# 18-740: Computer Architecture Recitation 3: Rethinking Memory System Design

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2015

September 15, 2015

# Agenda

- Review Assignments for Next Week

- Rethinking Memory System Design (Continued)
  - With a lot of discussion, hopefully

# Review Assignments for Next Week

# Required Reviews

- Due Tuesday Sep 22 @ 3pm
- Enter your reviews on the review website
- Please discuss ideas and thoughts on Piazza

# Review Paper 1 (Required)

- Yu Cai, Yixin Luo, Saugata Ghose, Erich F. Haratsch, Ken Mai, and Onur Mutlu,
  **"Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation"**
  *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (**DSN**)*, Rio de Janeiro, Brazil, June 2015.
  [Slides (pptx) (pdf)]

- Related
  - Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu,
    **"Data Retention in MLC NAND Flash Memory: Characterization, Optimization and Recovery"**
    *Proceedings of the 21st International Symposium on High-Performance Computer Architecture (**HPCA**)*, Bay Area, CA, February 2015.
    [Slides (pptx) (pdf)]

# Review Paper 2 (Required)

- Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger,
  **"Architecting Phase Change Memory as a Scalable DRAM Alternative"**
  *Proceedings of the 36th International Symposium on Computer Architecture (**ISCA**)*, pages 2-13, Austin, TX, June 2009. Slides (pdf)

- Related
  - Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger,
    **"Phase Change Technology and the Future of Main Memory"**
    *IEEE Micro, Special Issue: Micro's Top Picks from 2009 Computer Architecture Conferences (**MICRO TOP PICKS**)*, Vol. 30, No. 1, pages 60-70, January/February 2010.

# Review Paper 3 (Required)

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, **"Bottleneck Identification and Scheduling in Multithreaded Applications"**
  *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, London, UK, March 2012. Slides (ppt) (pdf)

- Related
  - M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt,
    **"Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"**
    *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 253-264, Washington, DC, March 2009. Slides (ppt)

# Review Paper 4 (Optional)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
  **"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"**
  *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)

# Project Proposal

- Due next week
  - September 25, 2015

# Another Possible Project

- GPU Warp Scheduling Championship

- http://adwaitjog.github.io/gpu_scheduling.html

# Rethinking Memory System Design

# Some Promising Directions

- New memory architectures
  - Rethinking DRAM and flash memory
  - **A lot of hope in fixing DRAM**

- Enabling emerging NVM technologies
  - Hybrid memory systems
  - Single-level memory and storage
  - **A lot of hope in hybrid memory systems and single-level stores**

- System-level memory/storage QoS
  - **A lot of hope in designing a predictable system**

# Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies
- How Can We Do Better?
- Summary

# Rethinking DRAM

- In-Memory Computation
- Refresh
- Reliability
- Latency
- Bandwidth
- Energy
- Memory Compression

# Recap: The DRAM Scaling Problem

## DRAM Process Scaling Challenges

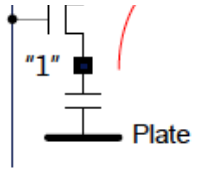
- **Refresh**
  - Difficult to build high-aspect ratio cell capacitors decreasing cell capacitance

THE MEMORY FORUM 2014

# Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling

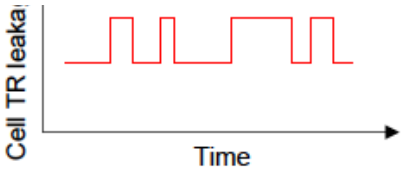
Uksong Kang, Hak-soo Yu, Churoo Park, *Hongzhong Zheng,
**John Halbert, **Kuljit Bains, SeongJin Jang, and Joo Sun Choi

*Samsung Electronics, Hwasung, Korea / *Samsung Electronics, San Jose / **Intel*




**Refresh**


**tWR**

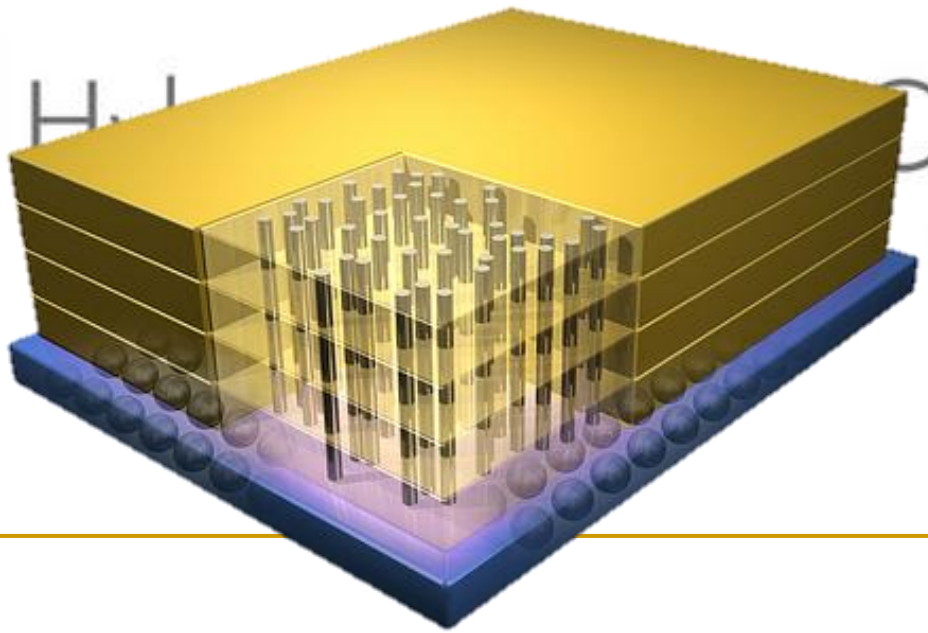



**VRT**

# Takeaways So Far

- DRAM Scaling is getting extremely difficult
  - To the point of threatening the foundations of secure systems

- Industry is very open to "different" system designs and "different" memories
  - Cost-per-bit is not the sole driving force any more

