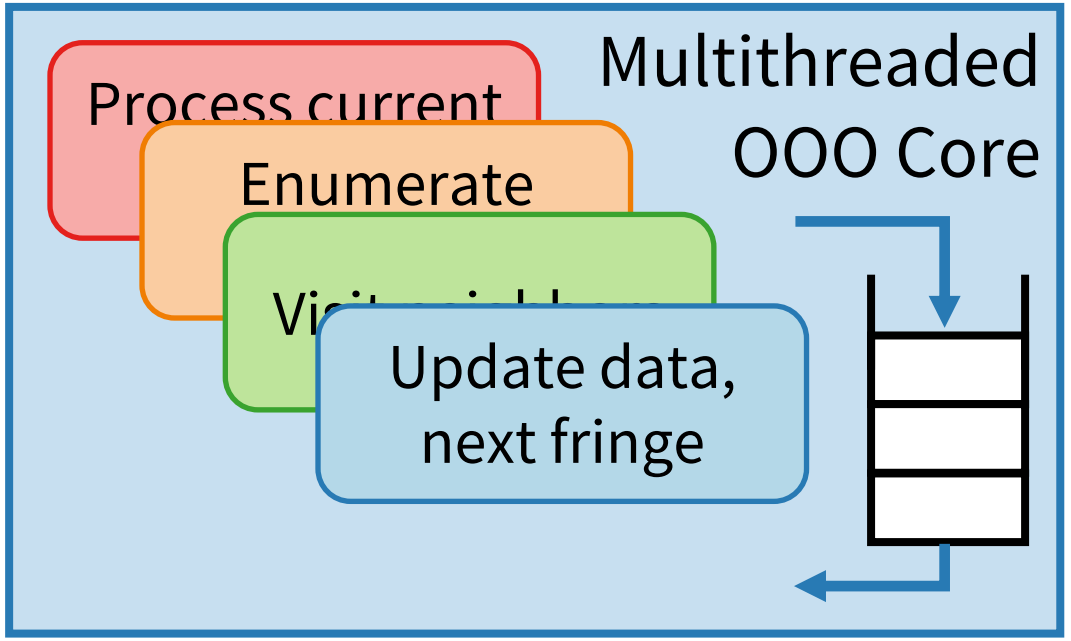
Visit neighbors

Update data, next fringe

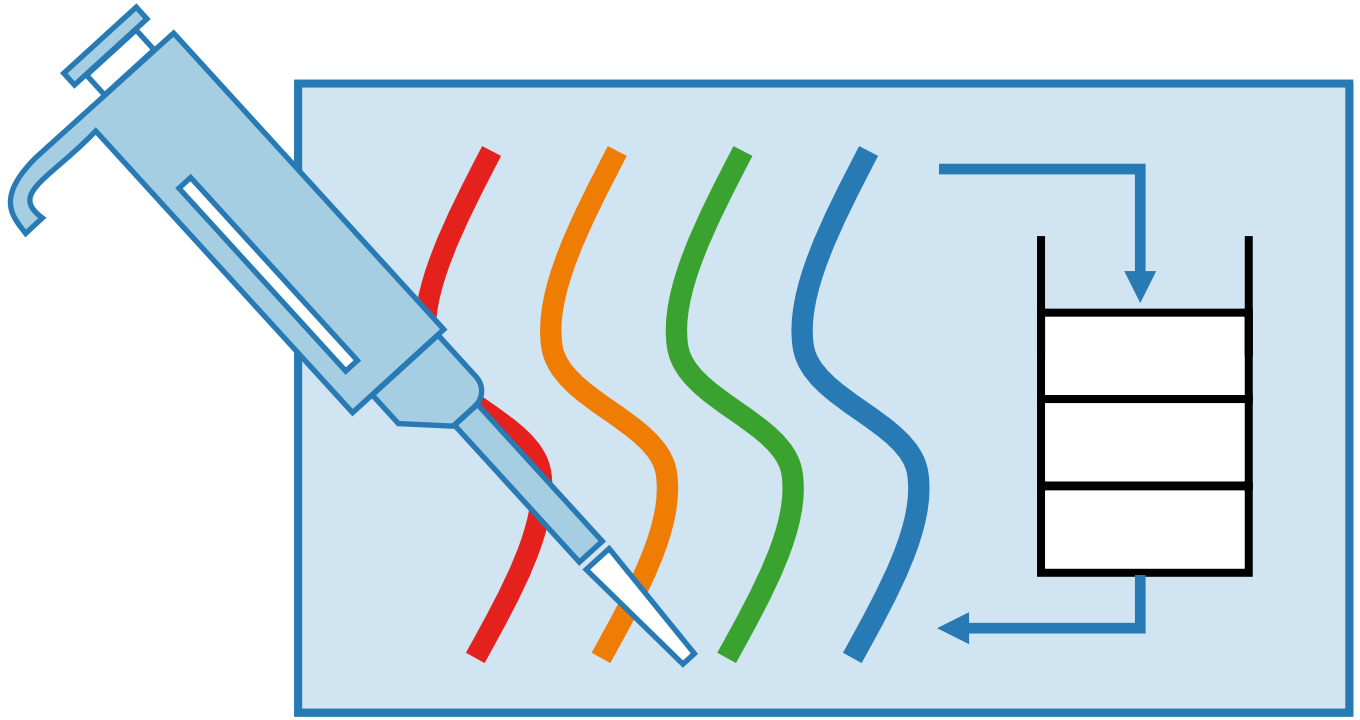
Insight: Exploit pipeline parallelism in a multithreaded core



Insight: Exploit pipeline parallelism in a multithreaded core



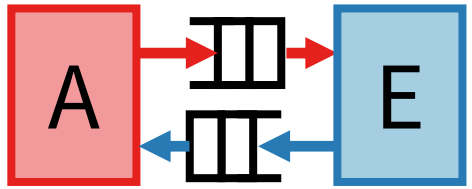
Insight: Exploit pipeline parallelism in a multithreaded core



