PIM-Enabled Instructions: A Low-Overhead, Locality-Aware PIM Architecture

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu[+], and Kiyoung Choi

Seoul National University [+]Carnegie Mellon University

Processing-in-Memory

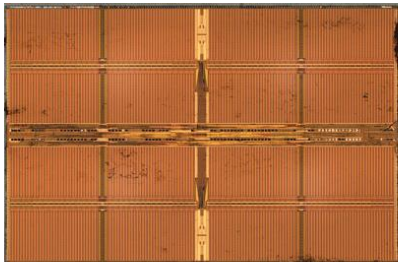
- Move computation to memory
  - Higher memory bandwidth
  - Lower memory latency
  - Better energy efficiency (e.g., off-chip links vs. TSVs)

- Originally studied in 1990s
  - Also known as processor-in-memory
  - e.g., DIVA, EXECUBE, FlexRAM, IRAM, Active Pages, …
  - *Not commercialized in the end*

Why was PIM unsuccessful in its first attempt?

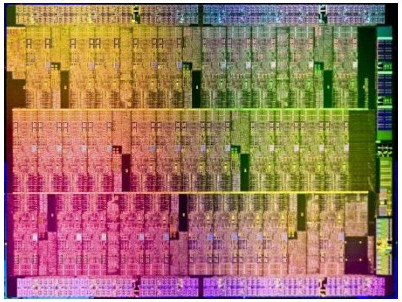
Challenges in Processing-in-Memory

Cost-effectiveness



DRAM die

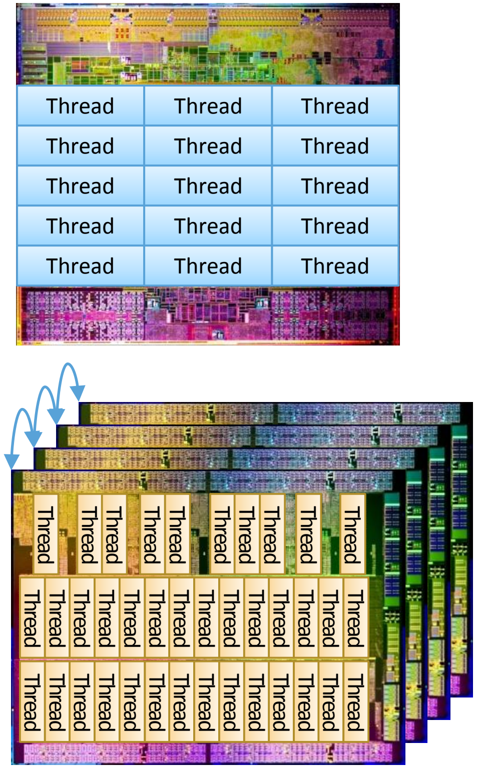
+



Complex Logic

Programming Model

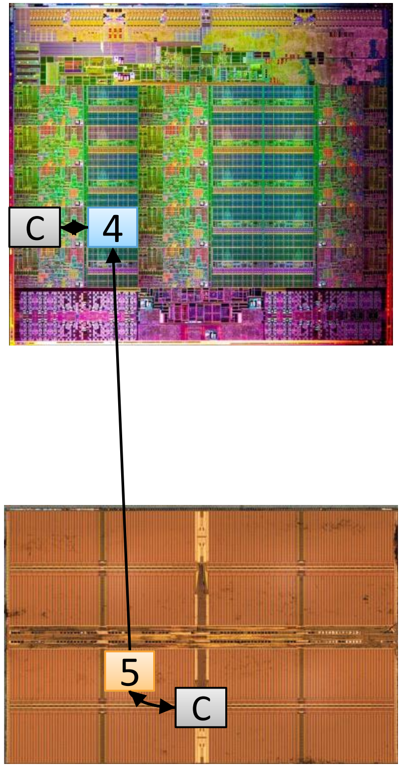
Host Processor



In-Memory Processors

Coherence & VM

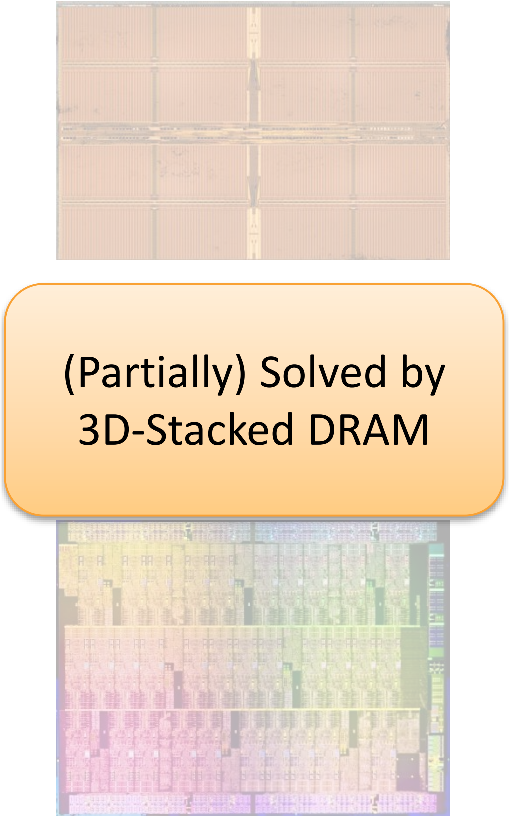
Host Processor



DRAM die

Challenges in Processing-in-Memory

Cost-effectiveness

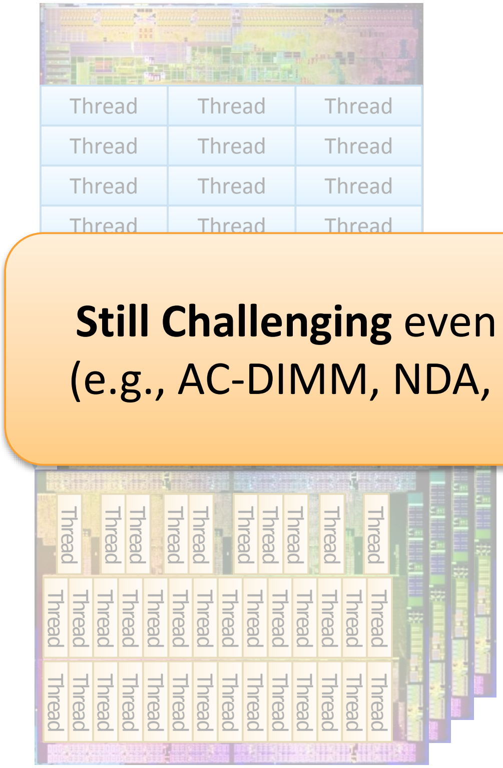


(Partially) Solved by 3D-Stacked DRAM

Complex Logic

Programming Model

Host Processor

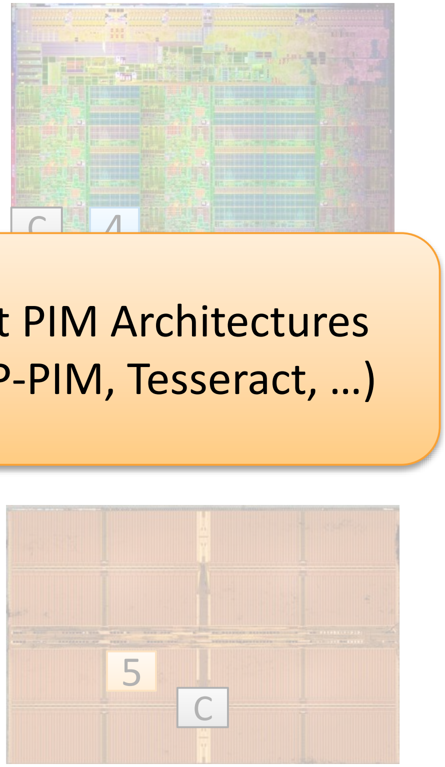


Still Challenging even in Recent PIM Architectures
(e.g., AC-DIMM, NDA, NDC, TOP-PIM, Tesseract, …)

In-Memory Processors

Coherence & VM

Host Processor



DRAM die

New Direction of PIM

- Objectives
  - Provide an intuitive programming model for PIM
  - Full support for cache coherence and virtual memory
  - Reduce the implementation overhead of PIM units
- Our solution: simple PIM operations as ISA extension
  - Simple: low-overhead implementation
  - PIM operations as host processor instructions: intuitive
  - Conventional PIM : Simple PIM ≈ GPGPU : SSE/AVX

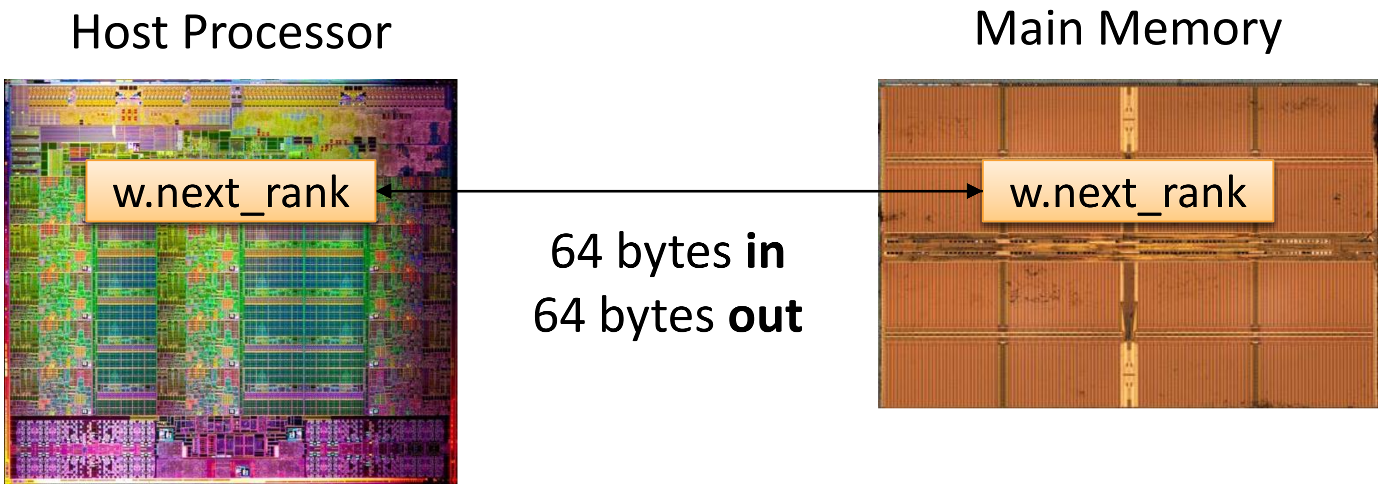
Potential of ISA Extension as PIM Interface

- Example: Parallel PageRank computation

```
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        w.next_rank += value;
    }
}
for (v: graph.vertices) {
    v.rank = v.next_rank; v.next_rank = alpha;
}
```

Potential of ISA Extension as PIM Interface

```
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        w.next_rank += value;
    }
}
```



Host Processor

Main Memory

w.next_rank

w.next_rank

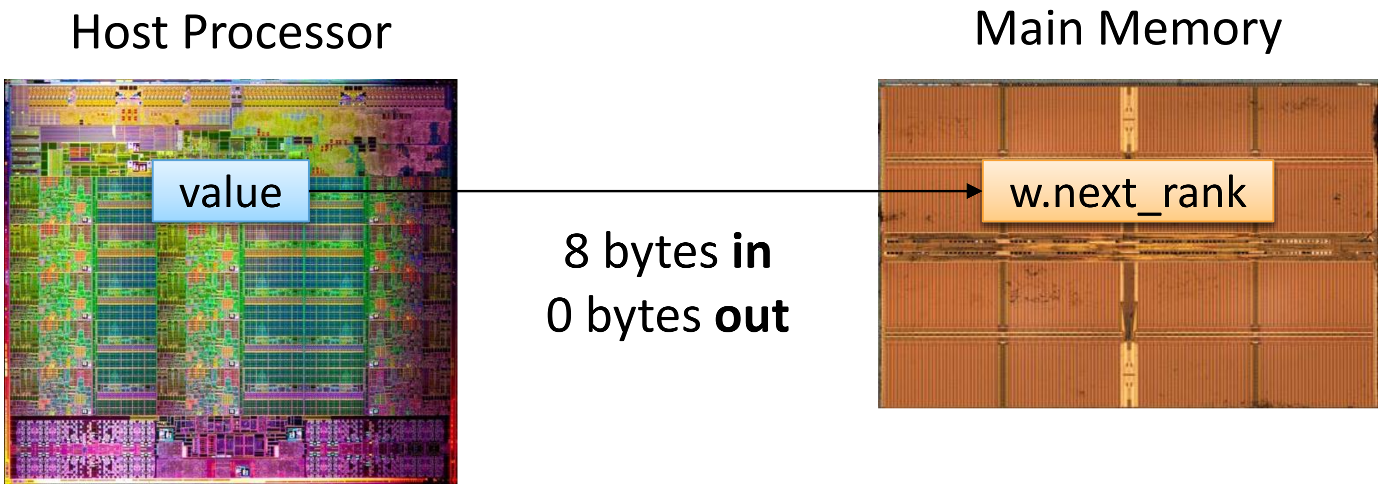
64 bytes **in**
64 bytes **out**

Conventional Architecture

Potential of ISA Extension as PIM Interface

```
for (v: graph.vertices) {
  value = weight * v.rank;
  for (w: v.successors) {
    __pim_add(&w.next_rank, value);
  }
}
```

pim.add r1, (r2)