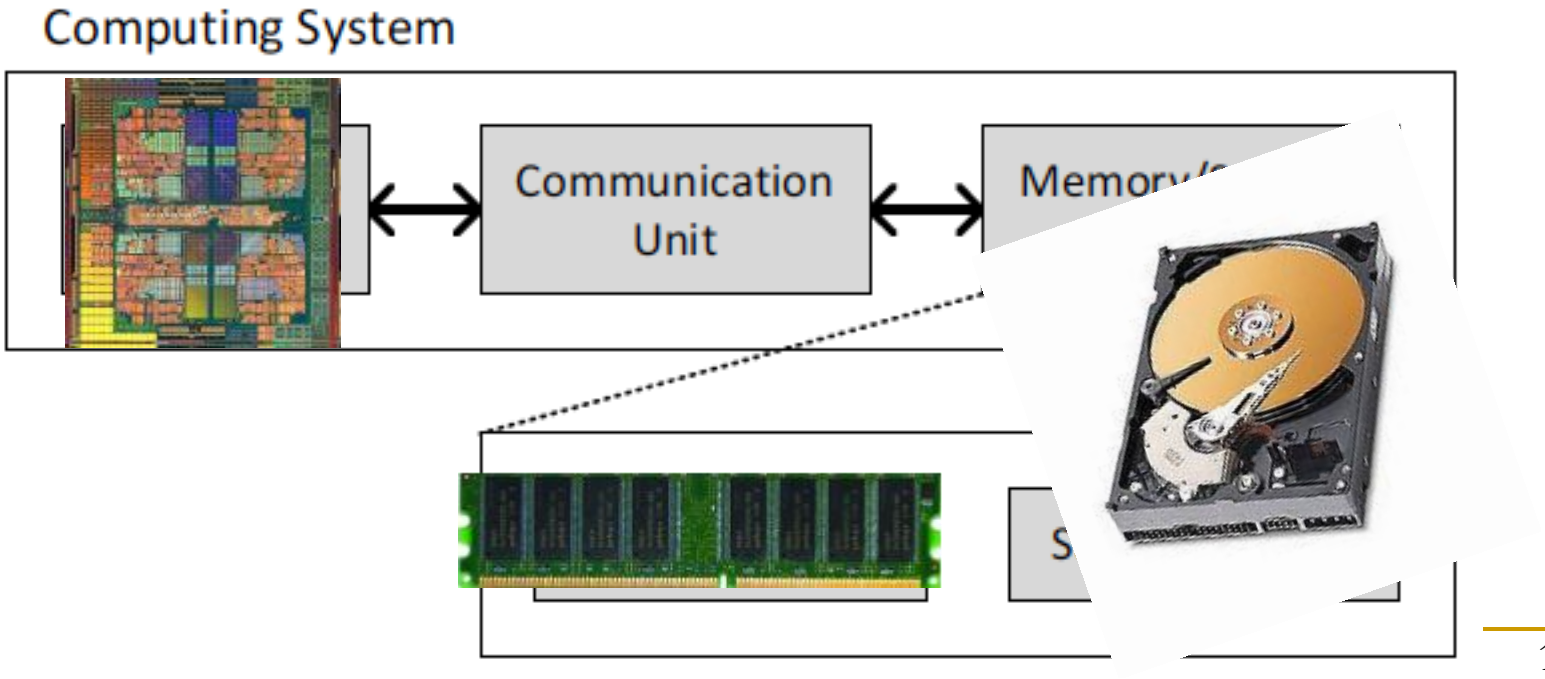


# Why In-Memory Computation Today?

- Push from Technology Trends
  - DRAM Scaling at jeopardy
    → Controllers close to DRAM
    → Industry open to new memory architectures
- Pull from Systems and Applications Trends
  - Data access is a major system and application bottleneck
  - Systems are energy limited
  - Data movement much more energy-hungry than computation

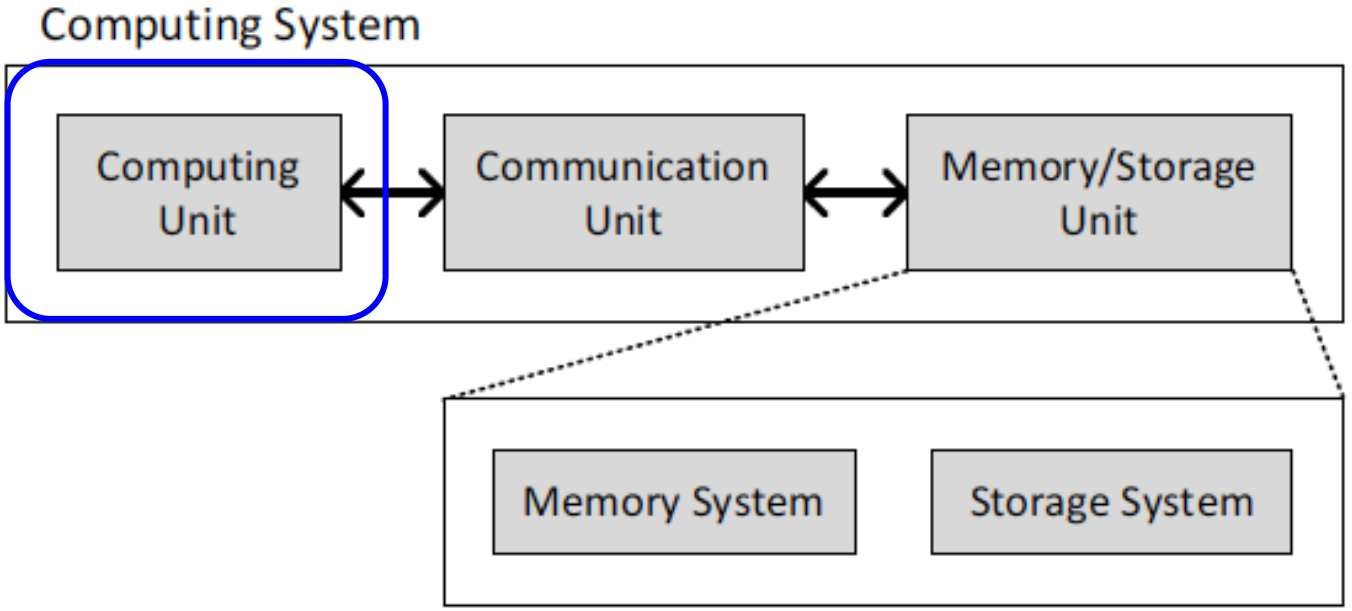
# A Computing System

- Three key components
- Computation
- Communication
- Storage/memory



# Today's Computing Systems

- Are overwhelmingly processor centric
- Processor is heavily optimized and is considered the master
- Many system-level tradeoffs are constrained or dictated by the processor – all data processed in the processor
- Data storage units are dumb slaves and are largely unoptimized (except for some that are on the processor die)



# Traditional Computing Systems

- Data stored far away from computational units
- Bring data to the computational units
- Operate on the brought data
- Cache it as much as possible
- Send back the results to data storage