Pipette's features

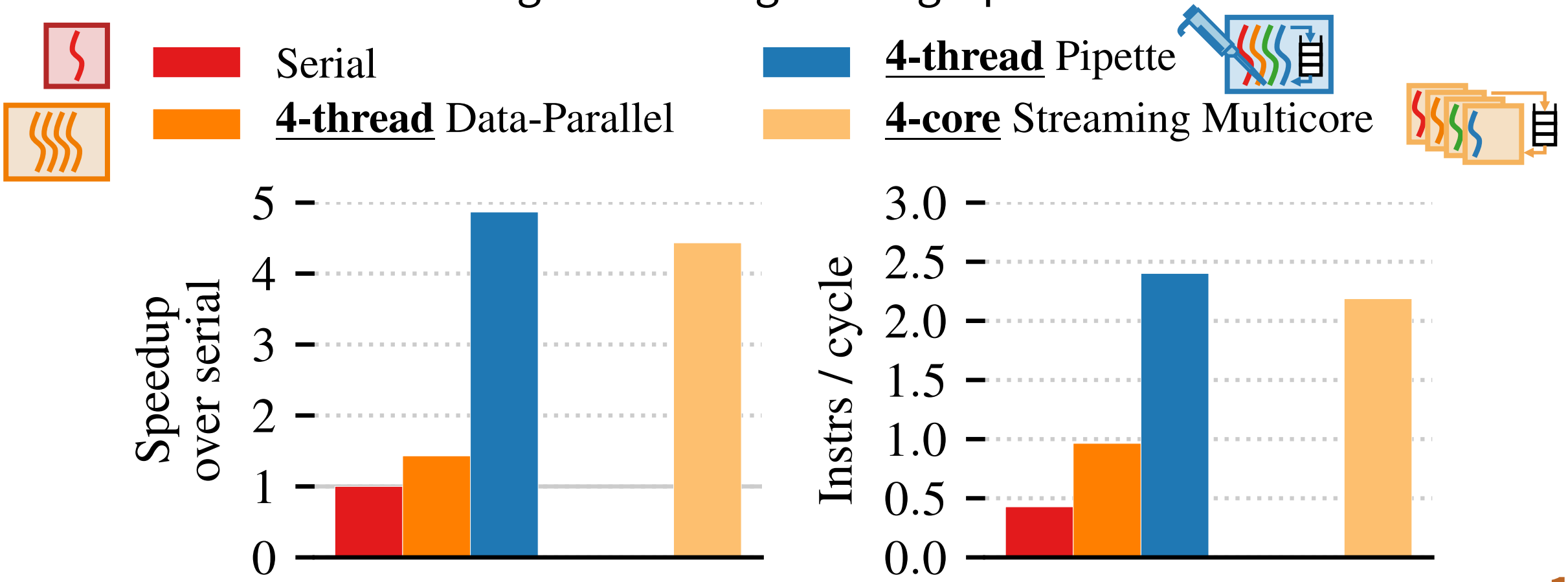
Feature	Achieves
Reuse PRF to build architecturally-visible queues for inter-thread communication	✓ Latency tolerance
Reuse SMT to time-multiplex stages	✓ Load balance
ISA primitives for fast queue operations & efficient control flow changes Cheap acceleration of common access patterns	✓ High performance

Prior work's missing ingredients

Prior work	Enough stages?	Load balance?	Flexible decoupling & control
 Decoupled access-execute (DAE [ISCA'82] , DeSC [MICRO'15], ...)	✗	✗	✗
 Streaming multicore (Raw [MICRO'02], MPPA [HPEC'13], ...)		✗	✗
Decoupled multithreaded cores (Outrider [ISCA'11], DSWP [PACT'04], ...)			✗
Data structure fetchers/prefetchers (HATS [MICRO'18], SQRL [PACT'14]; IMP [MICRO'15], ...)	✗ Domain specific; Area & power overheads		

Pipette accelerates irregular applications

- 6-wide OOO running BFS on large road graph



Agenda

Background

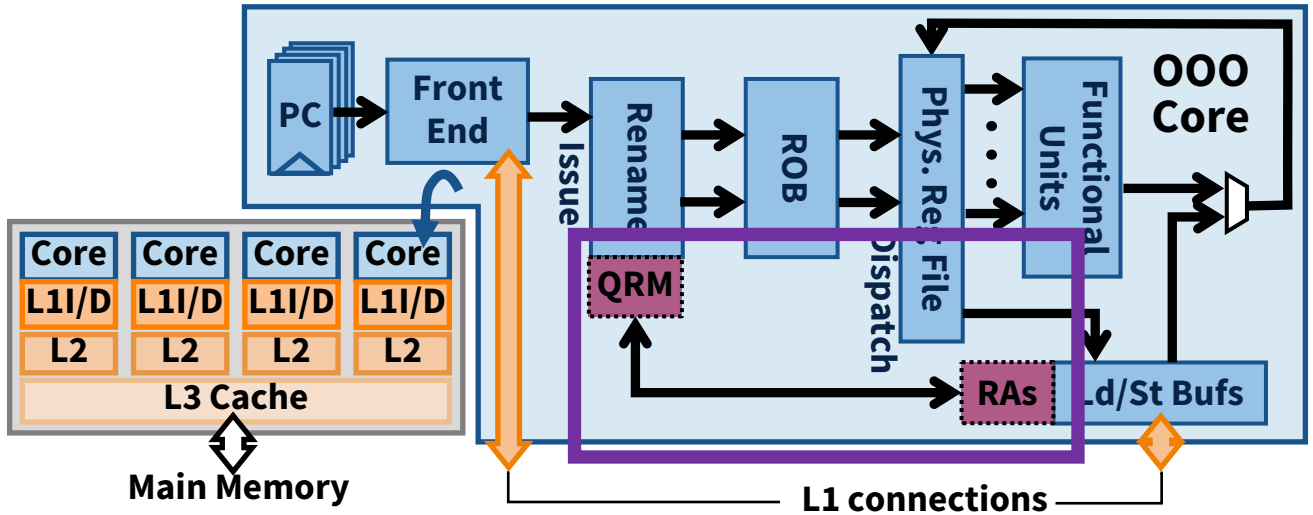
Pipette

- Architecturally visible queues
- Inter-thread control flow
- Further accelerating common access patterns