Host Processor

value

Main Memory

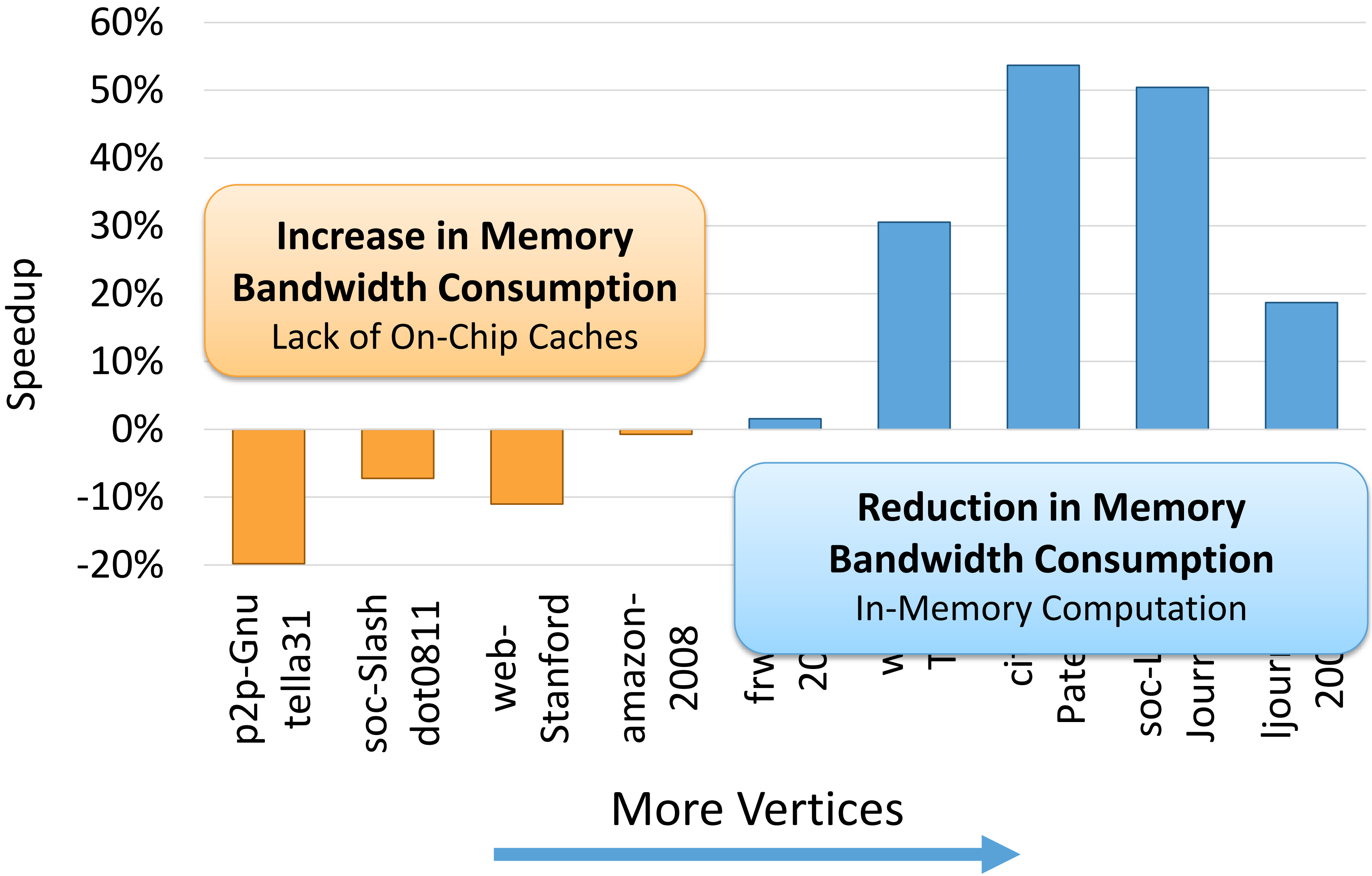
w.next_rank

8 bytes **in**

0 bytes **out**

In-Memory Addition

Potential of ISA Extension as PIM Interface



Increase in Memory Bandwidth Consumption
Lack of On-Chip Caches

Reduction in Memory Bandwidth Consumption
In-Memory Computation

Overview

1. How should simple PIM operations be interfaced to conventional systems?
   - Expose PIM operations as *cache-coherent, virtually-addressed host processor instructions*
   - No changes to the existing sequential programming model

2. What is the most efficient way of exploiting such simple PIM operations?
   - Dynamically determine the location of PIM execution based on data locality without software hints

PIM-Enabled Instructions

```
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        w.next_rank += value;
    }
}
```

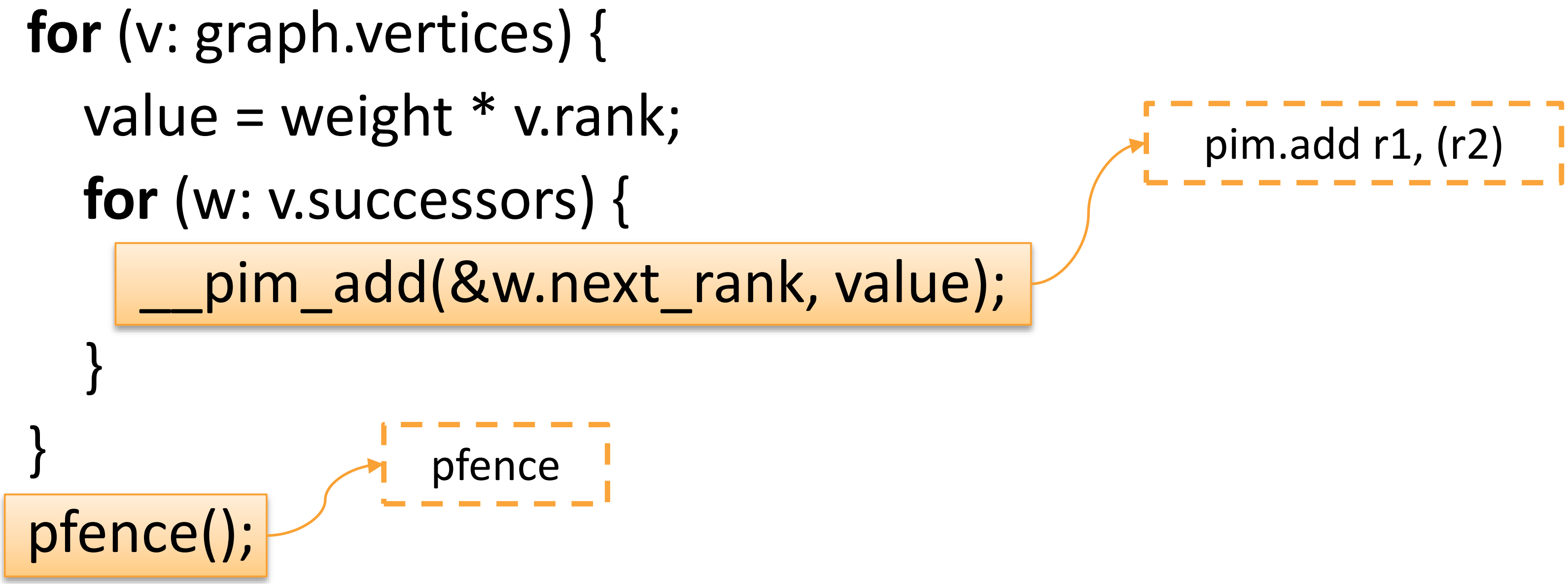
PIM-Enabled Instructions

```
for (v: graph.vertices) {
  value = weight * v.rank;
  for (w: v.successors) {
    __pim_add(&w.next_rank, value);
  }
}
```

```
pim.add r1, (r2)
```

- Executed either in memory or in the host processor
- Cache-coherent, virtually-addressed
- Atomic between different PEIs
- *Not* atomic with normal instructions (use *pfence*)

PIM-Enabled Instructions



```
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        __pim_add(&w.next_rank, value);
    }
}
pfence();
```

pim.add r1, (r2)

pfence

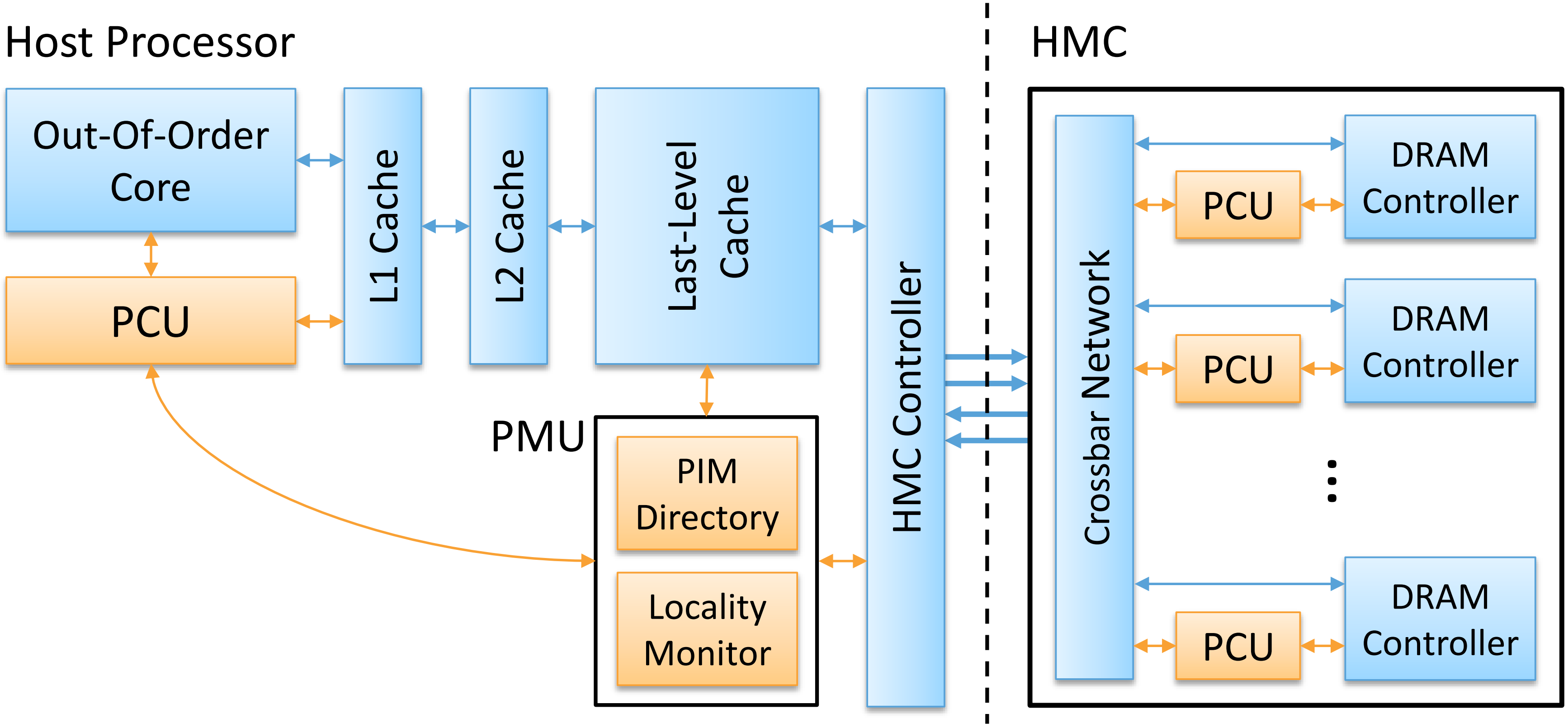
- Executed either in memory or in the host processor
- Cache-coherent, virtually-addressed
- Atomic between different PEIs
- *Not* atomic with normal instructions (use *pfence*)

PIM-Enabled Instructions

- Key to practicality: single-cache-block restriction
  - Each PEI can access *at most one last-level cache block*
  - Similar restrictions exist in atomic instructions

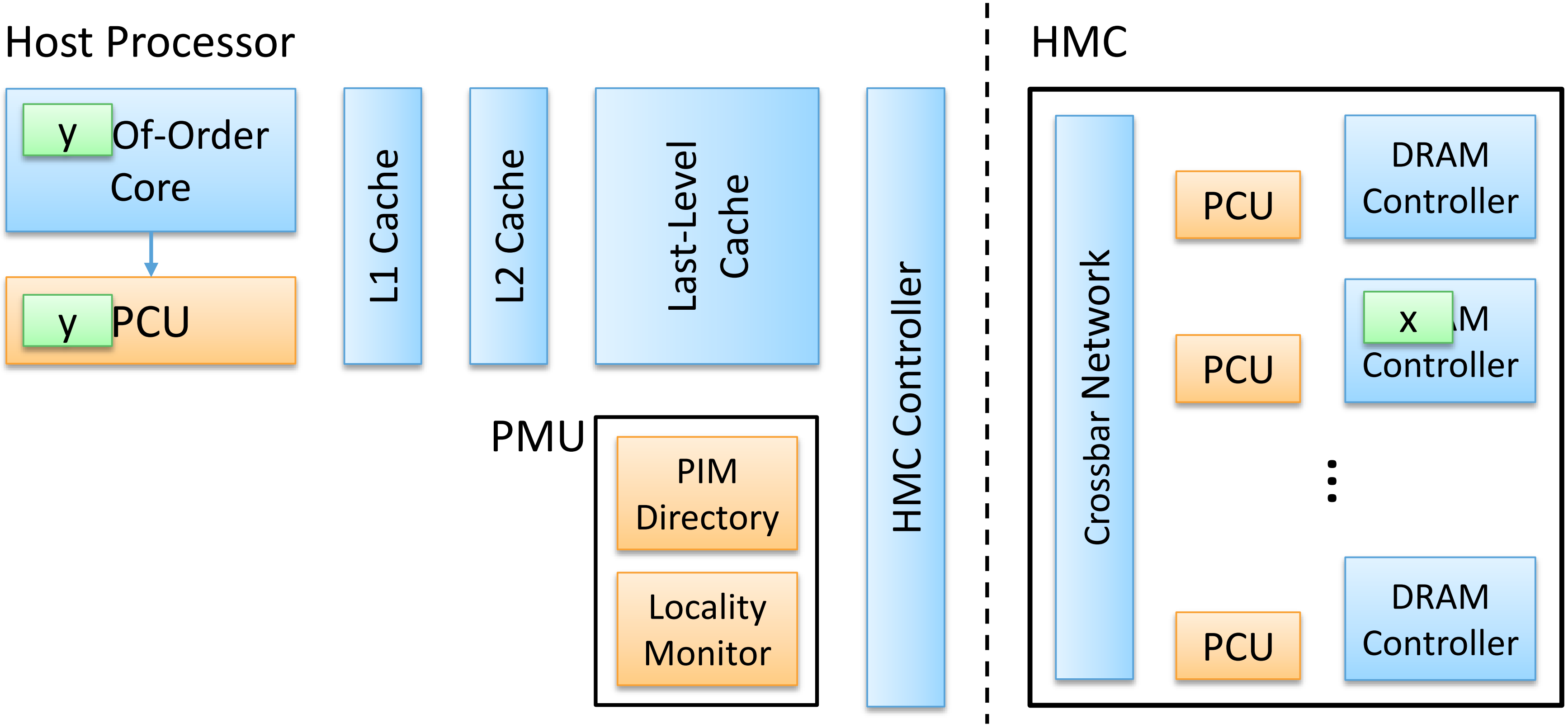
- Benefits
  - **Localization**: each PEI is bounded to one memory module
  - **Interoperability**: easier support for cache coherence and virtual memory
  - **Simplified locality monitoring**: data locality of PEIs can be identified by LLC tag checks or similar methods