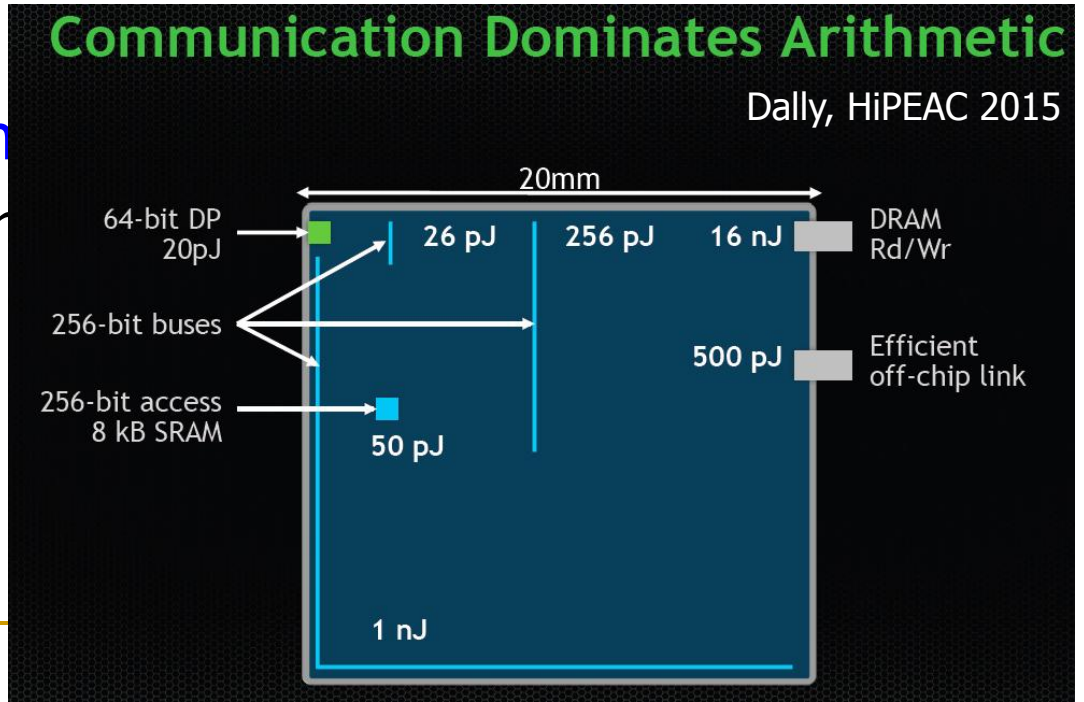
- This may not be an efficient approach given three key systems trends

# Three Key Systems Trends

1. Data access from memory is a major bottleneck
   - Limited pin bandwidth
   - High energy memory bus
   - Applications are increasingly data hungry

2. Energy consumption is a key limiter in systems

3. Data movement is much
   - Especially true for off-ch



# The Problem

- Today's systems overwhelmingly move data towards computation, exercising the three bottlenecks

- This is a huge problem when the amount of data access is huge relative to the amount of computation
  - The case with many data-intensive workloads

36 Million
Wikipedia Pages


1.4 Billion
Facebook Users

300 Million
Twitter Users

30 Billion
Instagram Photos

# In-Memory Computation: Goals and Approaches

- Goals
  - Enable computation capability where data resides (e.g., in memory, in caches)
  - Enable system-level mechanisms to exploit near-data computation capability
    - E.g., to decide where it makes the most sense to perform the computation

- Approaches
  1. Minimally change DRAM to enable simple yet powerful computation primitives
  2. Exploit the control logic in 3D-stacked memory to enable more comprehensive computation near memory

# Why This is Not All Déjà Vu

- Past approaches to PIM (e.g., logic-in-memory, NON-VON, Execube, IRAM) had little success due to three reasons:

1. They were too costly. Placing a full processor inside DRAM technology is still not a good idea today.
2. The time was not ripe:
   - Memory scaling was not pressing. Today it is critical.
   - Energy and bandwidth were not critical scalability limiters. Today they are.
   - New technologies were not as prevalent, promising or needed. Today we have 3D stacking, STT-MRAM, etc. which can help with computation near data.
3. They did not consider all issues that limited adoption (e.g., coherence, appropriate partitioning of computation)

# Two Approaches to In-Memory Processing

- 1. Minimally change DRAM to enable simple yet powerful computation primitives
  - RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data (Seshadri et al., MICRO 2013)
  - Fast Bulk Bitwise AND and OR in DRAM (Seshadri et al., IEEE CAL 2015)