Evaluation

Pipette's ISA makes queue operations fast

- Map architectural registers to architectural *queues*
- Register write → enqueue; read → dequeue
- Queue operations frequent; implicit enqueue/dequeue semantics reduce instruction overheads



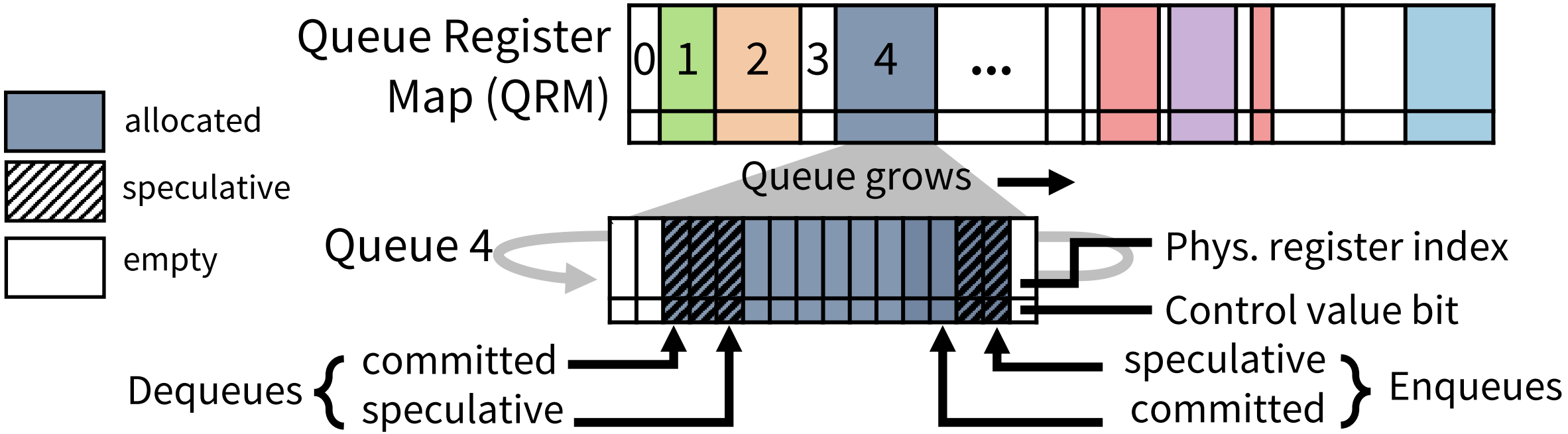
Queue Register

```
map_enq  q1, r3;
map_deq  q2, r4;
   ...
   addi  r3, r4, 5;
```

Reading r4 dequeues q2
Writing r3 enqueues q1

Reusing the PRF to build queues

- Physical register file (PRF) underutilized for irregular applications
- Insight: reuse storage & OOO rename to manage queues!
- Queue Register Map (QRM), a simple extension to manage queues



Agenda

Background

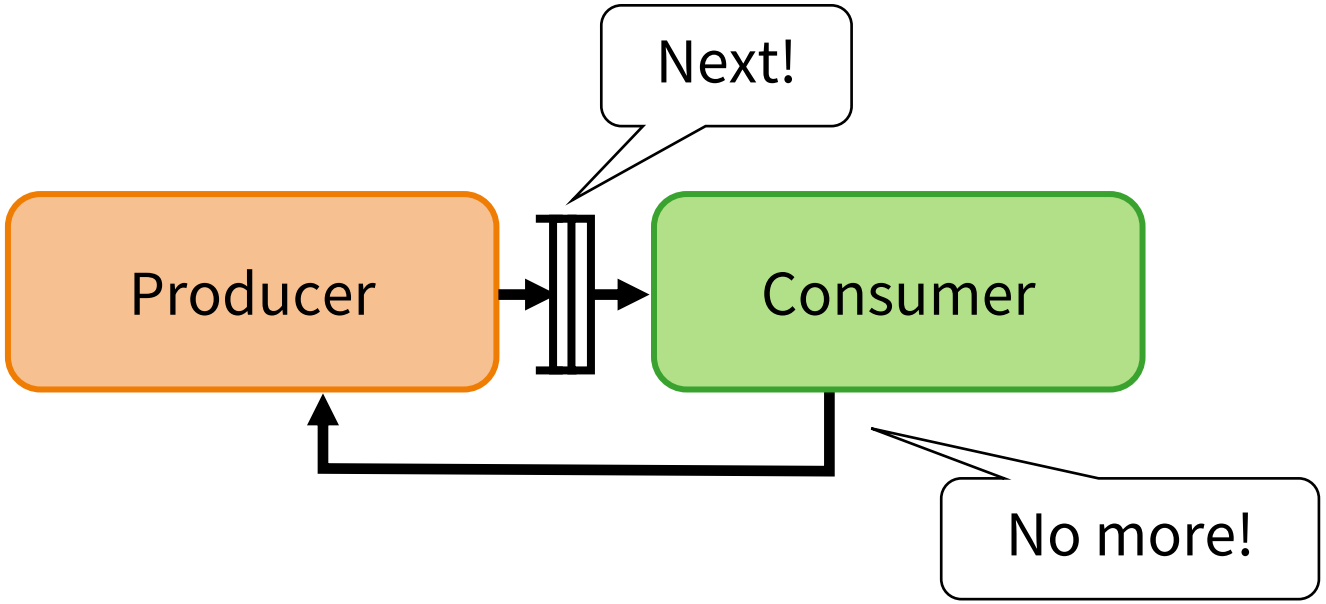
Pipette

- Architecturally visible queues
- Inter-thread control flow
- Further accelerating common access patterns