Architecture



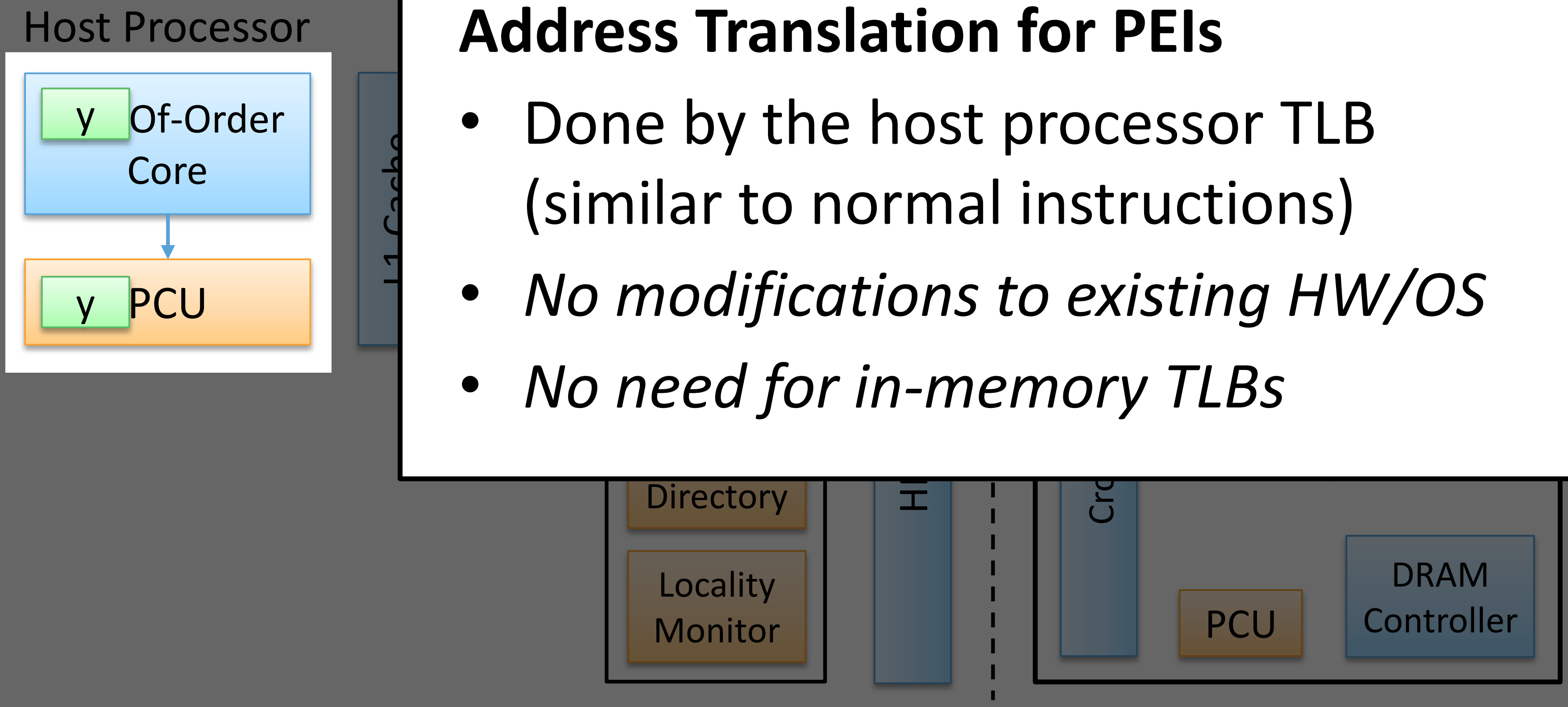
Proposed PEI Architecture

Memory-side PEI Execution



pim.add y, &x

Memory-side PEI Execution

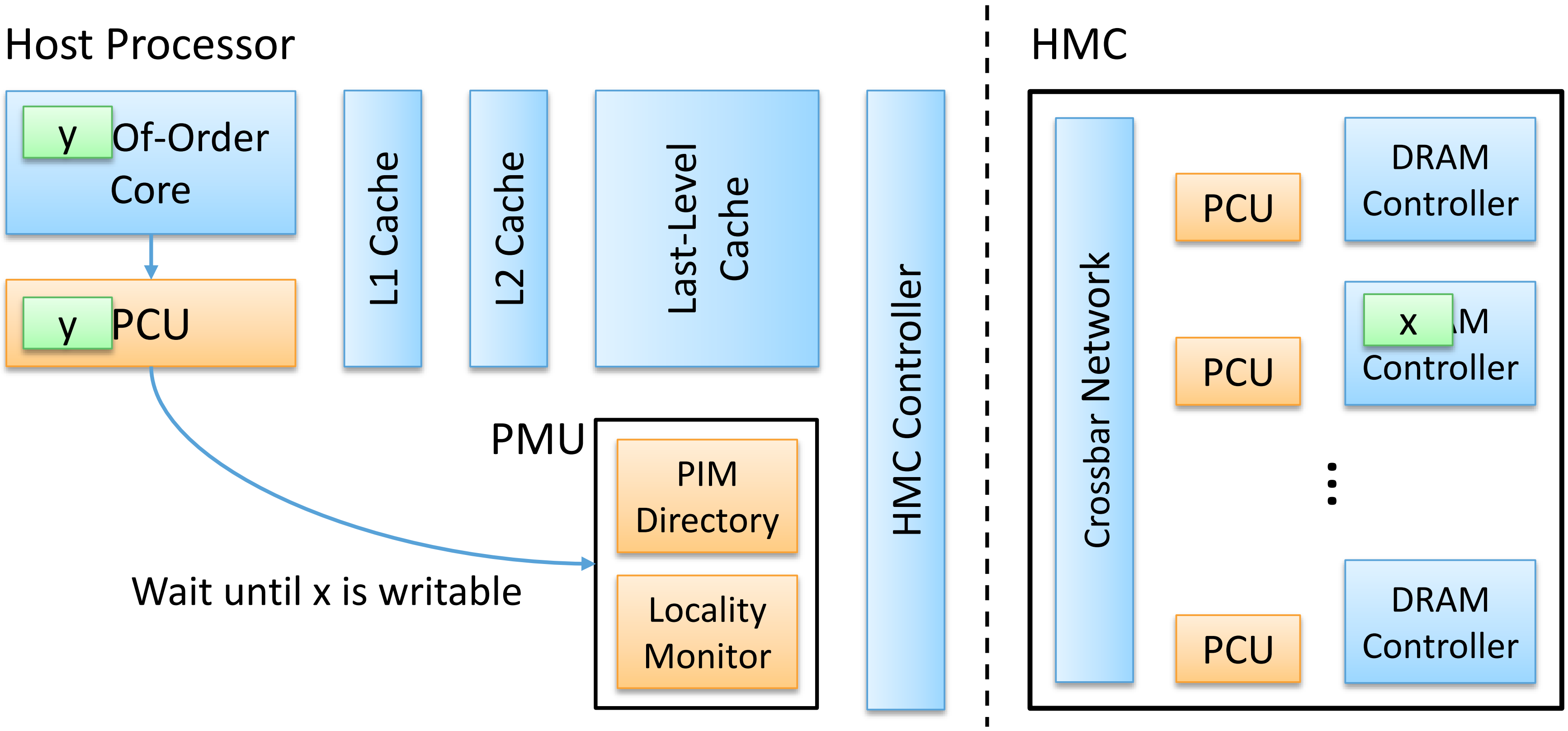


Address Translation for PEIs

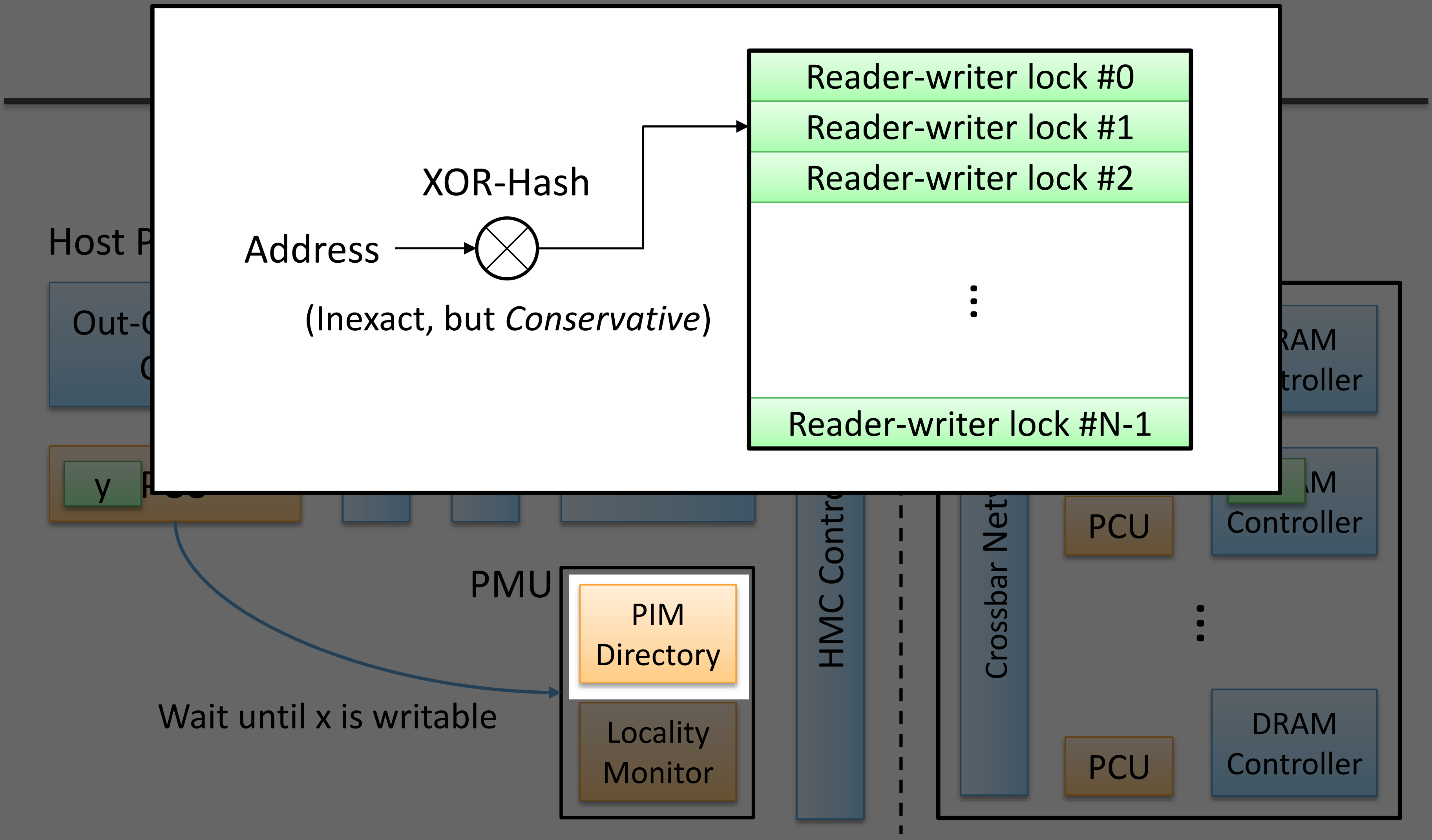
- Done by the host processor TLB (similar to normal instructions)
- *No modifications to existing HW/OS*
- *No need for in-memory TLBs*

pim.add y, &x

Memory-side PEI Execution



pim.add y, &x



XOR-Hash

Address

(Inexact, but *Conservative*)