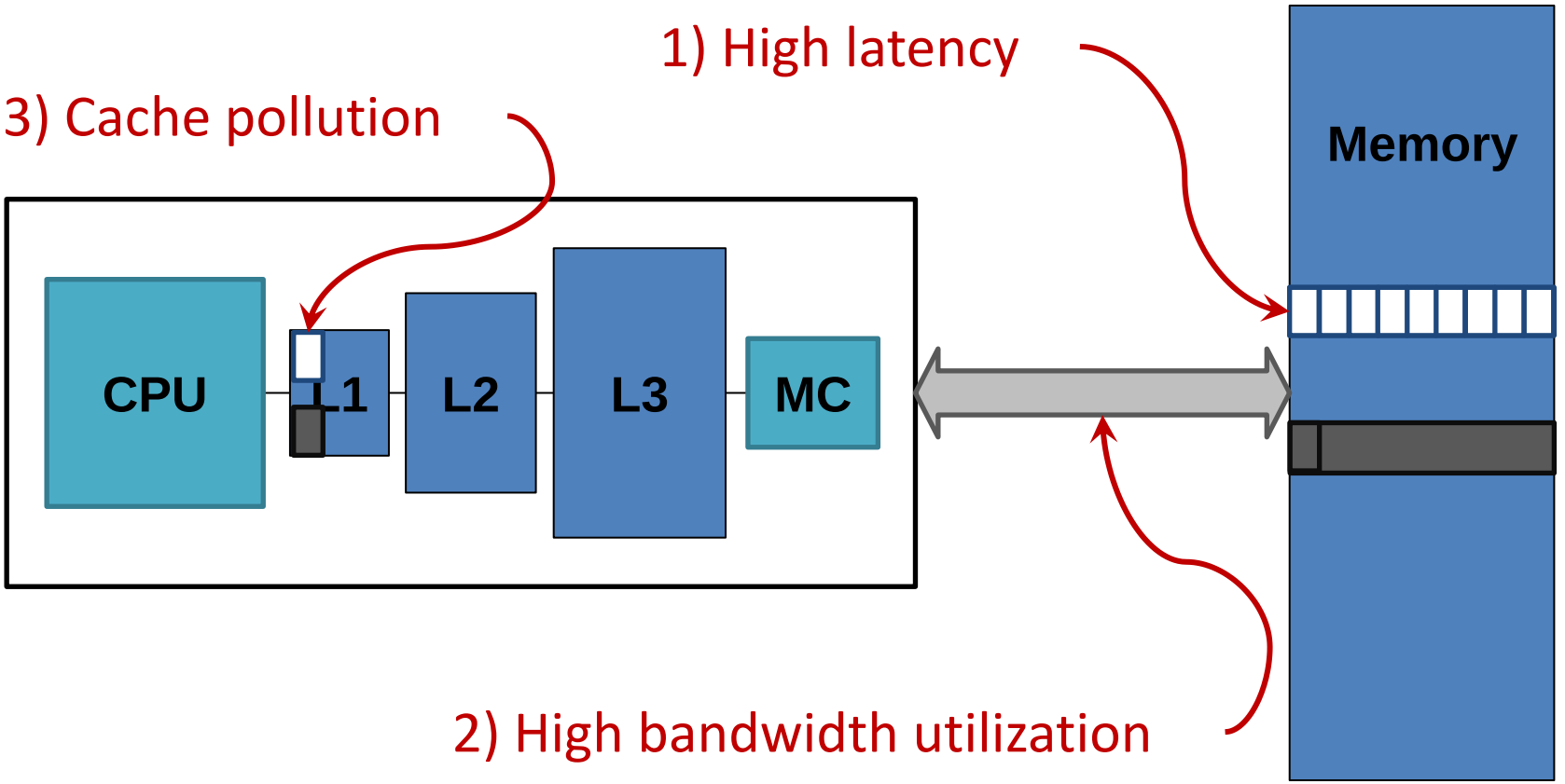
- 2. Exploit the control logic in 3D-stacked memory to enable more comprehensive computation near memory

# Approach 1: Minimally Changing DRAM

- DRAM has great capability to perform bulk data movement and computation internally with small changes
  - Can exploit internal bandwidth to move data
  - Can exploit analog computation capability
  - …

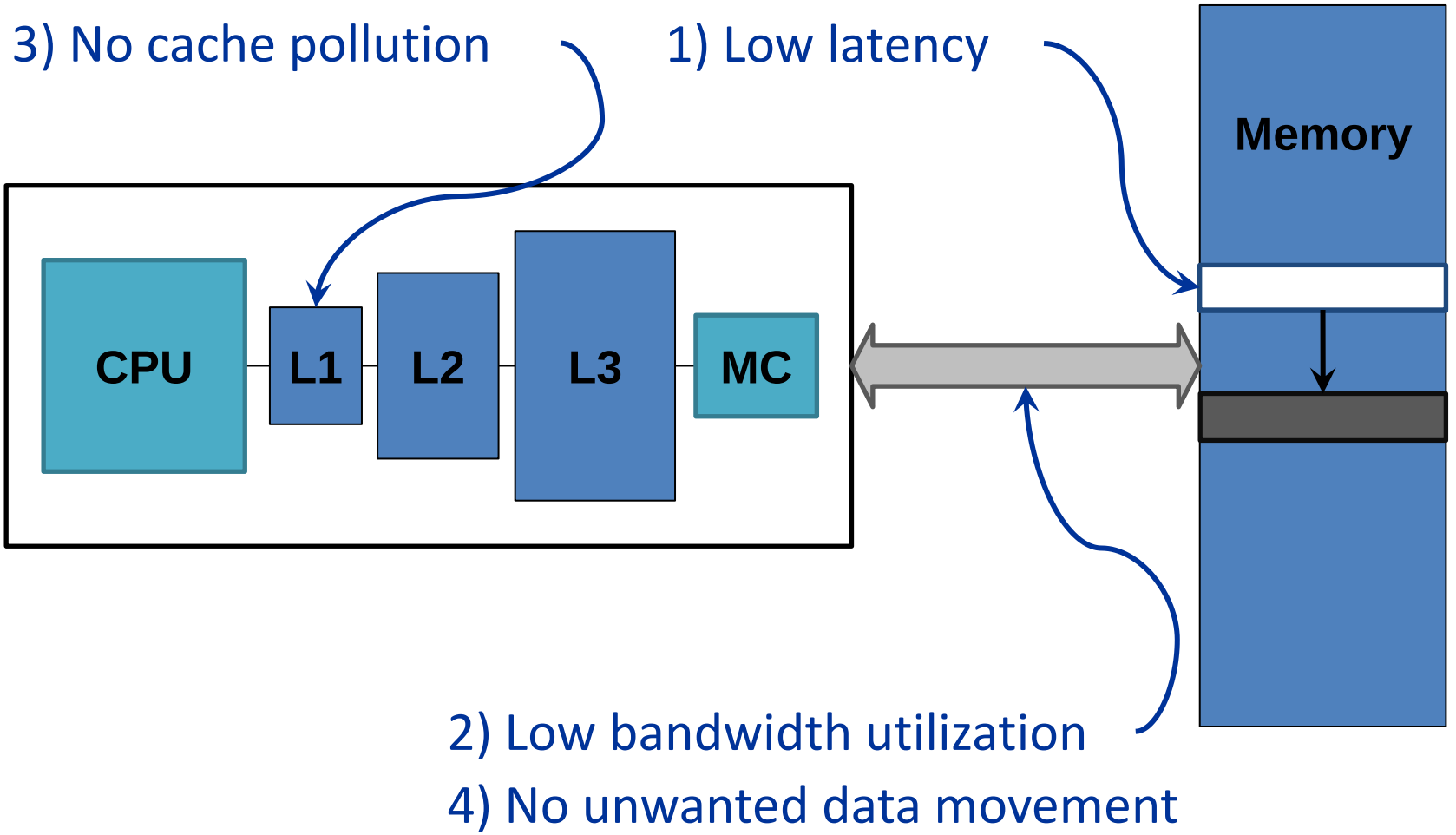
- Examples: RowClone and In-DRAM AND/OR
  - RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data (Seshadri et al., MICRO 2013)
  - Fast Bulk Bitwise AND and OR in DRAM (Seshadri et al., IEEE CAL 2015)
  - …