Evaluation

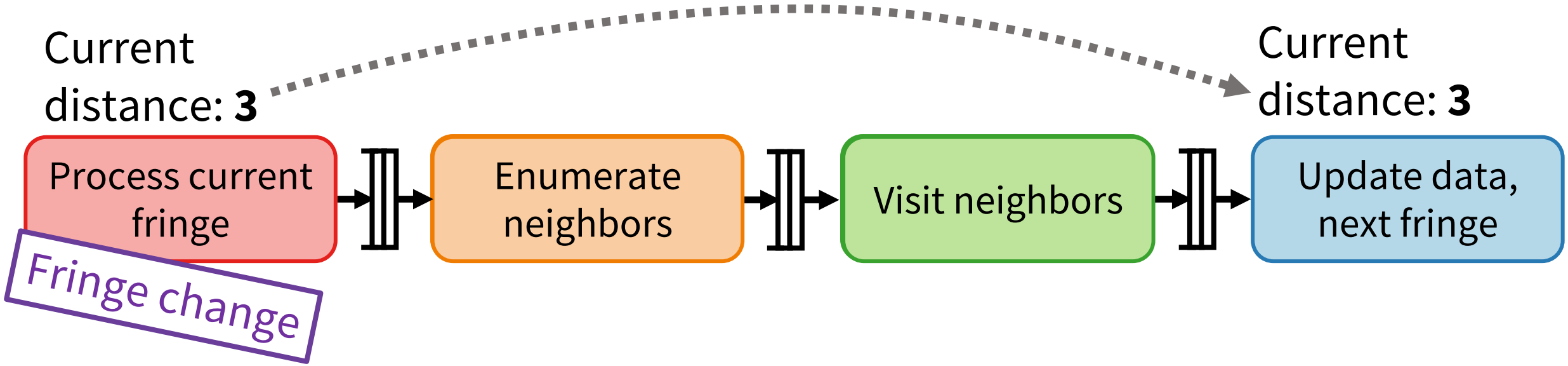
Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



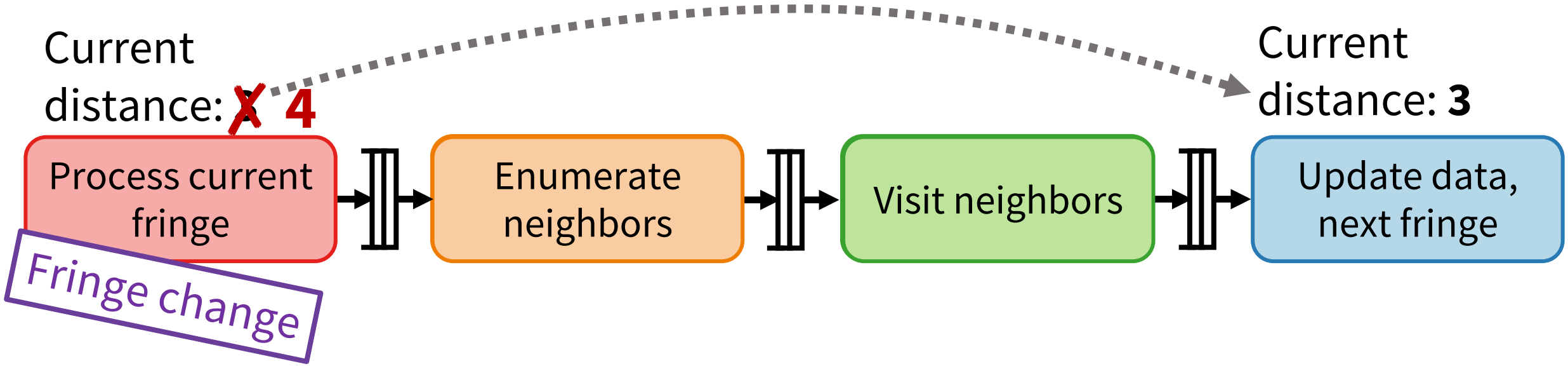
Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



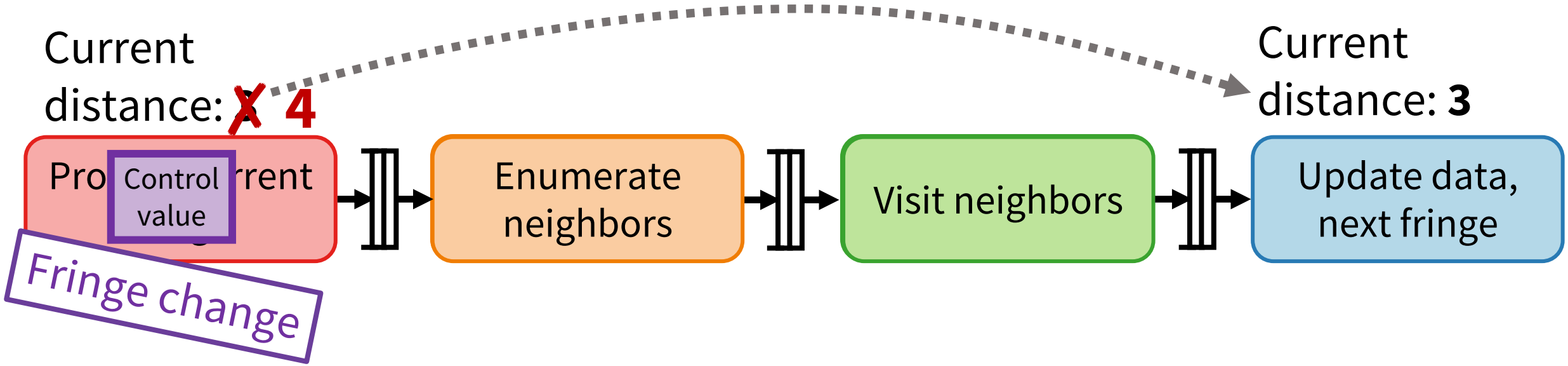
Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