Reader-writer lock #0

Reader-writer lock #1

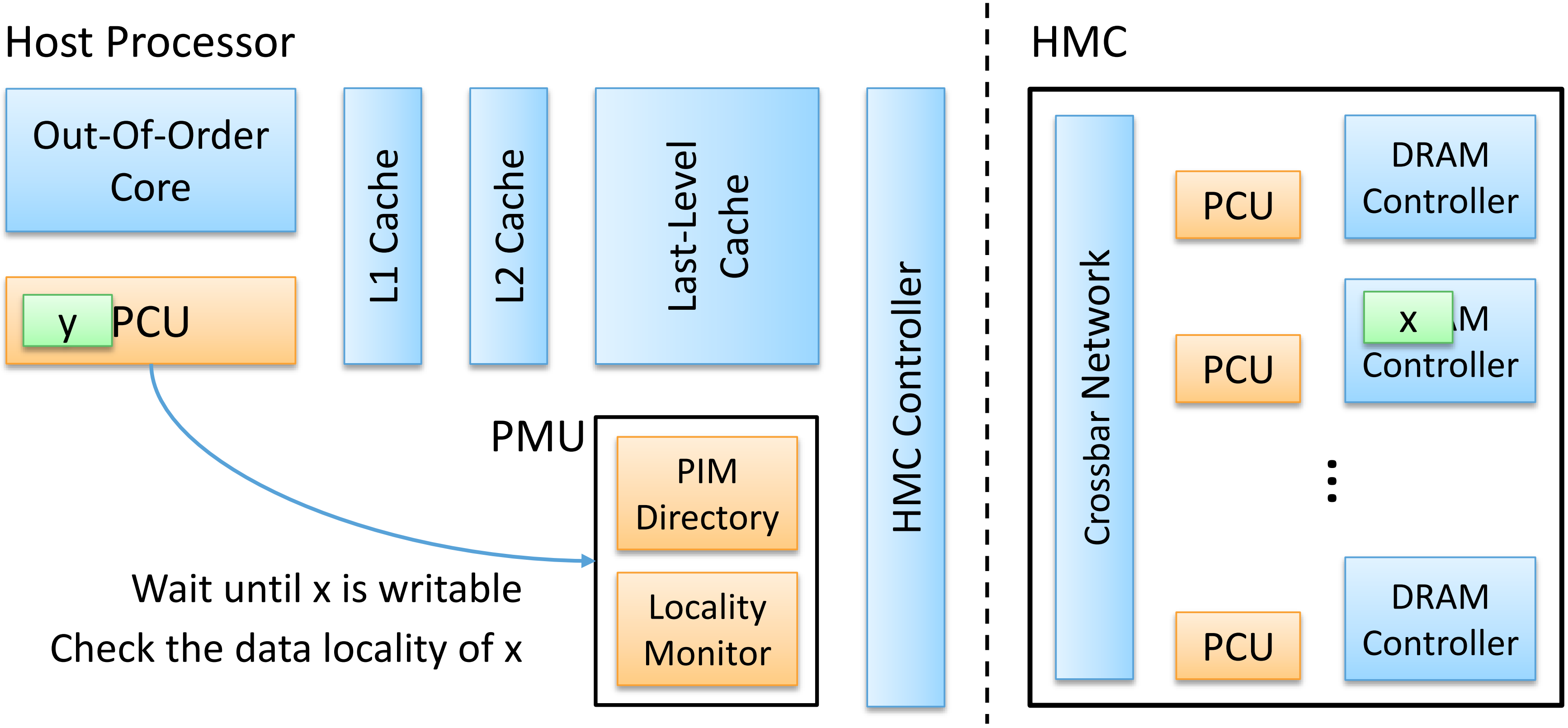
Reader-writer lock #2

⋮

Reader-writer lock #N-1

pim.add y, &x

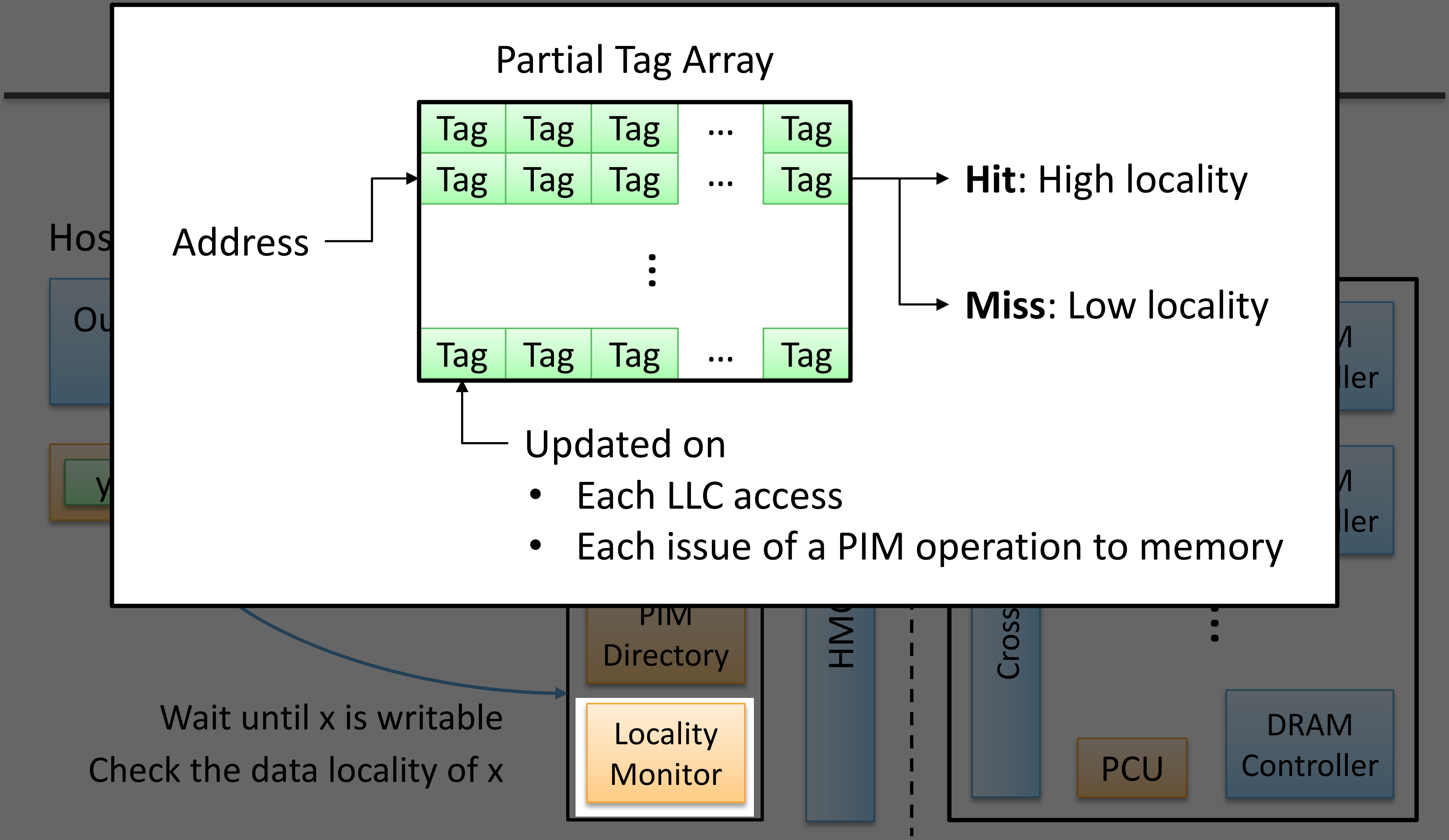
Memory-side PEI Execution



Wait until x is writable

Check the data locality of x

pim.add y, &x



Partial Tag Array

Updated on

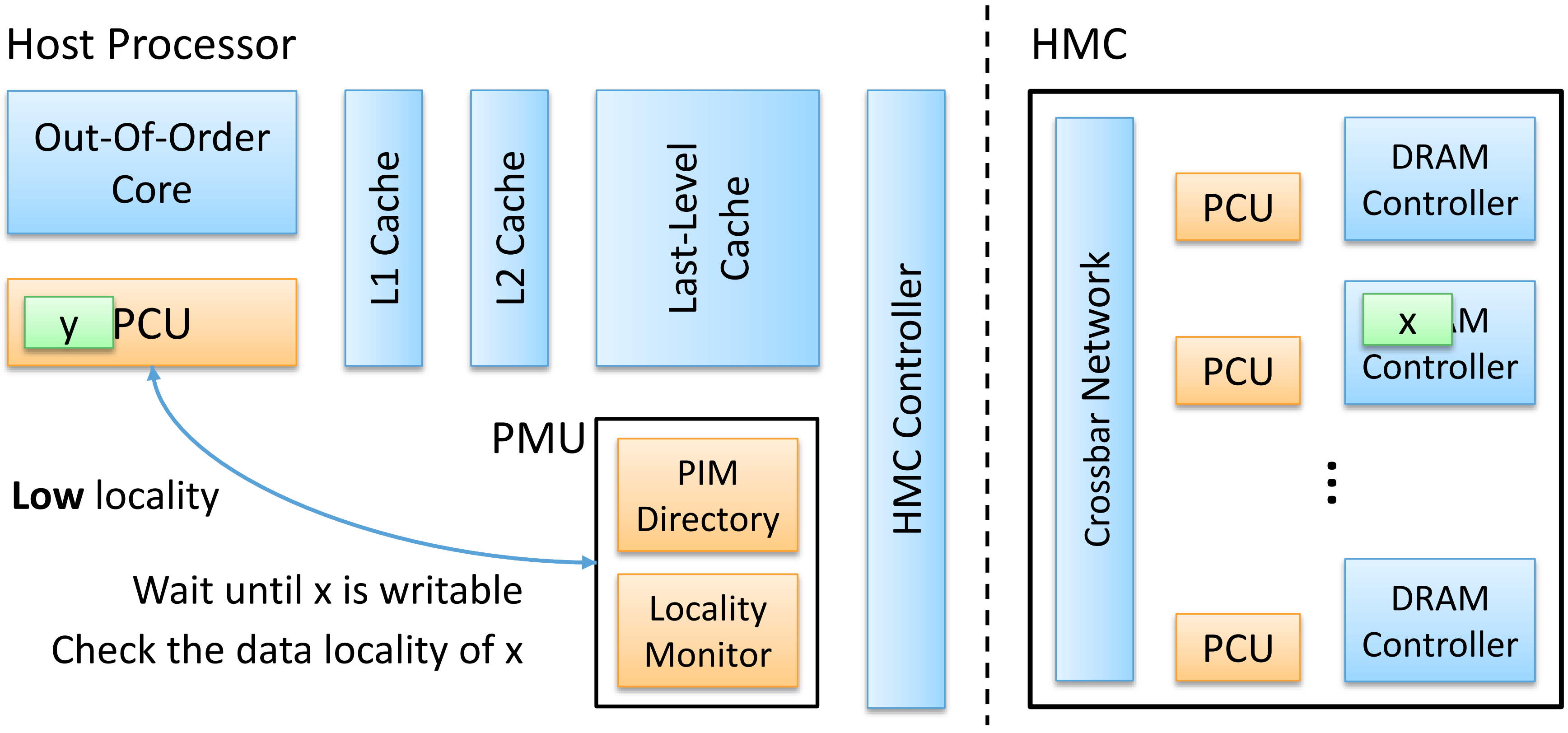
- Each LLC access
- Each issue of a PIM operation to memory

Wait until x is writable

Check the data locality of x

pim.add y, &x

Memory-side PEI Execution



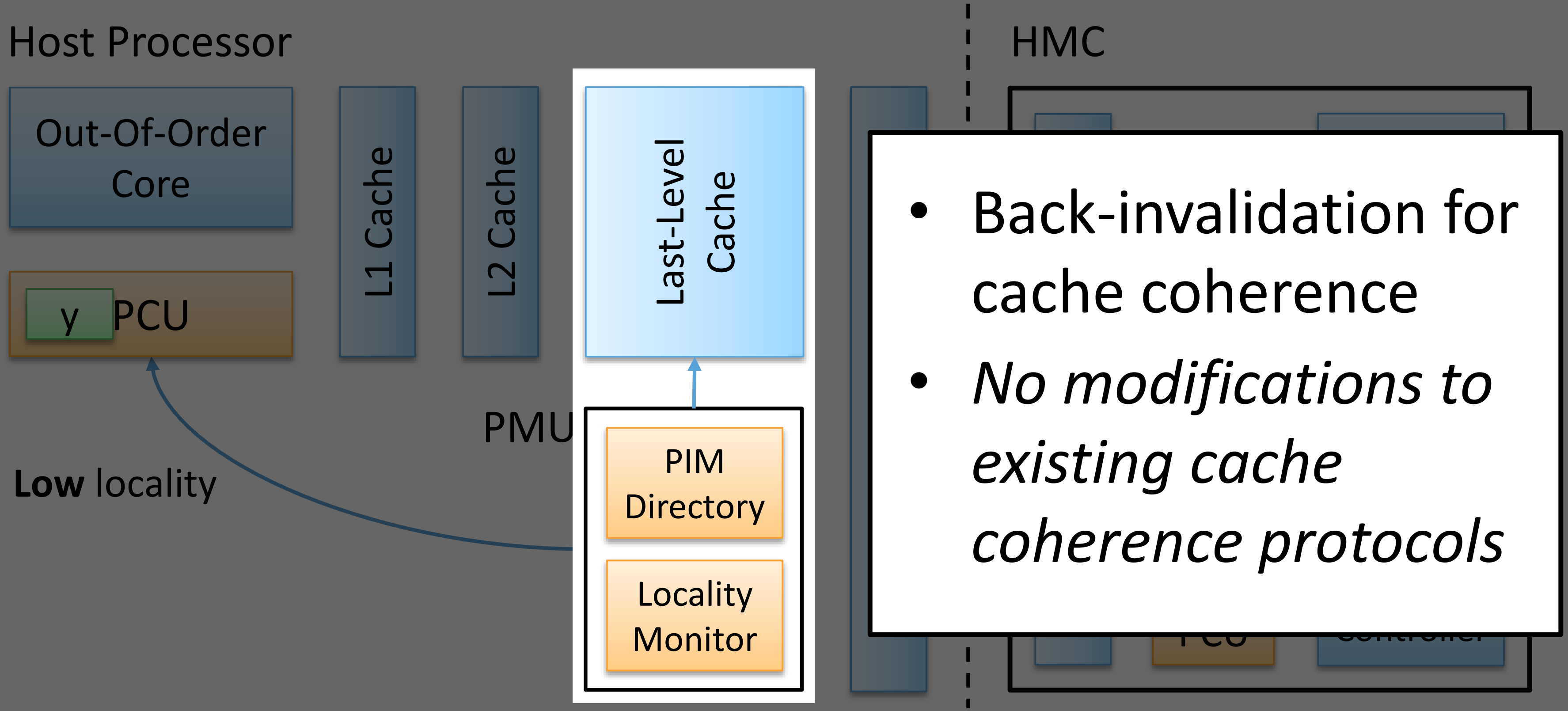
Low locality

Wait until x is writable

Check the data locality of x

pim.add y, &x

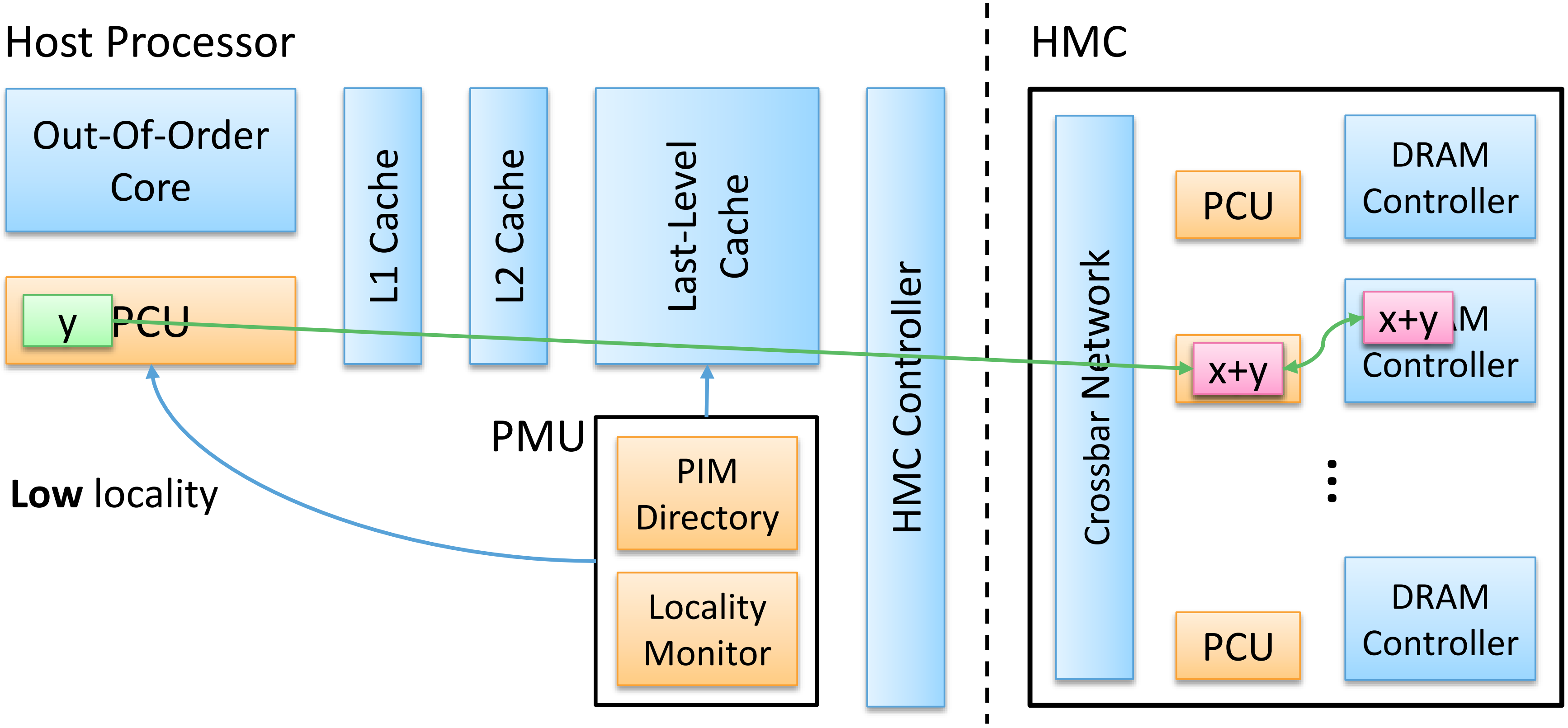
Memory-side PEI Execution



- Back-invalidation for cache coherence
- *No modifications to existing cache coherence protocols*