# Today's Memory: Bulk Data Copy

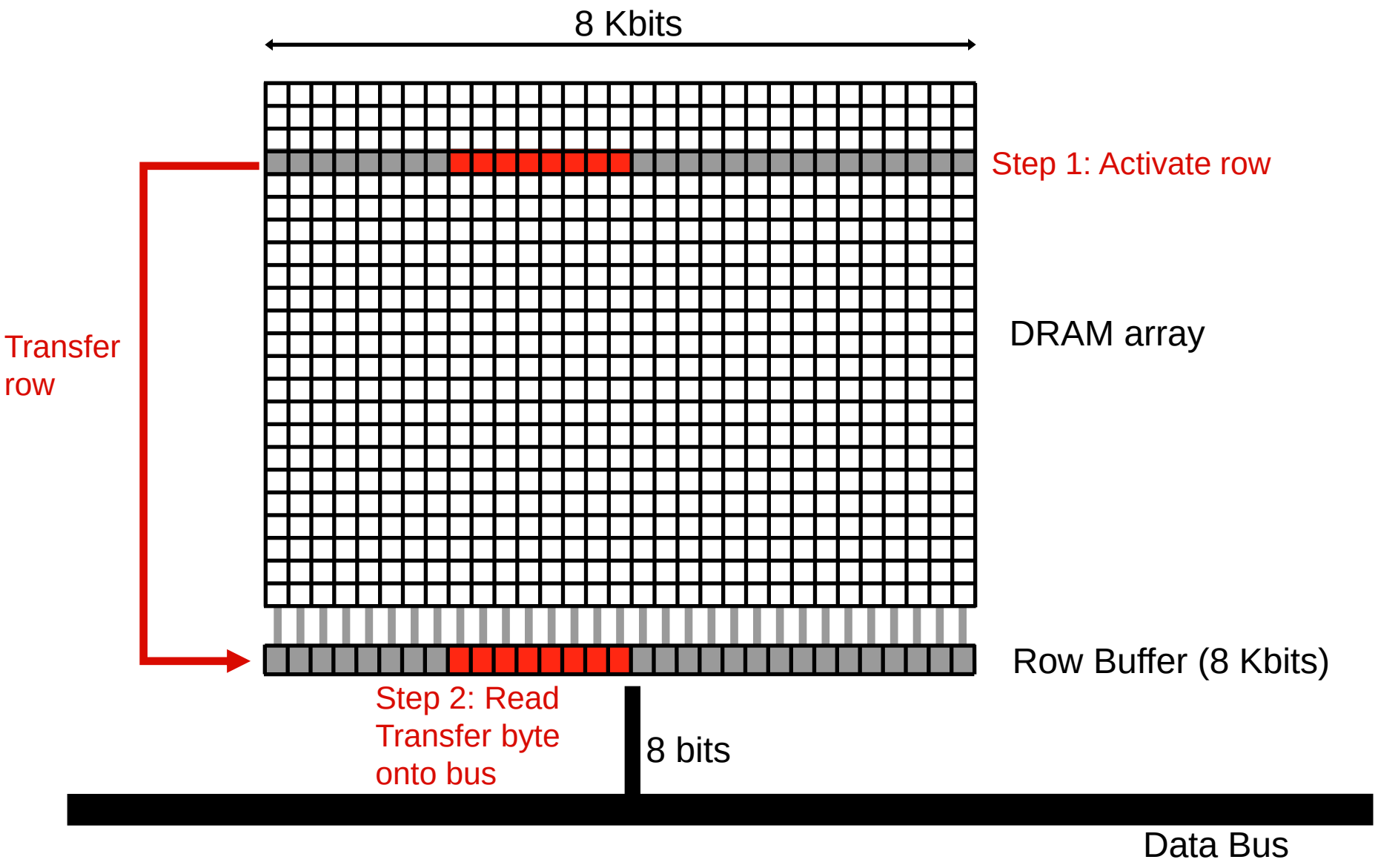



1046ns, 3.6uJ (for 4KB page copy via DMA)

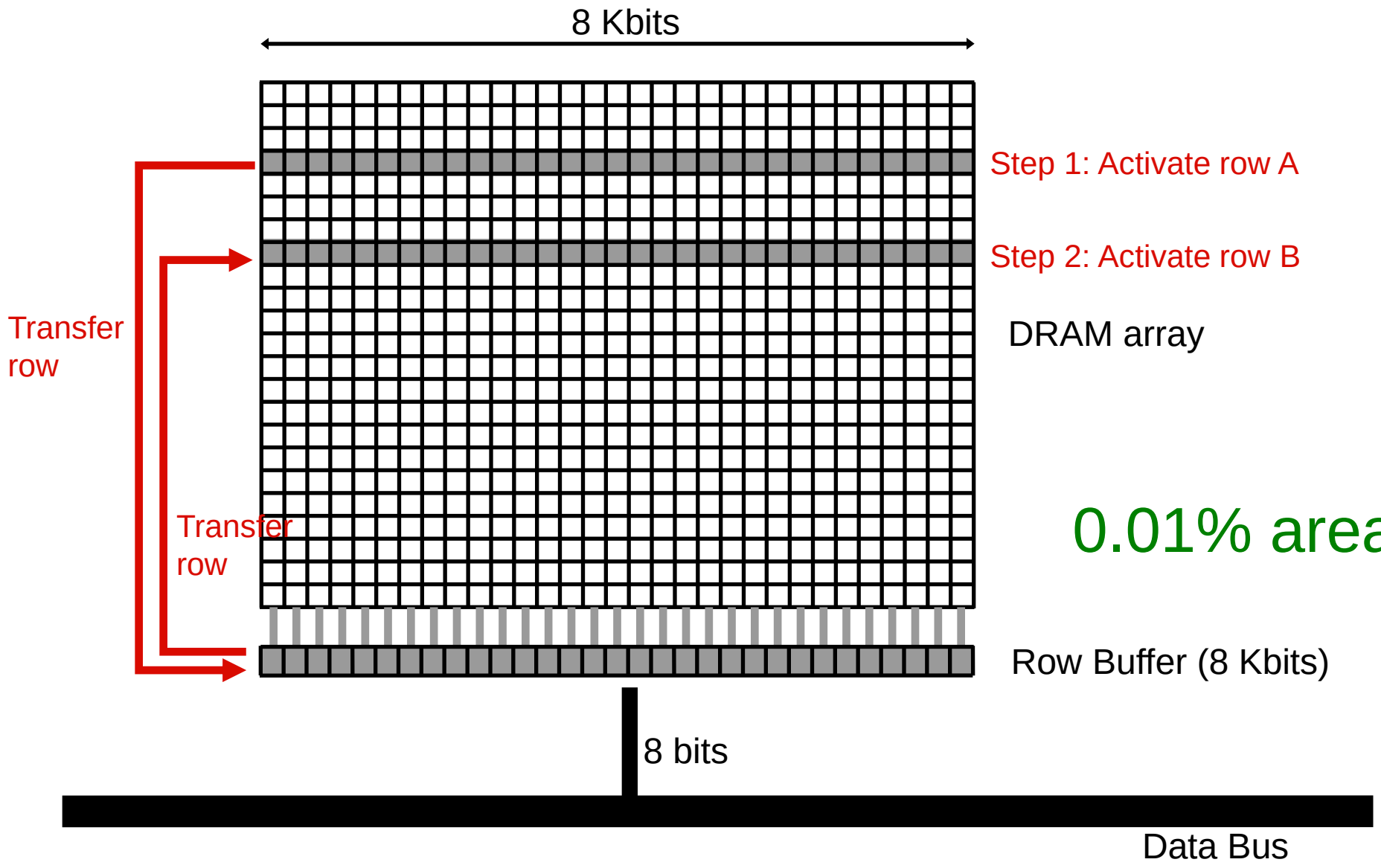
# Future: RowClone (In-Memory Copy)