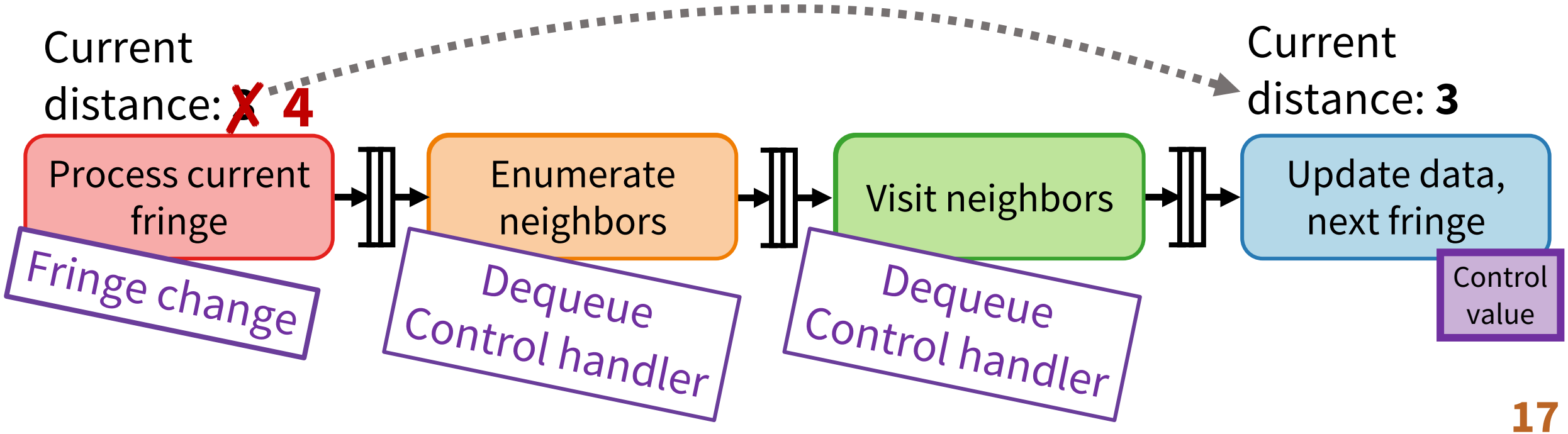
Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



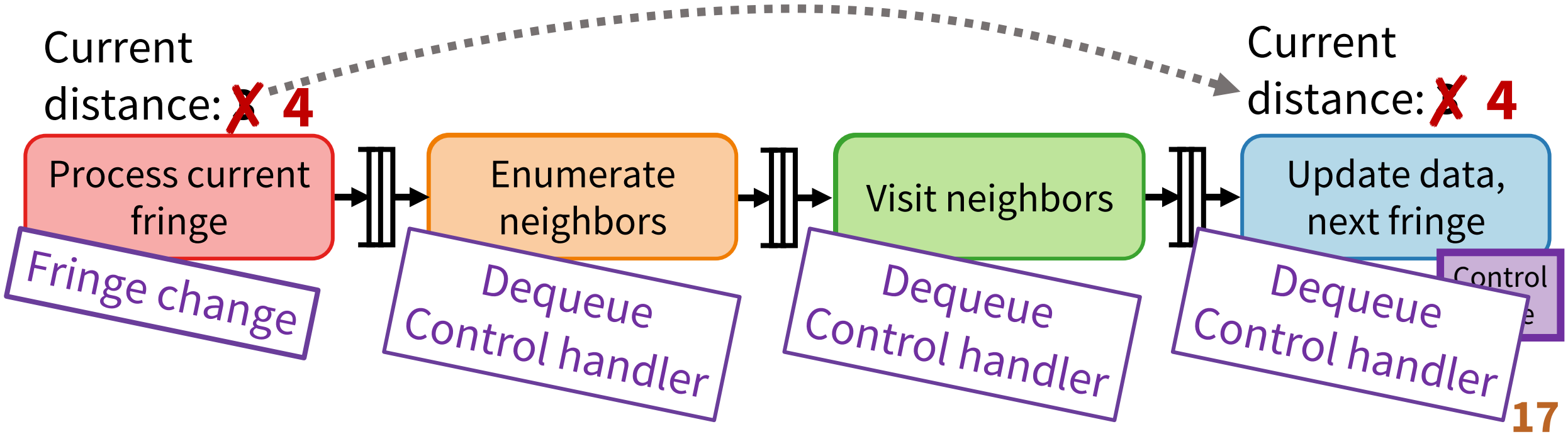
Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



Efficient control flow with Pipette

- Add hardware support to communicate control flow changes
- Producer ⟶ Consumer: Control values & dequeue control handlers
- Consumer ⟶ Producer: Enqueue control handlers



Agenda

Background

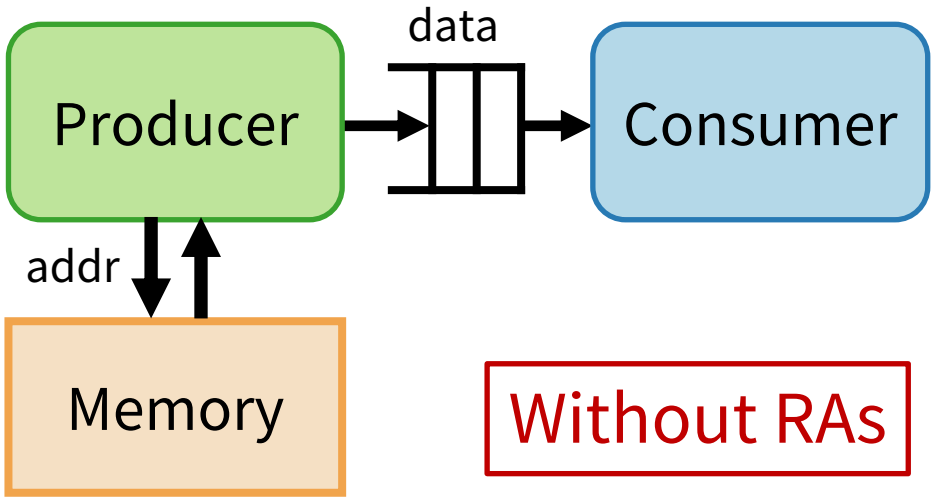
Pipette

- Architecturally visible queues
- Inter-thread control flow
- Further accelerating common access patterns