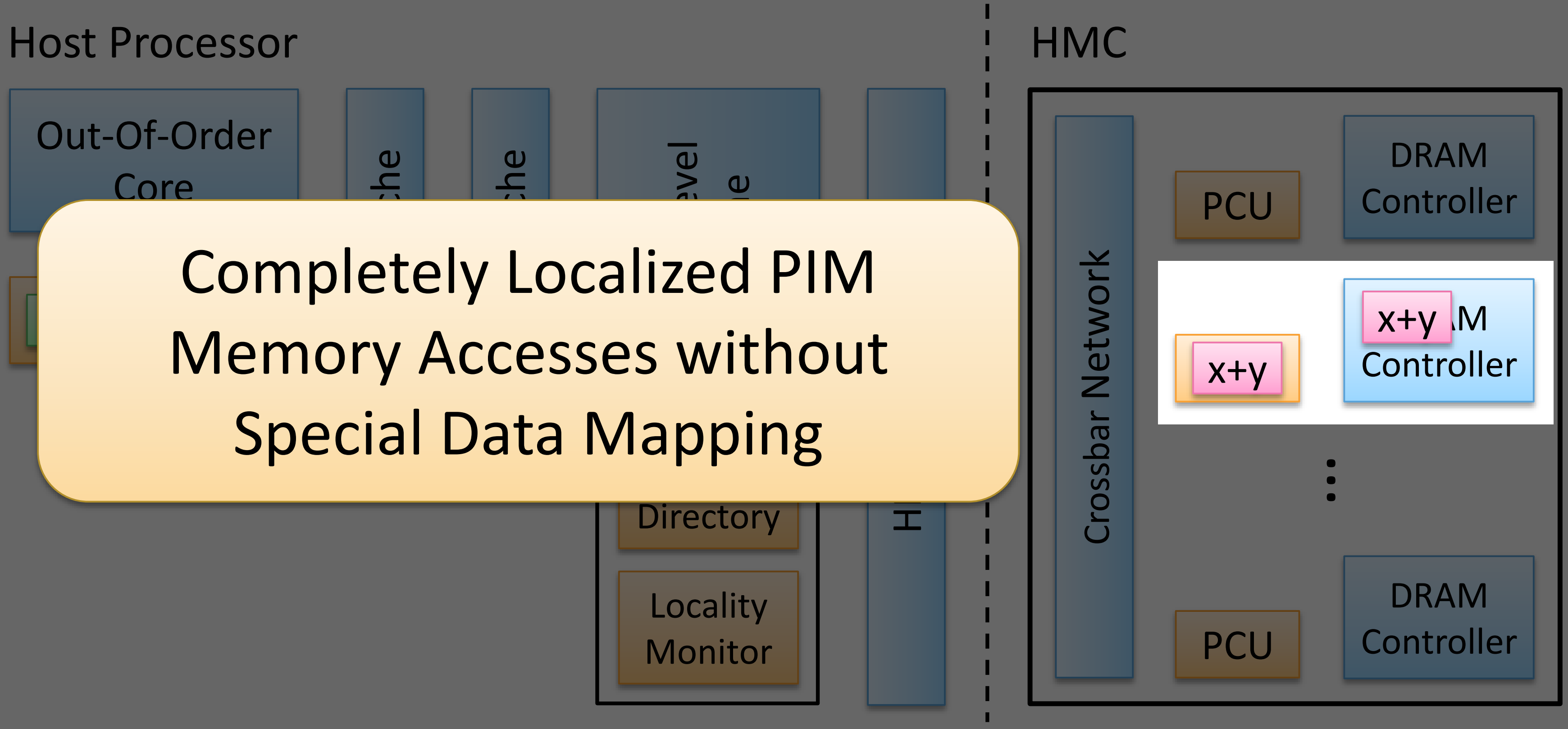
pim.add y, &x

Memory-side PEI Execution



pim.add y, &x

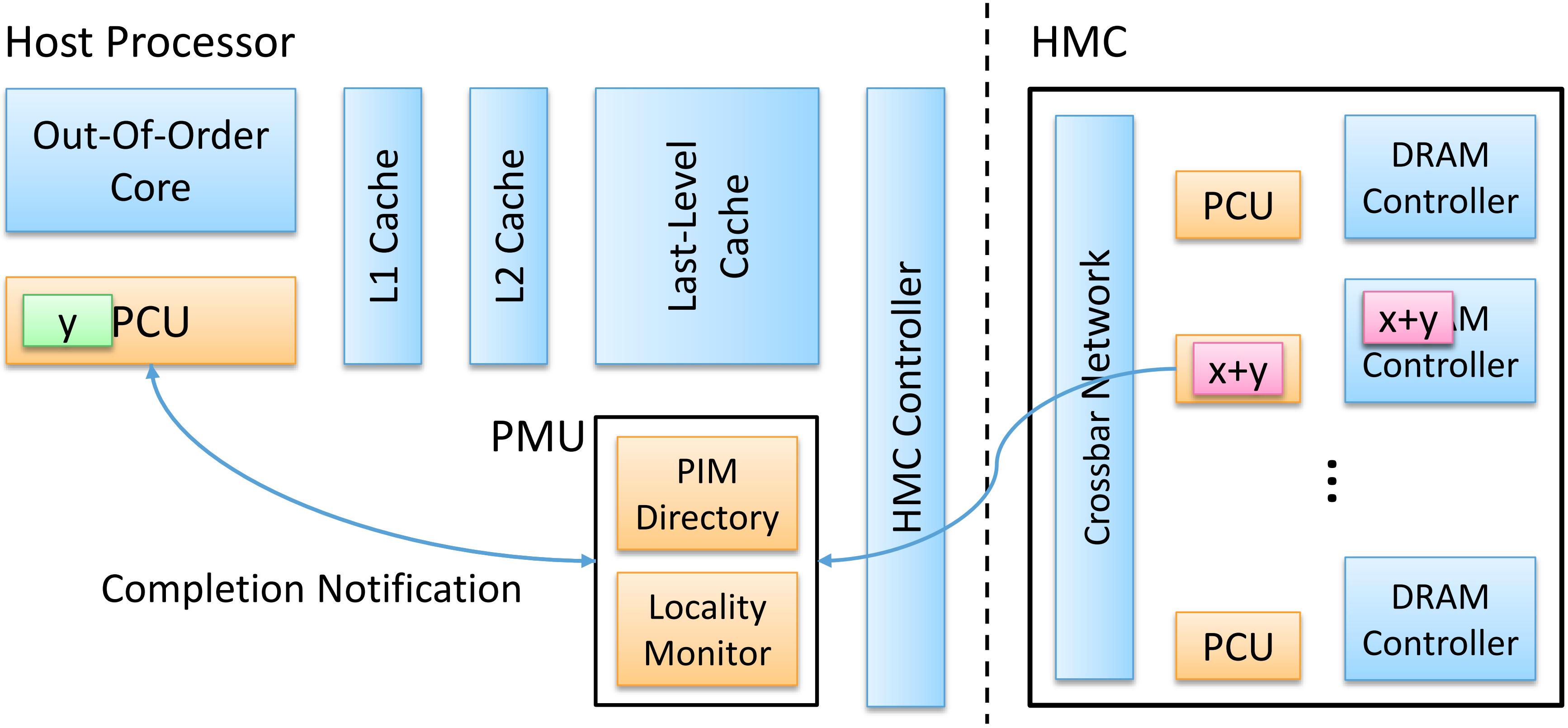
Memory-side PEI Execution



Completely Localized PIM Memory Accesses without Special Data Mapping

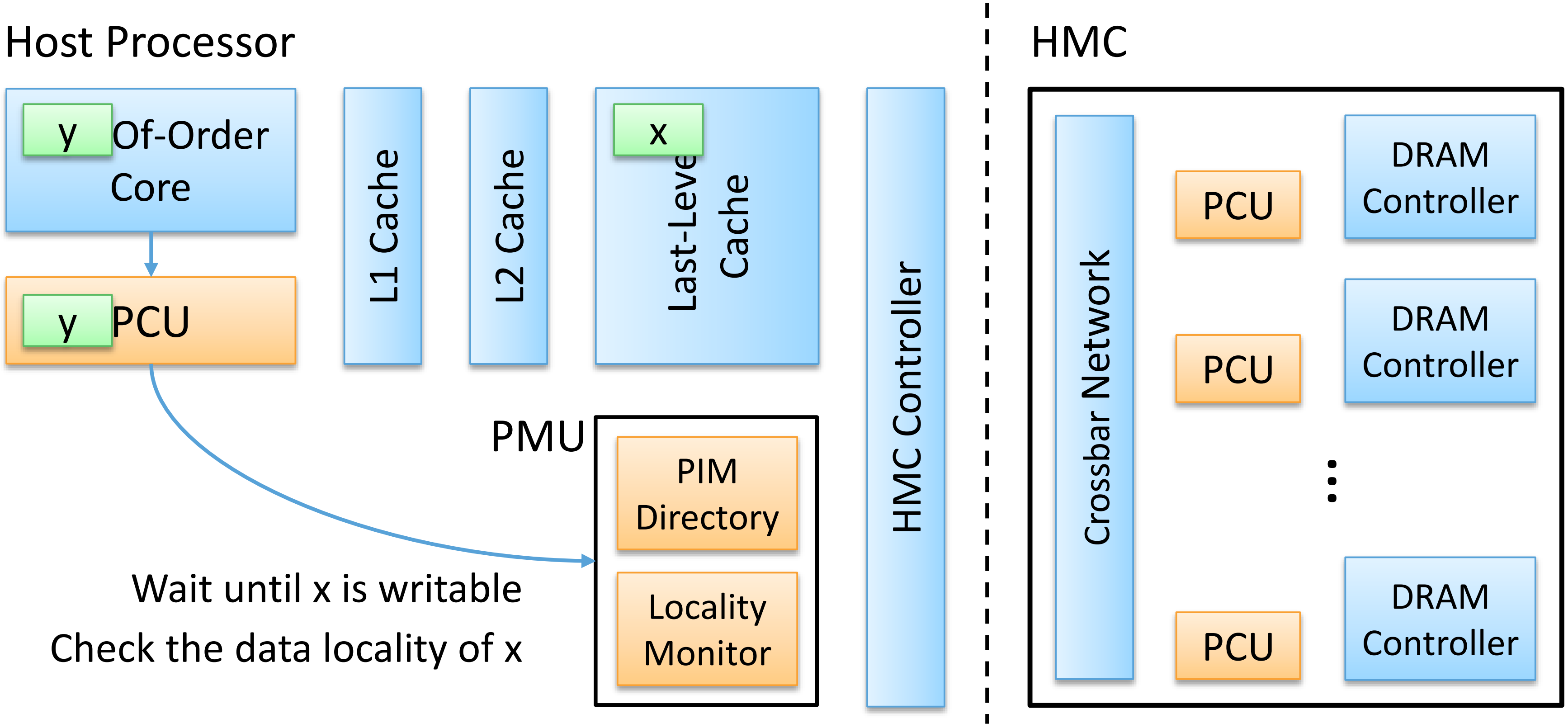
pim.add y, &x

Memory-side PEI Execution



pim.add y, &x

Host-side PEI Execution

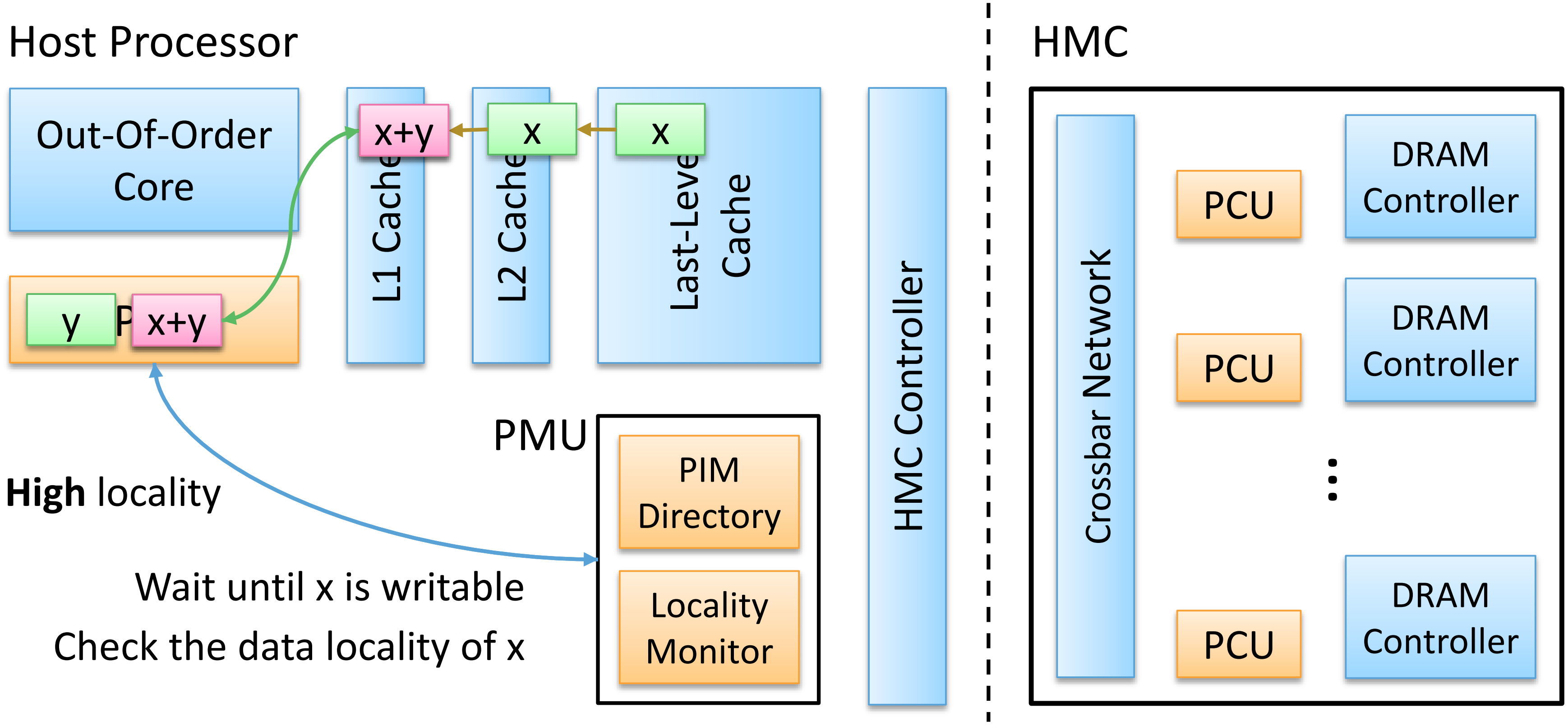


Wait until x is writable

Check the data locality of x

pim.add y, &x

Host-side PEI Execution



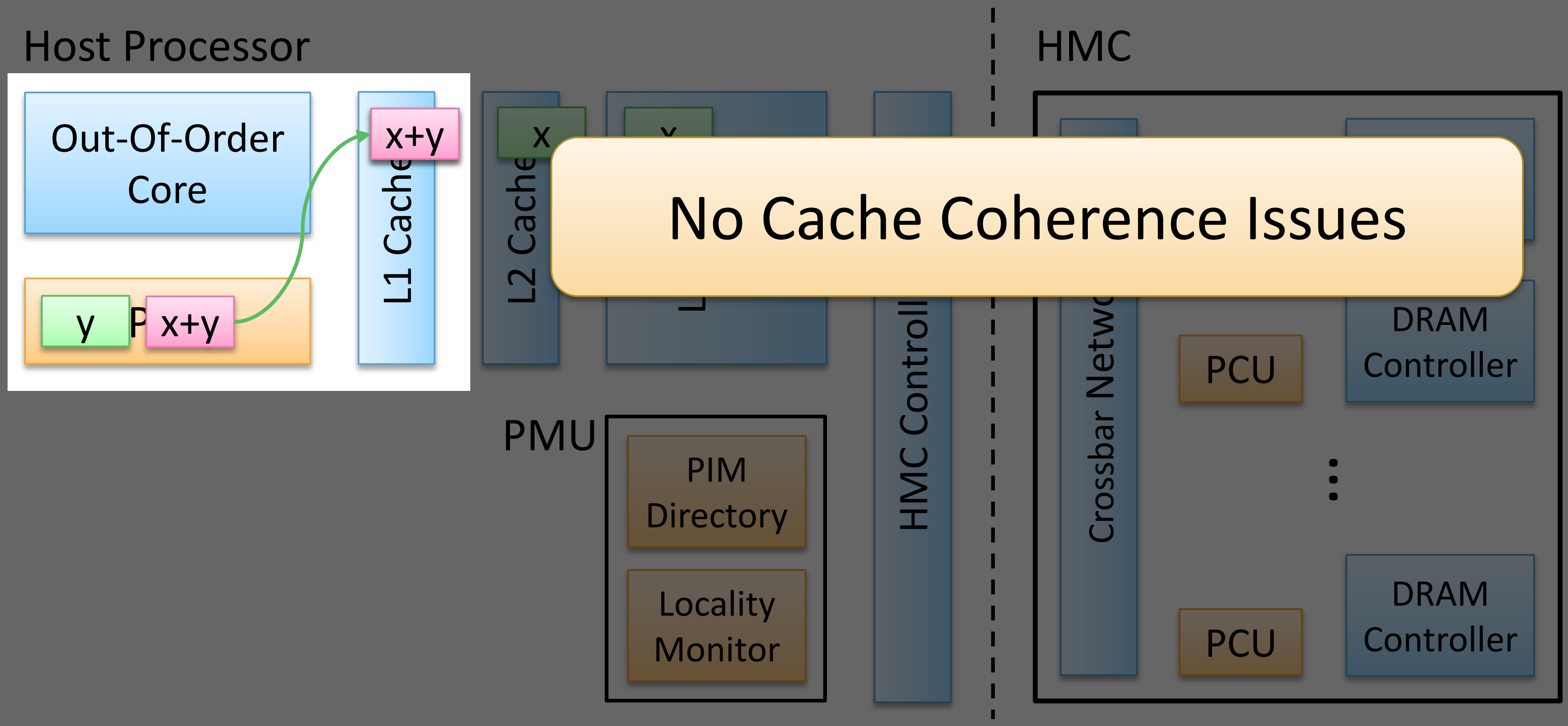
High locality

Wait until x is writable

Check the data locality of x

pim.add y, &x

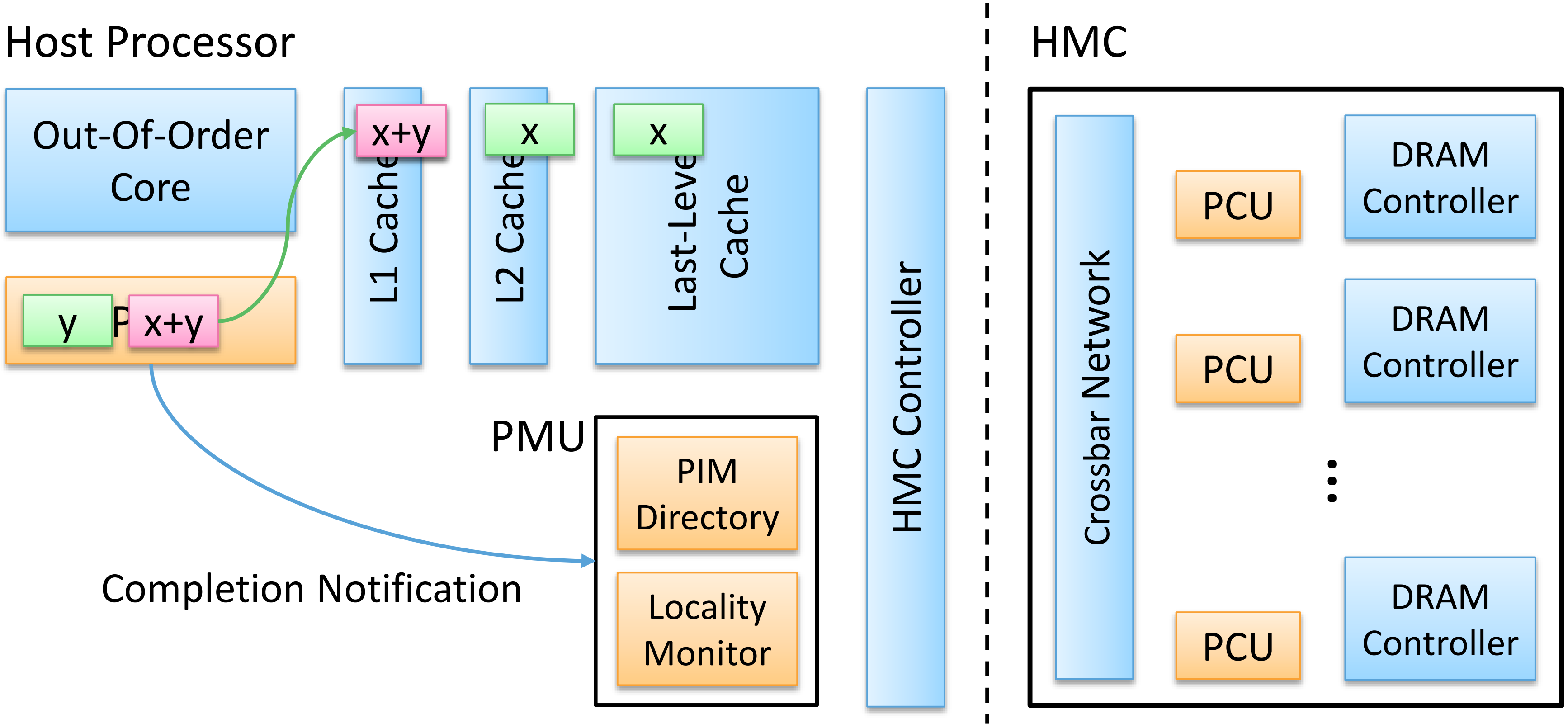
Host-side PEI Execution



No Cache Coherence Issues

pim.add y, &x

Host-side PEI Execution



pim.add y, &x

Mechanism Summary

- Atomicity of PEIs
  - PIM directory implements reader-writer locks