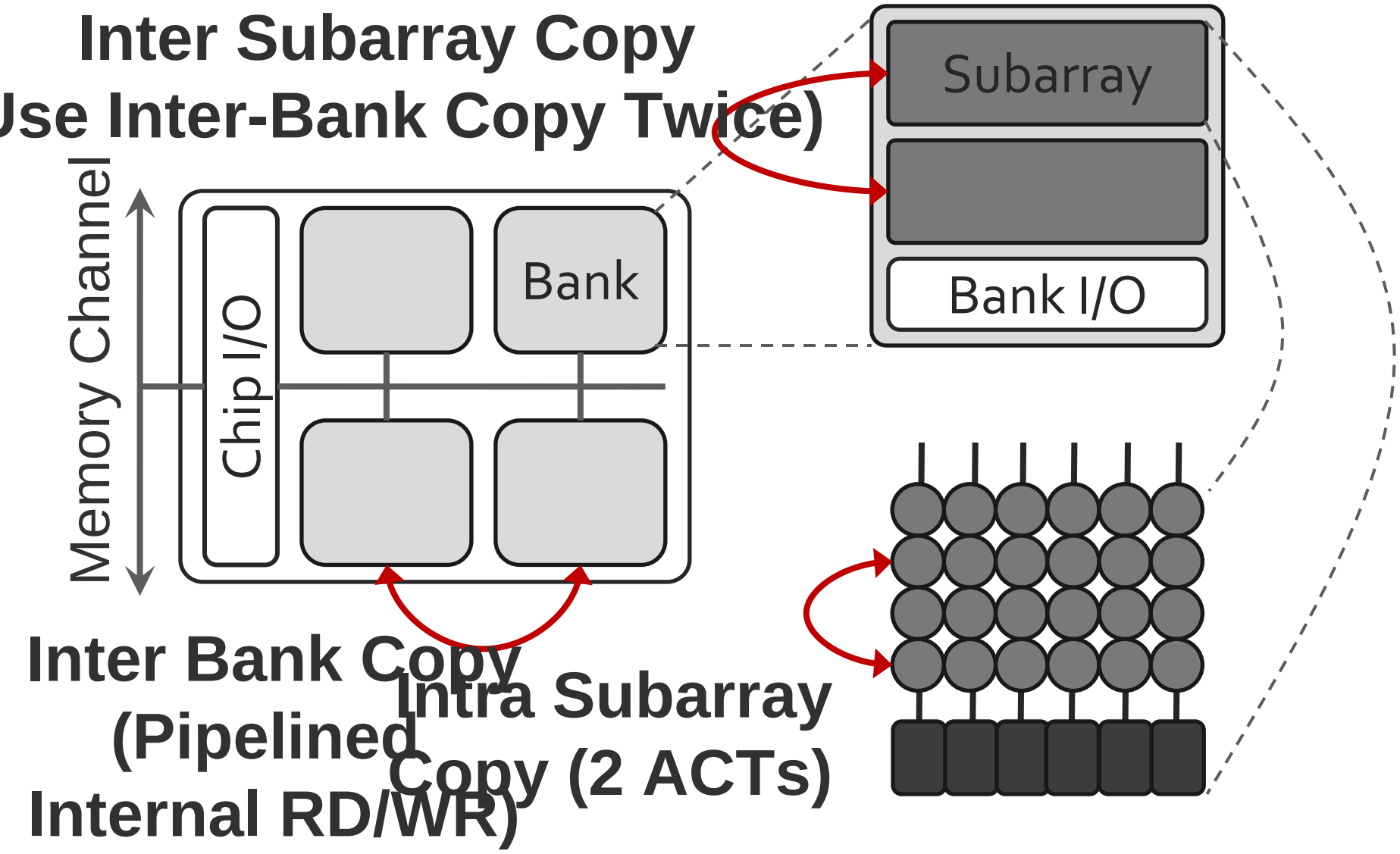
# DRAM Subarray Operation (load one byte)