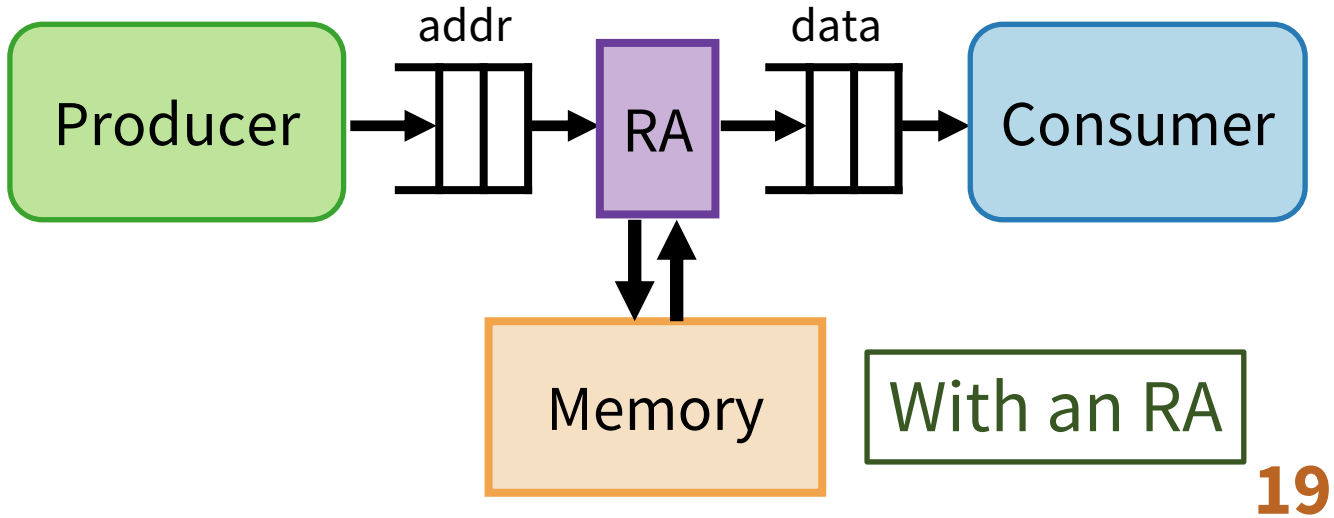
Evaluation

Further accelerating common access patterns

- Memory access patterns are often easy to compute
- Further decouple long-latency accesses with reference accelerators (RAs), connected to queues
- Improves decoupling without filling load/store queues



Without RAs



With an RA

Agenda

Background

Pipette

Evaluation

Evaluation

- Event-driven cycle-level simulator
- 6-wide OOO core (similar to Intel Skylake)
- Baseline:

data-parallel 4-way multithreaded

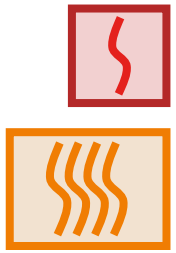
- Additional comparison:

4-core “Streaming Multicore” (using Pipette ISA)

Applications evaluated:

- Graph analytics:
  - Breadth-first search (BFS)
  - Connected components (CC)
  - PageRank-Delta (PRD)
  - Radii Estimation (Radii)
- Sparse linear algebra:
  - Sparse matrix-matrix multiply (SpMM)
- Databases:
  - Silo