- Locality-aware PEI execution
  - Locality monitor simulates cache replacement behavior
- Cache coherence for PEIs
  - Memory-side: back-invalidation/back-writeback
  - Host-side: no need for consideration
- Virtual memory for PEIs
  - Host processor performs address translation before issuing a PEI

Simulation Configuration

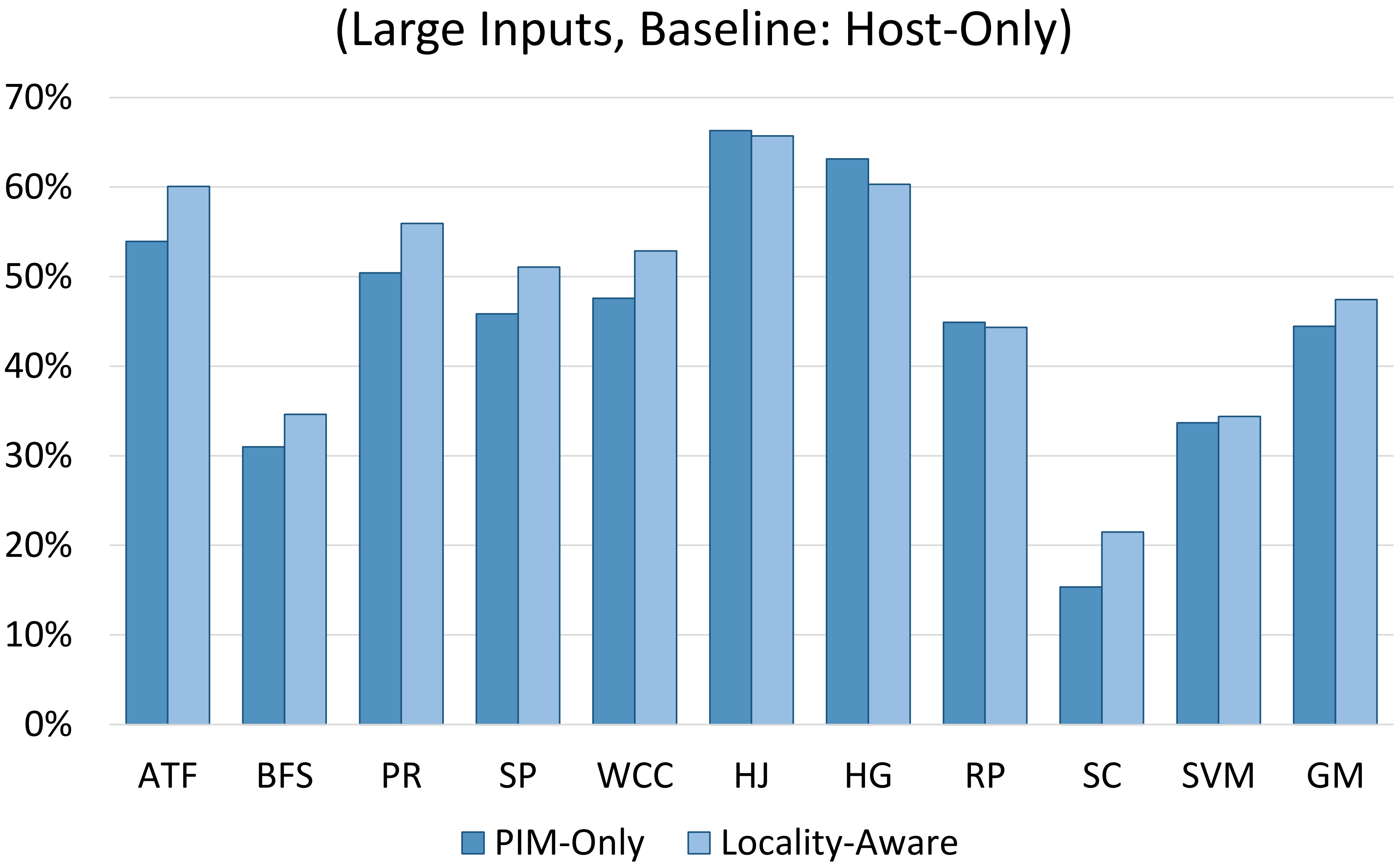
- In-house x86-64 simulator based on Pin
  - 16 out-of-order cores, 4GHz, 4-issue
  - 32KB private L1 I/D-cache, 256KB private L2 cache
  - 16MB shared 16-way L3 cache, 64B blocks
  - 32GB main memory with 8 daisy-chained HMCs (80GB/s)
- PCU
  - 1-issue computation logic, 4-entry operand buffer
  - 16 host-side PCUs at 4GHz, 128 memory-side PCUs at 2GHz
- PMU
  - PIM directory: 2048 entries (3.25KB)
  - Locality monitor: similar to LLC tag array (512KB)