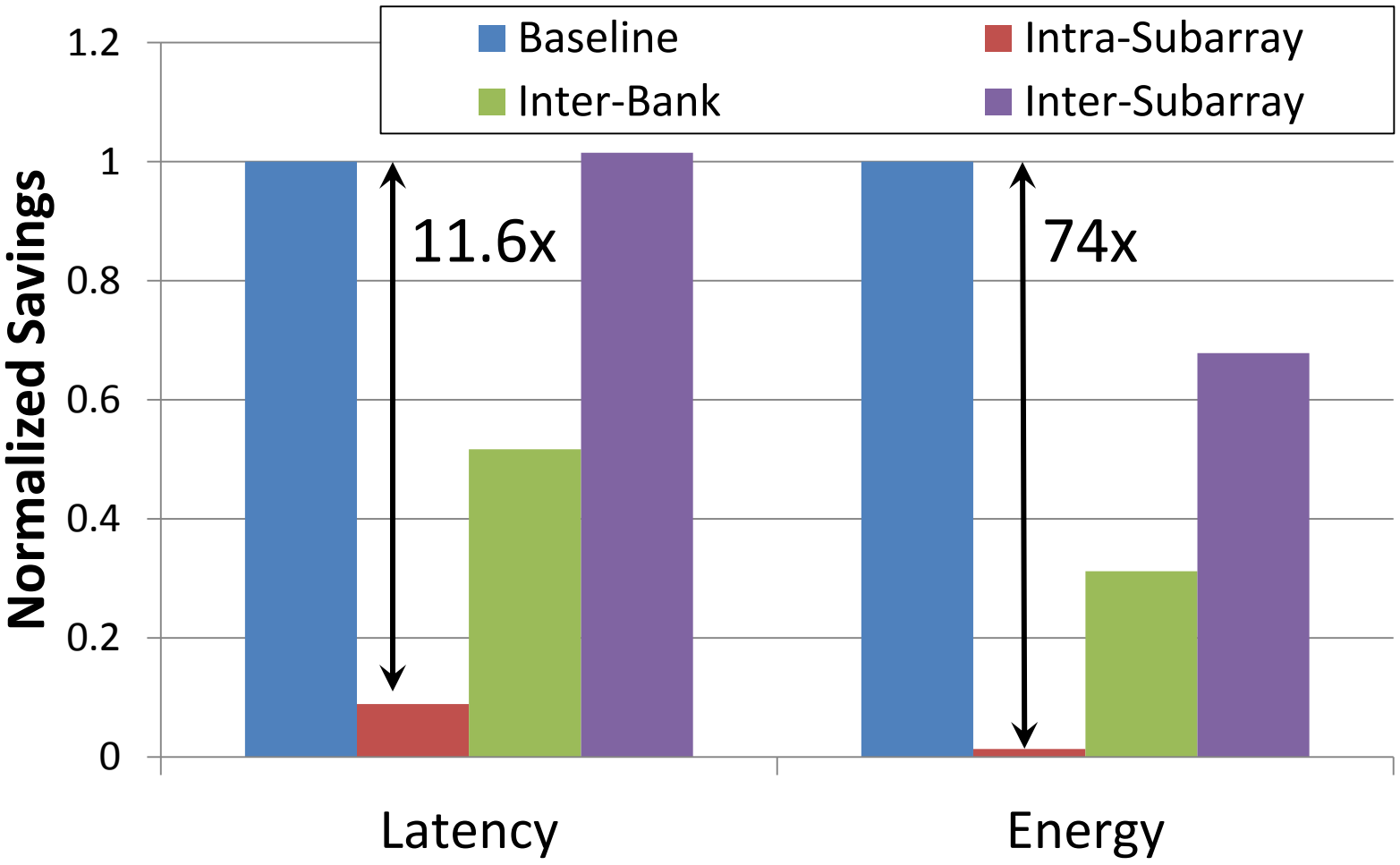
# RowClone: In-DRAM Row Copy



# Generalized RowClone

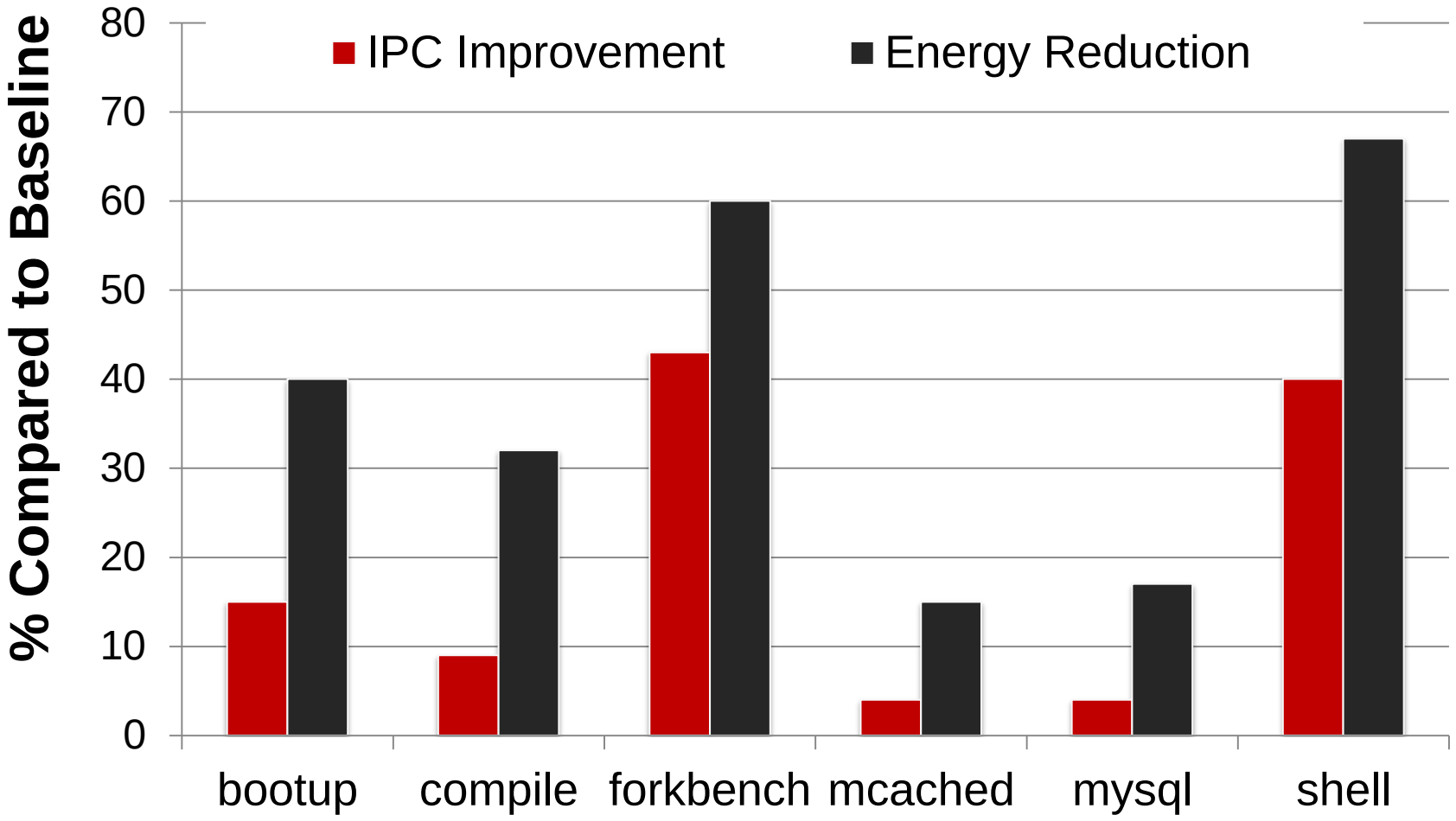


# RowClone: Latency and Energy Savings

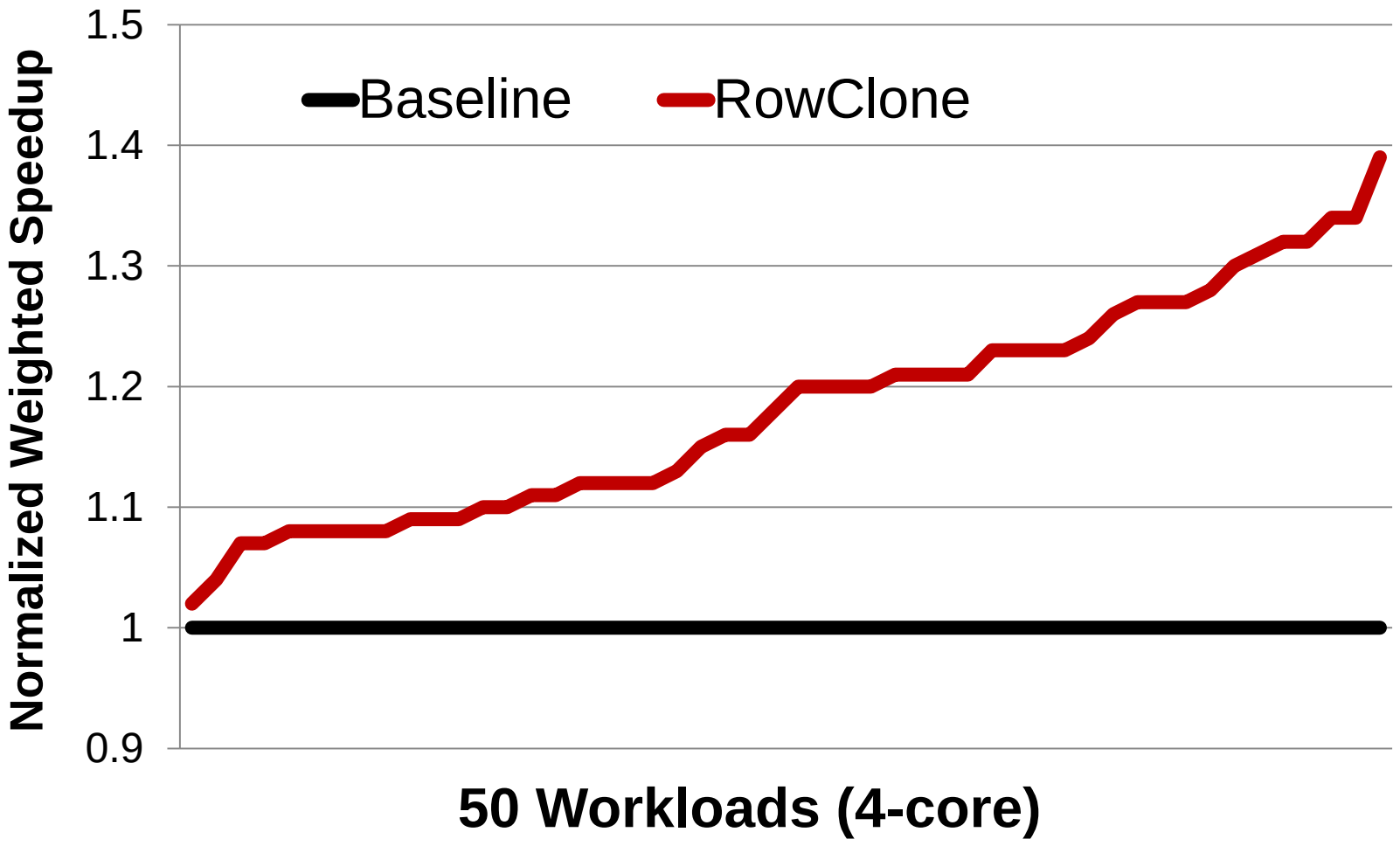