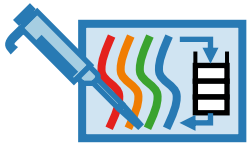
Pipette achieves significant speedups



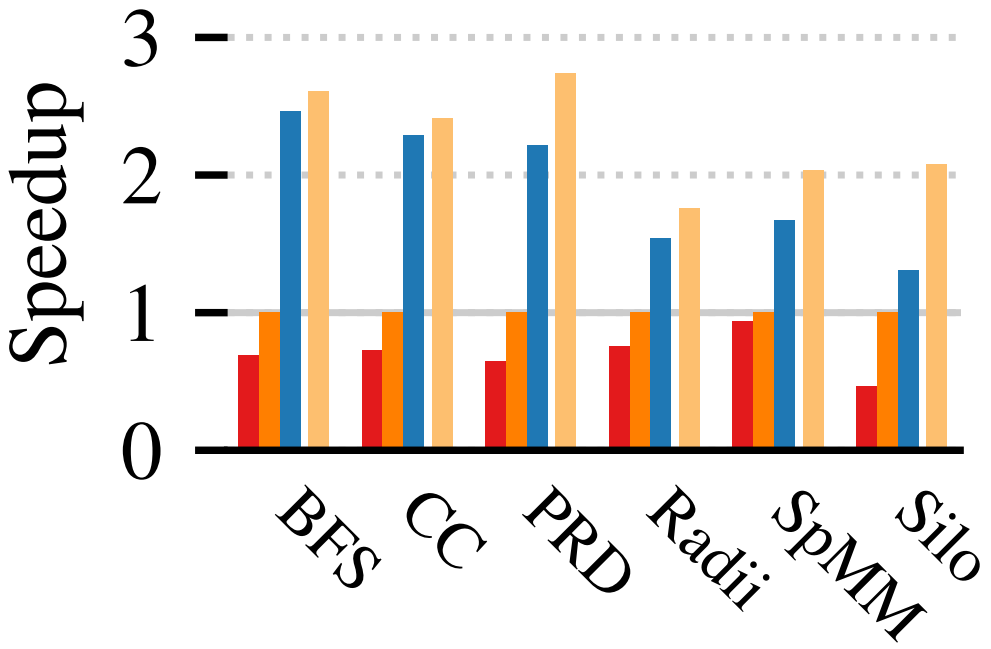
Serial 1-core 1-thread

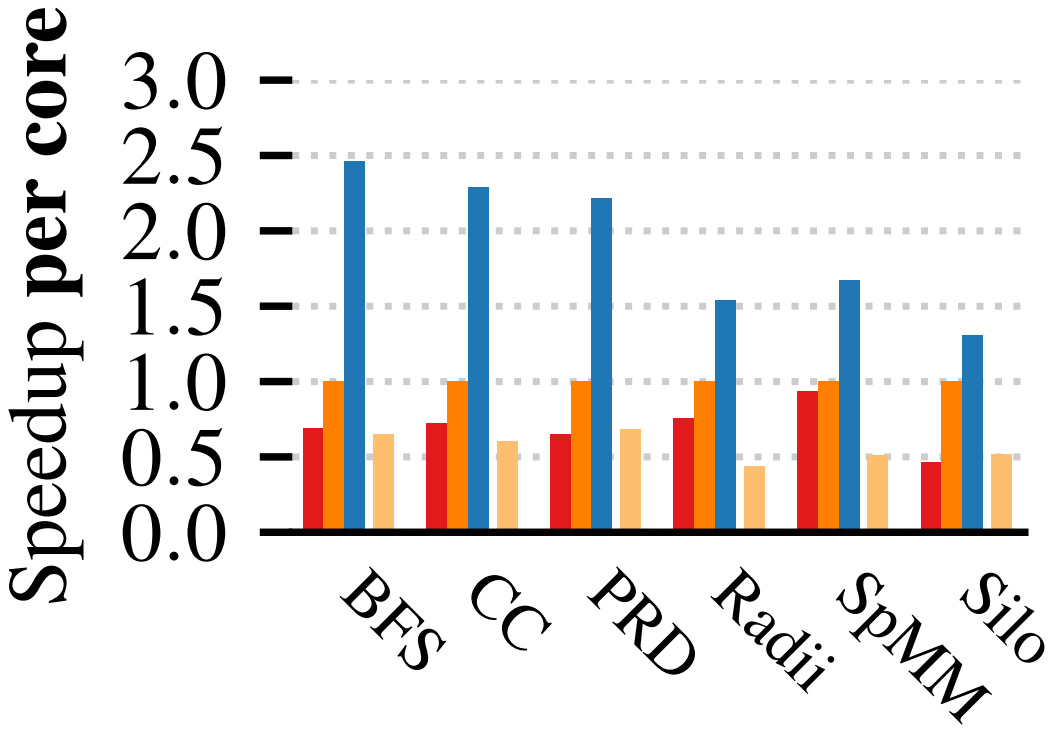
Data-parallel **1-core 4-thread**

Pipette **1-core 4-thread**



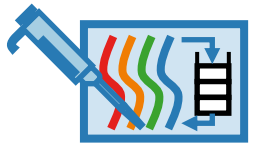
Streaming **4-core 1-thread**



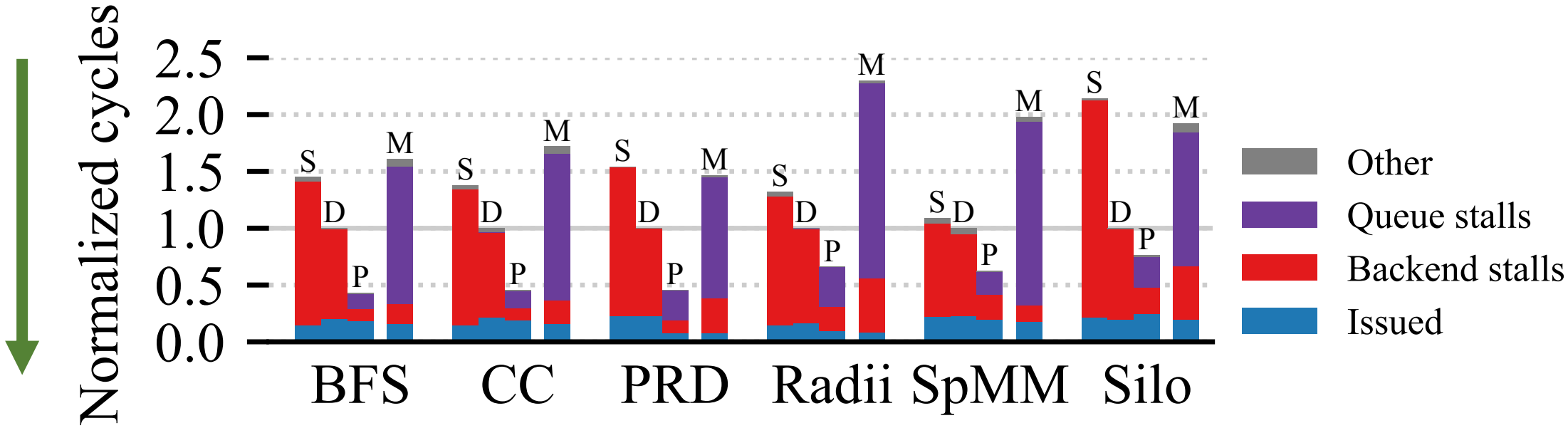


Pipette effectively tolerates latency and load imbalance





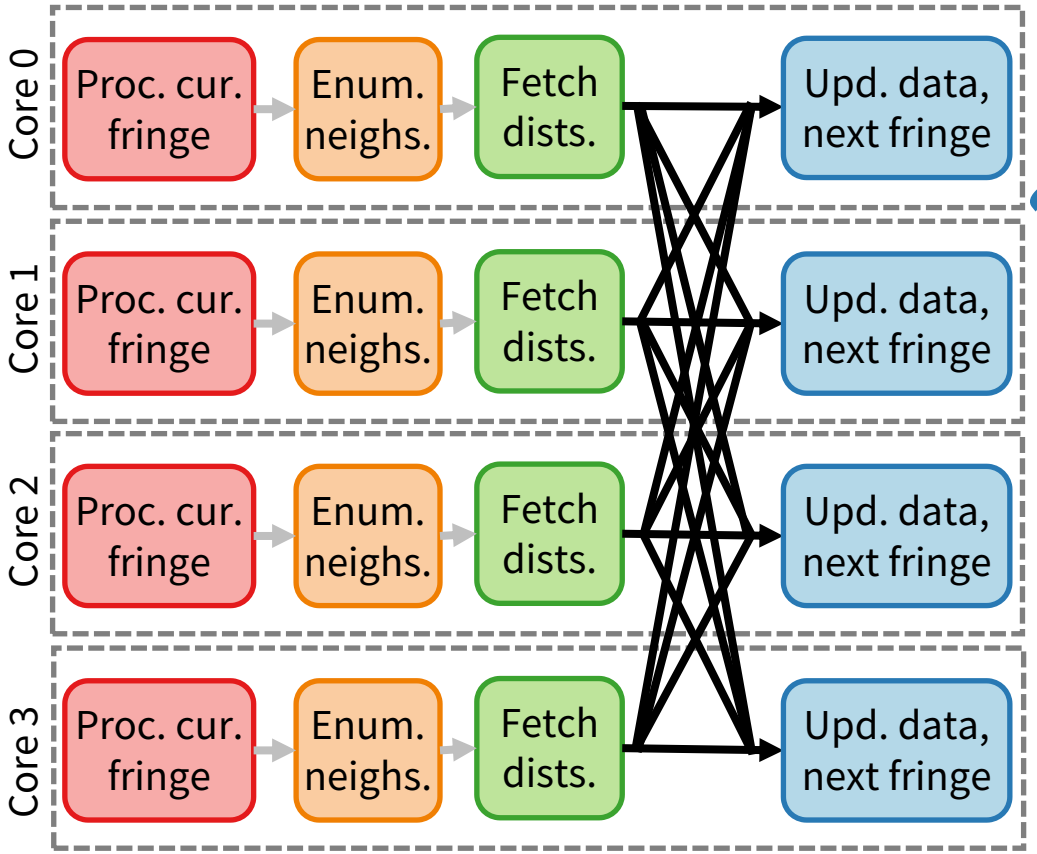

S: Serial D: Data-parallel (baseline) P: Pipette M: streaming Multicore

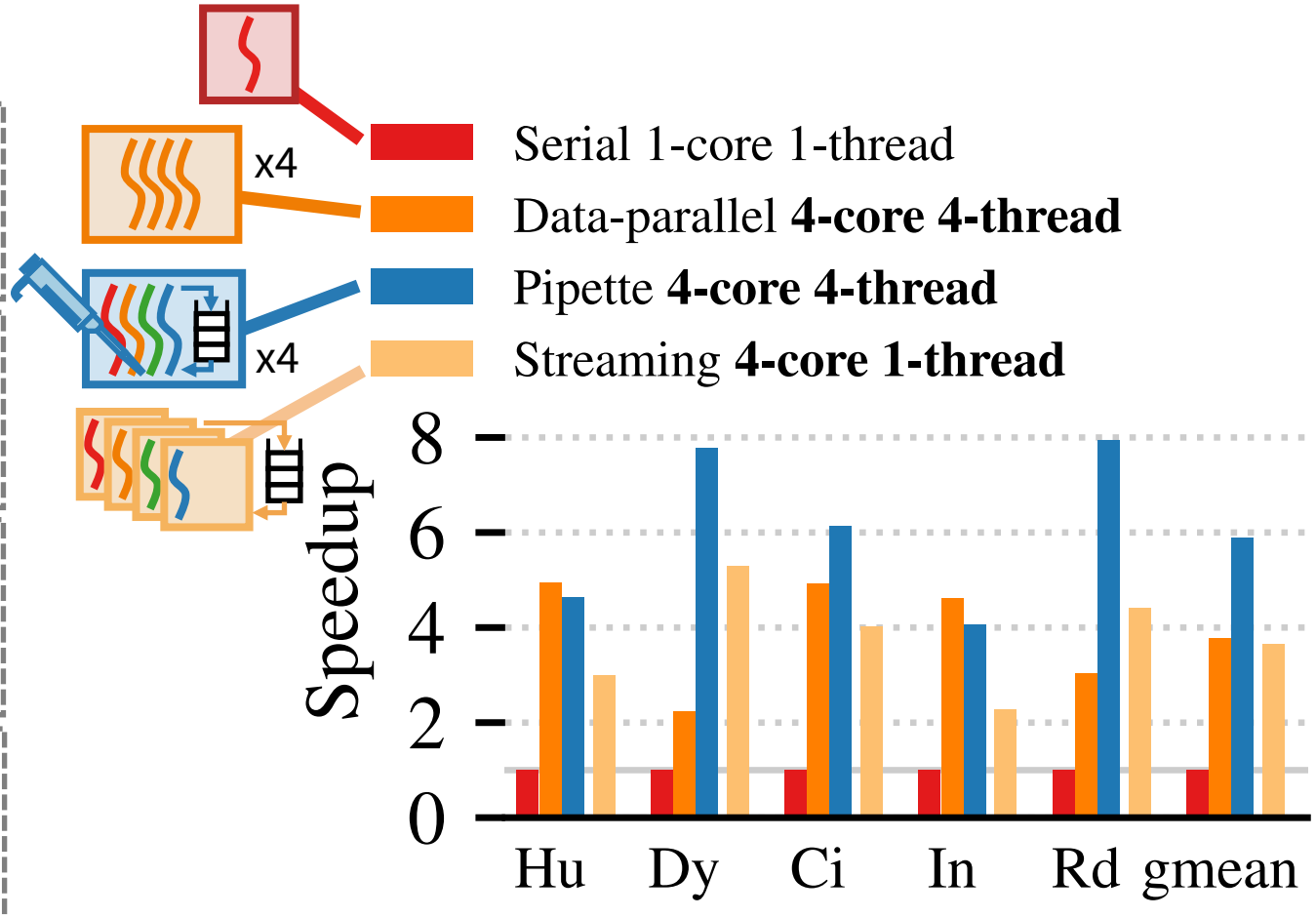


Pipette reduces impact of memory latency (**smaller red bars**) and load imbalance (**smaller purple bars**)

Case study:

Multicore, multithreaded BFS





See paper for:

- Specifics on OOO core reuse & interaction with speculation
- Low-cost implementation of reference accelerators
- Connecting queues *across* cores
- Detailed performance analysis, including energy savings up to 2×
- Additional analyses of number of stages, PRF size, use of RAs

Conclusion

- Irregular applications hamper core utilization
- Pipette reuses OOO core structures to efficiently implement irregular applications as pipeline-parallel programs
- Speedups of gmean 1.9×, up to 3.9× over SMT baseline in challenging irregular benchmarks

Thank you!

Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism

Quan Nguyen and Daniel Sanchez

qmn@csail.mit.edu and sanchez@csail.mit.edu

MICRO-53

Live session: Session 4A: Microarchitecture II

(October 20, 2020 at 2 PM EDT)