Target Applications

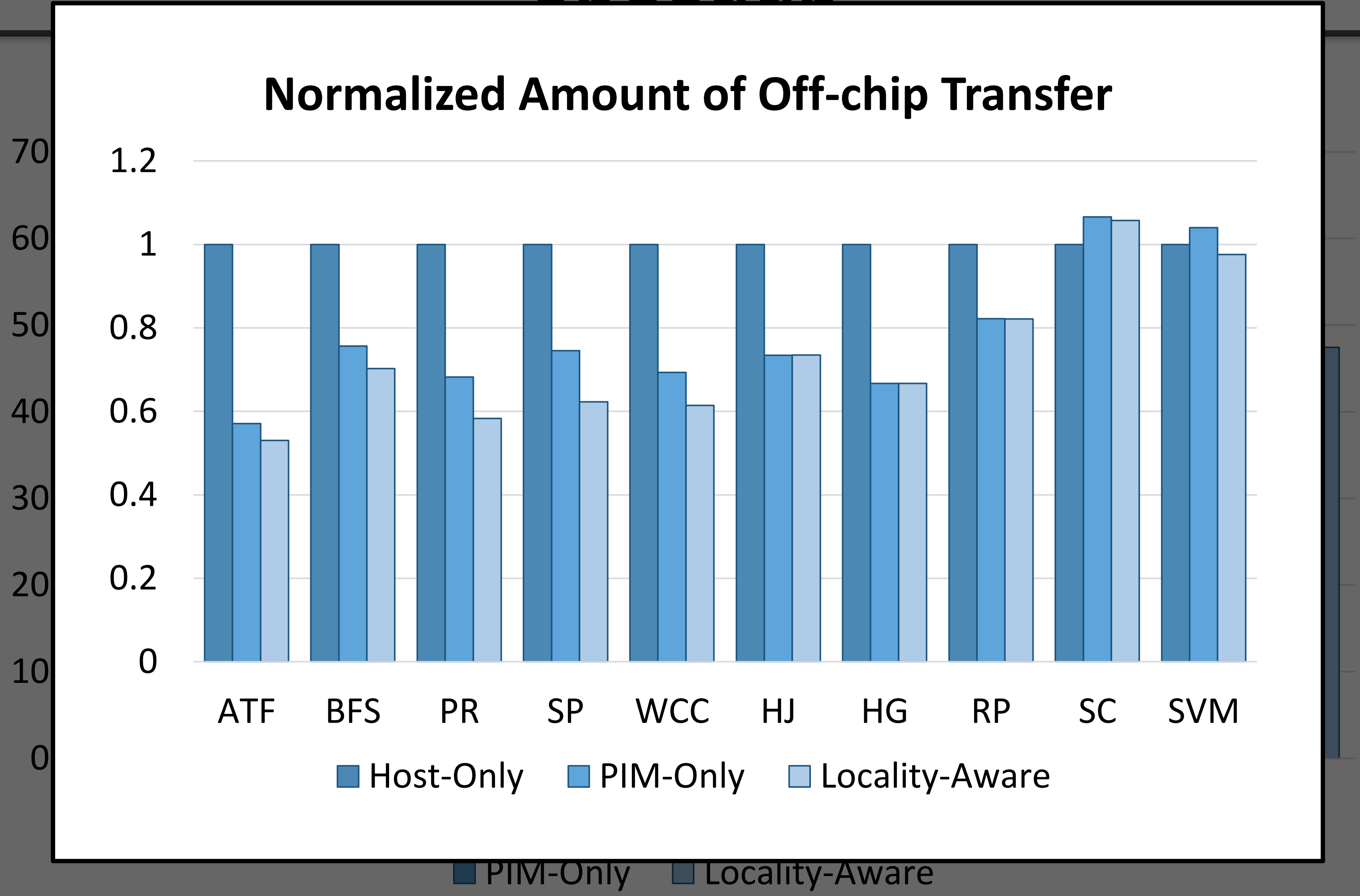
- Ten emerging data-intensive workloads
  - Large-scale graph processing
    - Average teenage followers, BFS, PageRank, single-source shortest path, weakly connected components
  - In-memory data analytics
    - Hash join, histogram, radix partitioning
  - Machine learning and data mining
    - Streamcluster, SVM-RFE

- Three input sets (small, medium, large) for each workload to show the impact of data locality

Speedup

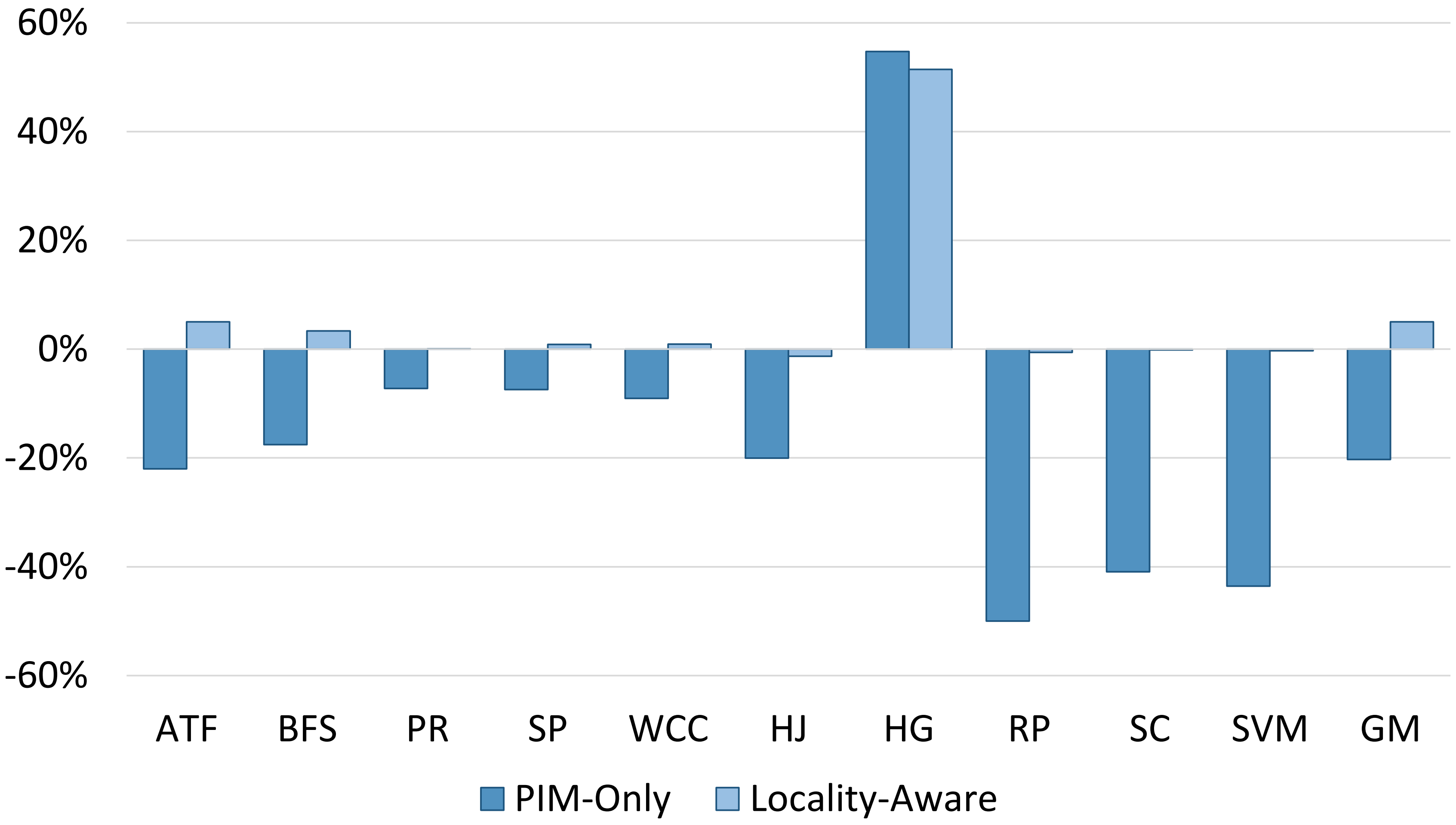

(Large Inputs, Baseline: Host-Only)
70%
60%
50%
40%
30%
20%
10%
0%
ATF
BFS
PR
SP
WCC
HJ
HG
RP
SC
SVM
GM
PIM-Only
Locality-Aware

Speedup

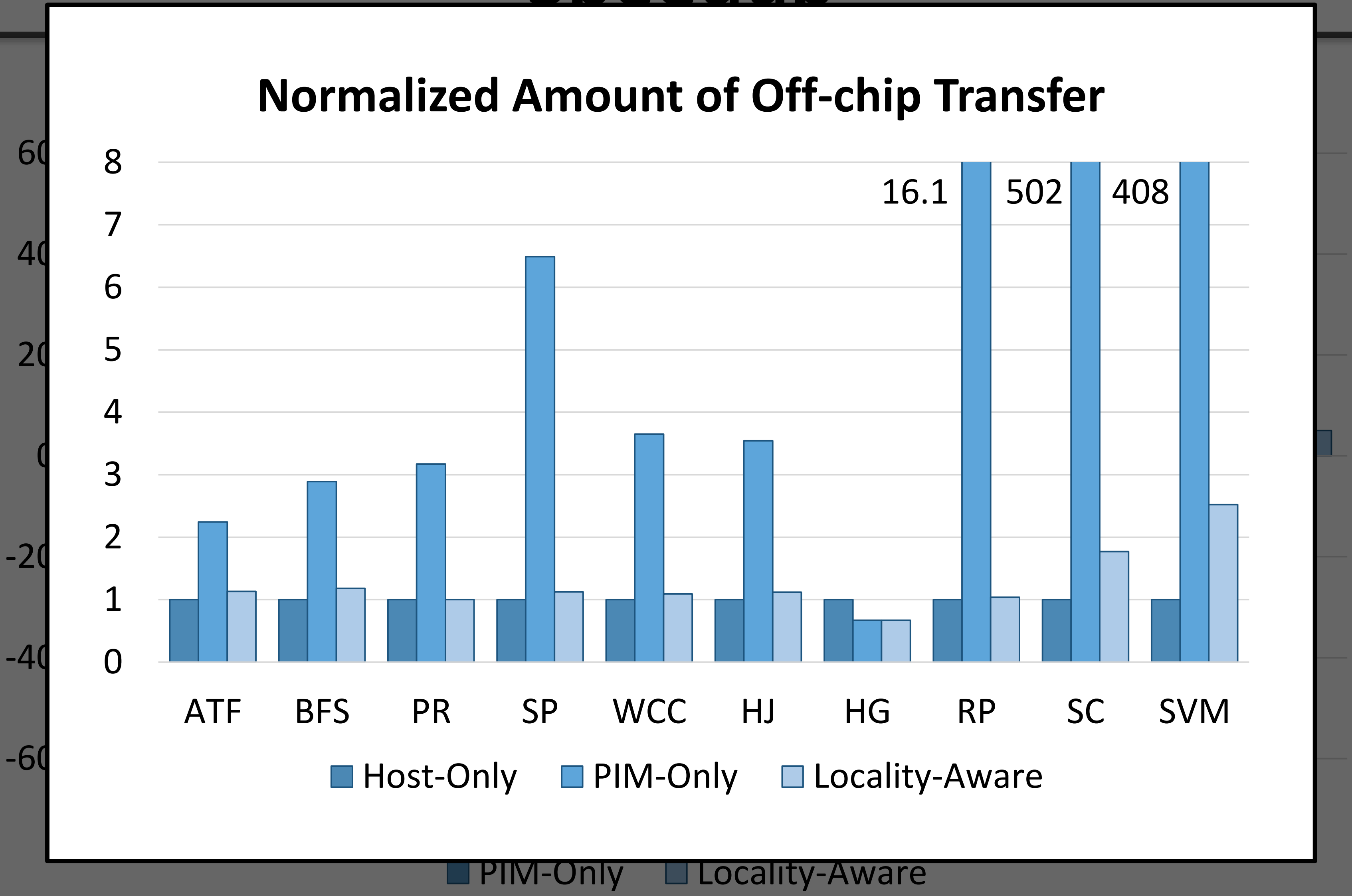

Normalized Amount of Off-chip Transfer
1.2
1
0.8
0.6
0.4
0.2
0
ATF
BFS
PR
SP
WCC
HJ
HG
RP
SC
SVM
Host-Only
PIM-Only
Locality-Aware
70
60
50
40
30
20
10
0
PIM-Only
Locality-Aware

Speedup

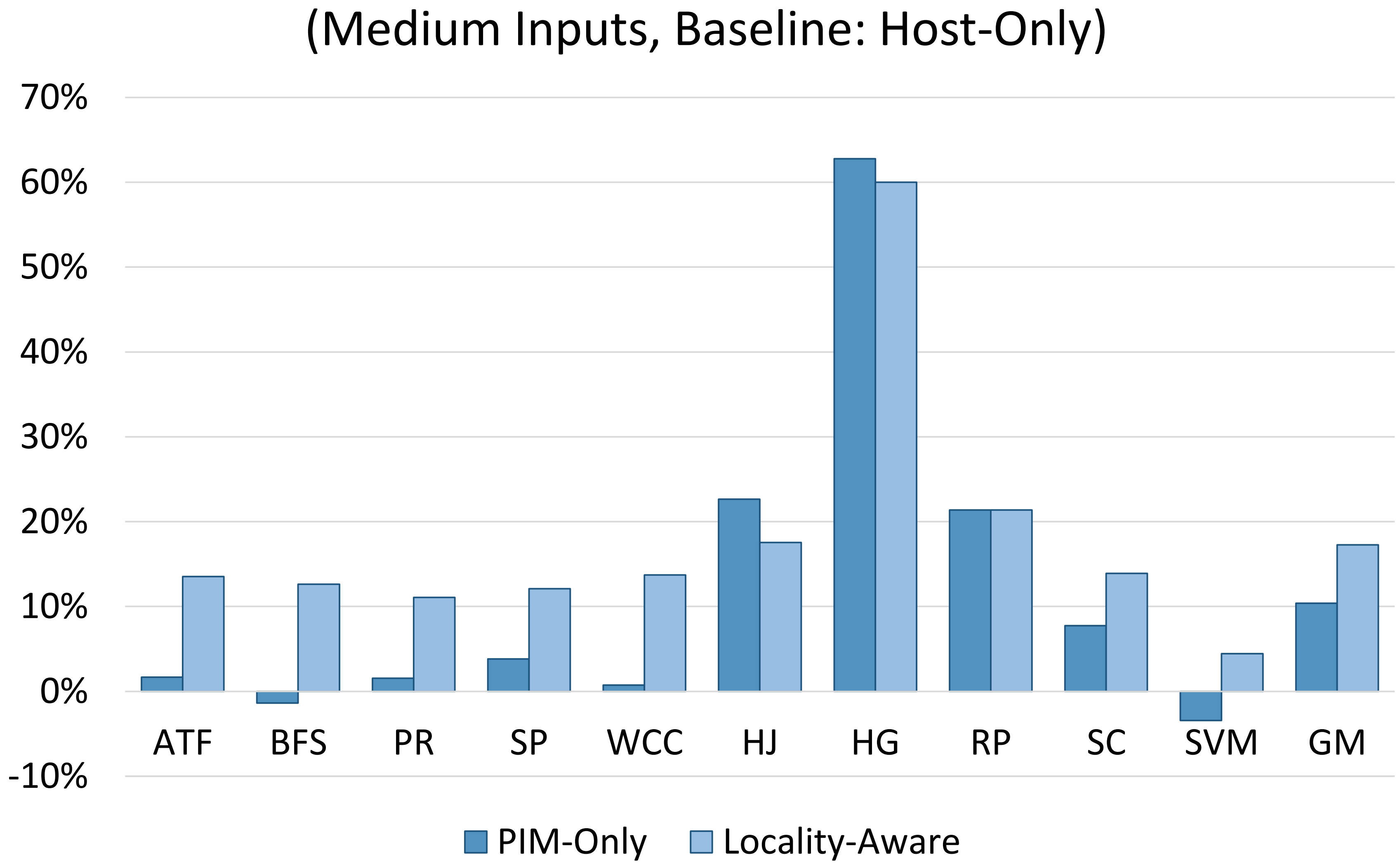
(Small Inputs, Baseline: Host-Only)



Speedup

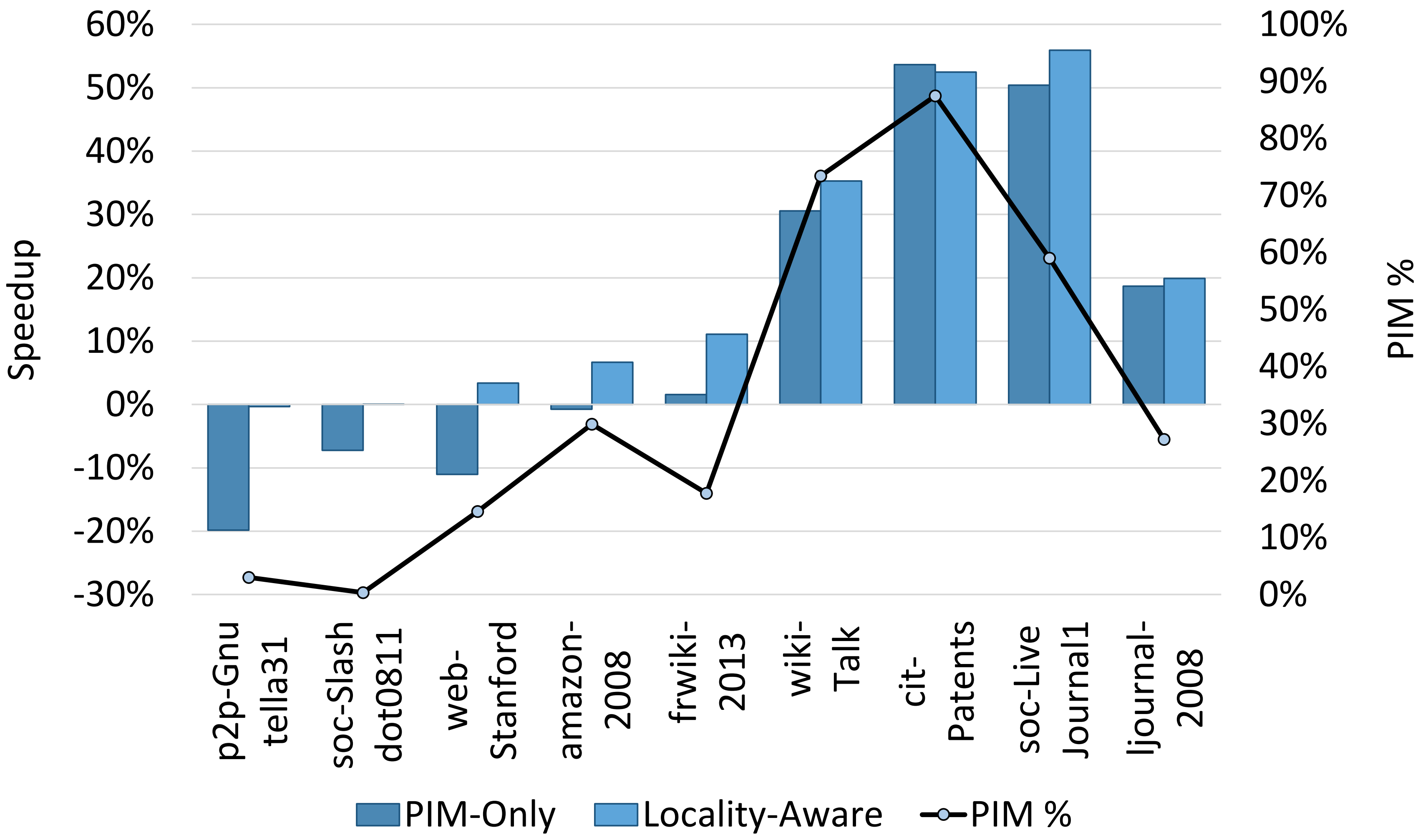

Normalized Amount of Off-chip Transfer
0
1
2
3
4
5
6
7
8
16.1
502
408
ATF
BFS
PR
SP
WCC
HJ
HG
RP
SC
SVM
Host-Only
PIM-Only
Locality-Aware
60
40
20
0
-20
-40
-60
PIM-Only
Locality-Aware

Speedup

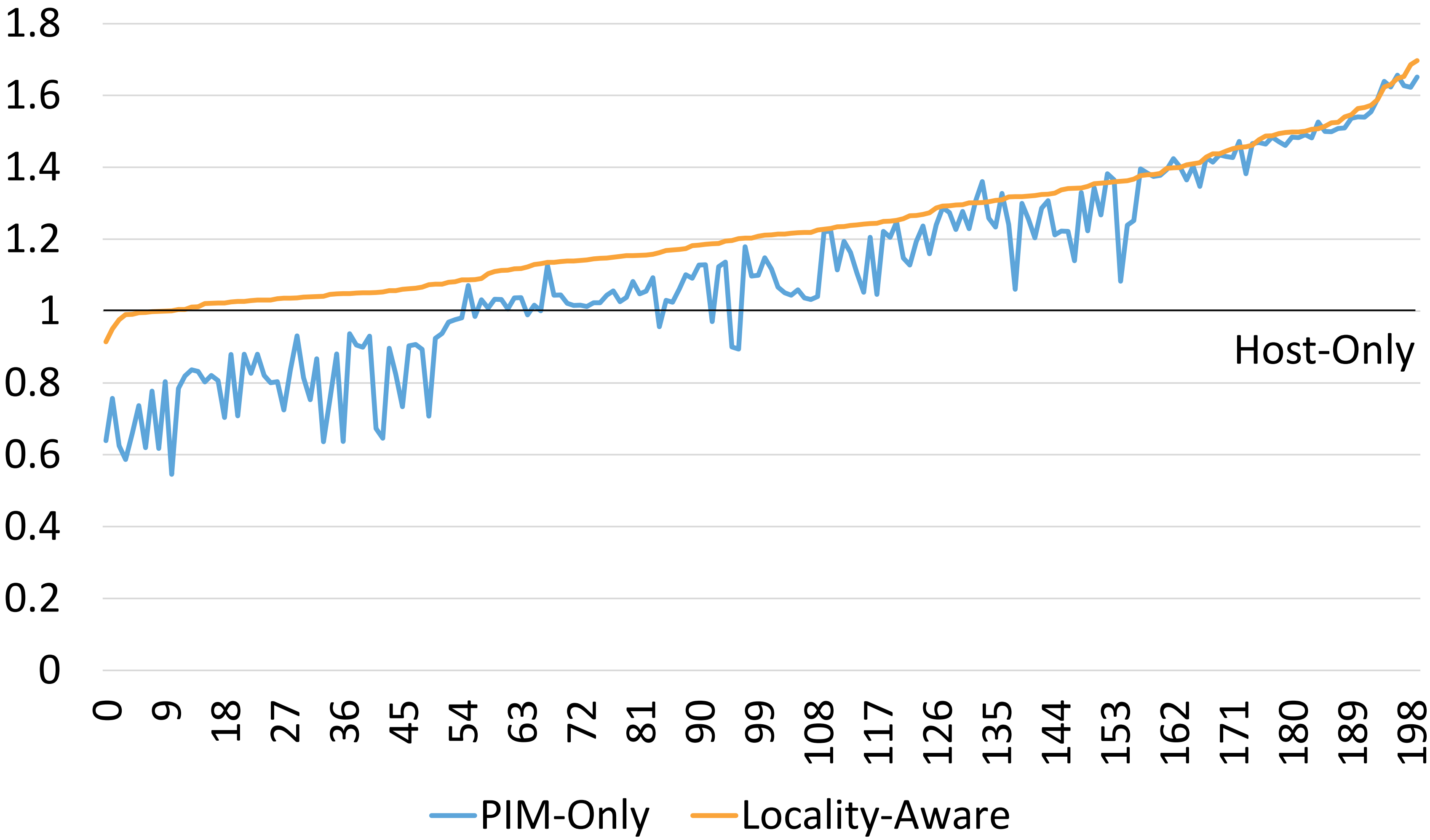


(Medium Inputs, Baseline: Host-Only)
70%
60%
50%
40%
30%
20%
10%
0%
-10%
ATF
BFS
PR
SP
WCC
HJ
HG
RP
SC
SVM
GM
PIM-Only
Locality-Aware

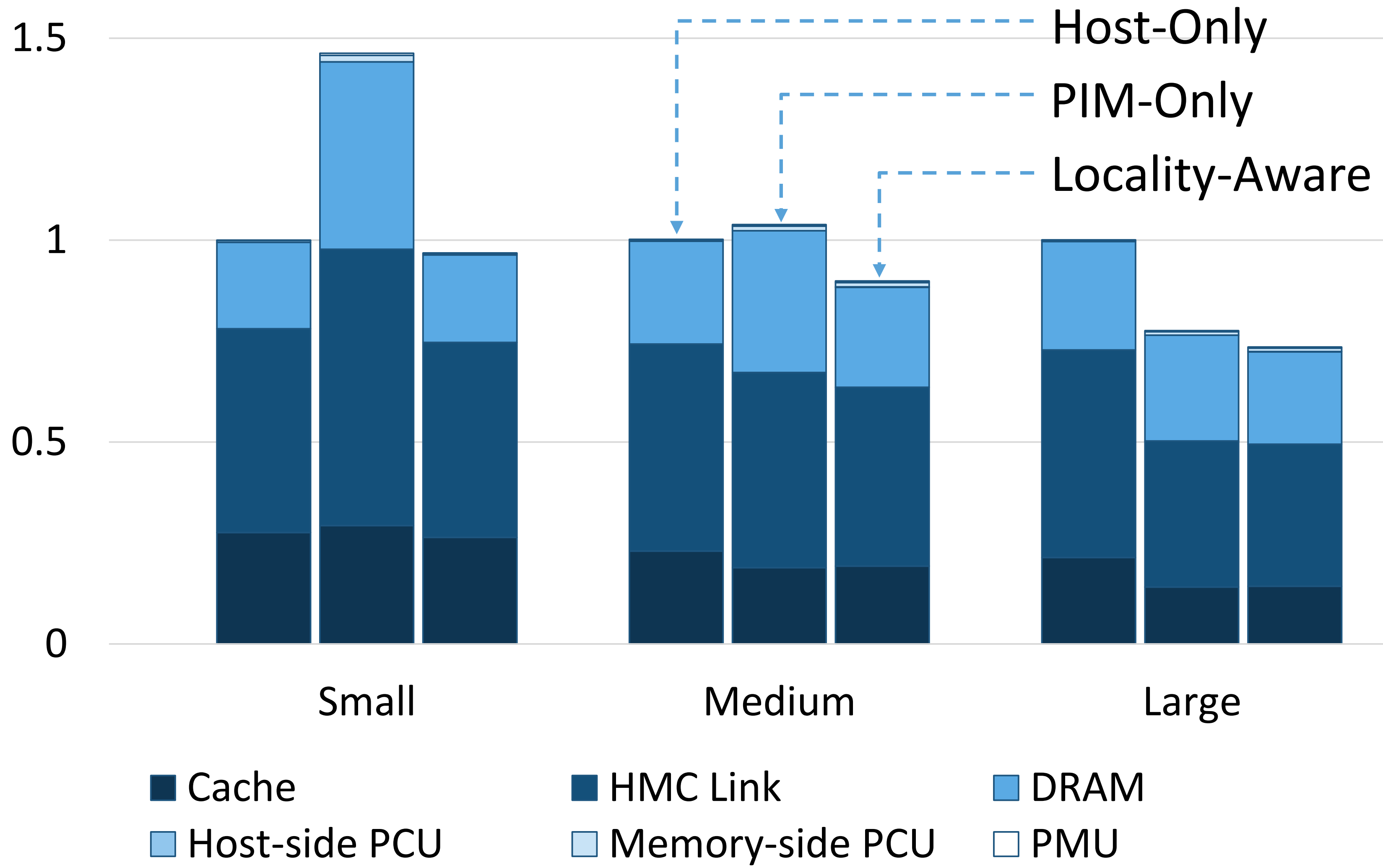
Sensitivity to Input Size



Multiprogrammed Workloads



Energy Consumption



Conclusion

- Challenges of PIM architecture design
  - Cost-effective integration of logic and memory
  - Unconventional programming models
  - Lack of interoperability with caches and virtual memory
- PIM-enabled instruction: low-cost PIM abstraction & HW
  - Interfaces PIM operations as ISA extension
  - Simplifies cache coherence and virtual memory support for PIM
  - Locality-aware execution of PIM operations
- Evaluations
  - 47%/32% speedup over Host/PIM-Only in large/small inputs
  - Good adaptivity across randomly generated workloads

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware PIM Architecture

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu[+], and Kiyoung Choi

Seoul National University [+]Carnegie Mellon University