Seshadri et al., "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," MICRO 2013.

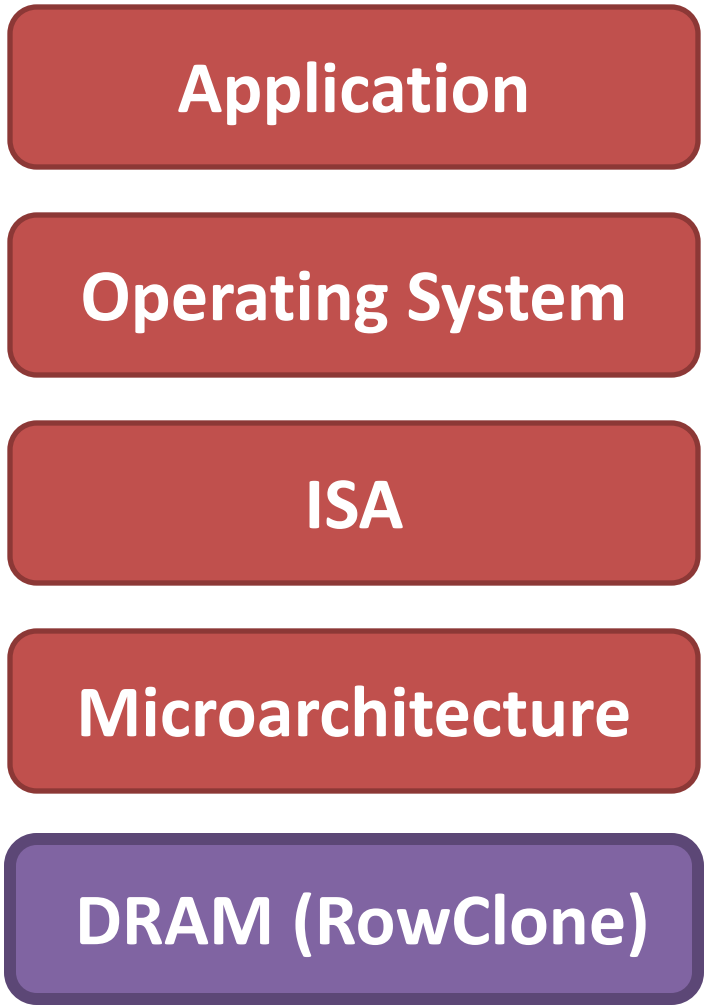
# RowClone: Application Performance



# RowClone: Multi-Core Performance



# End-to-End System Design



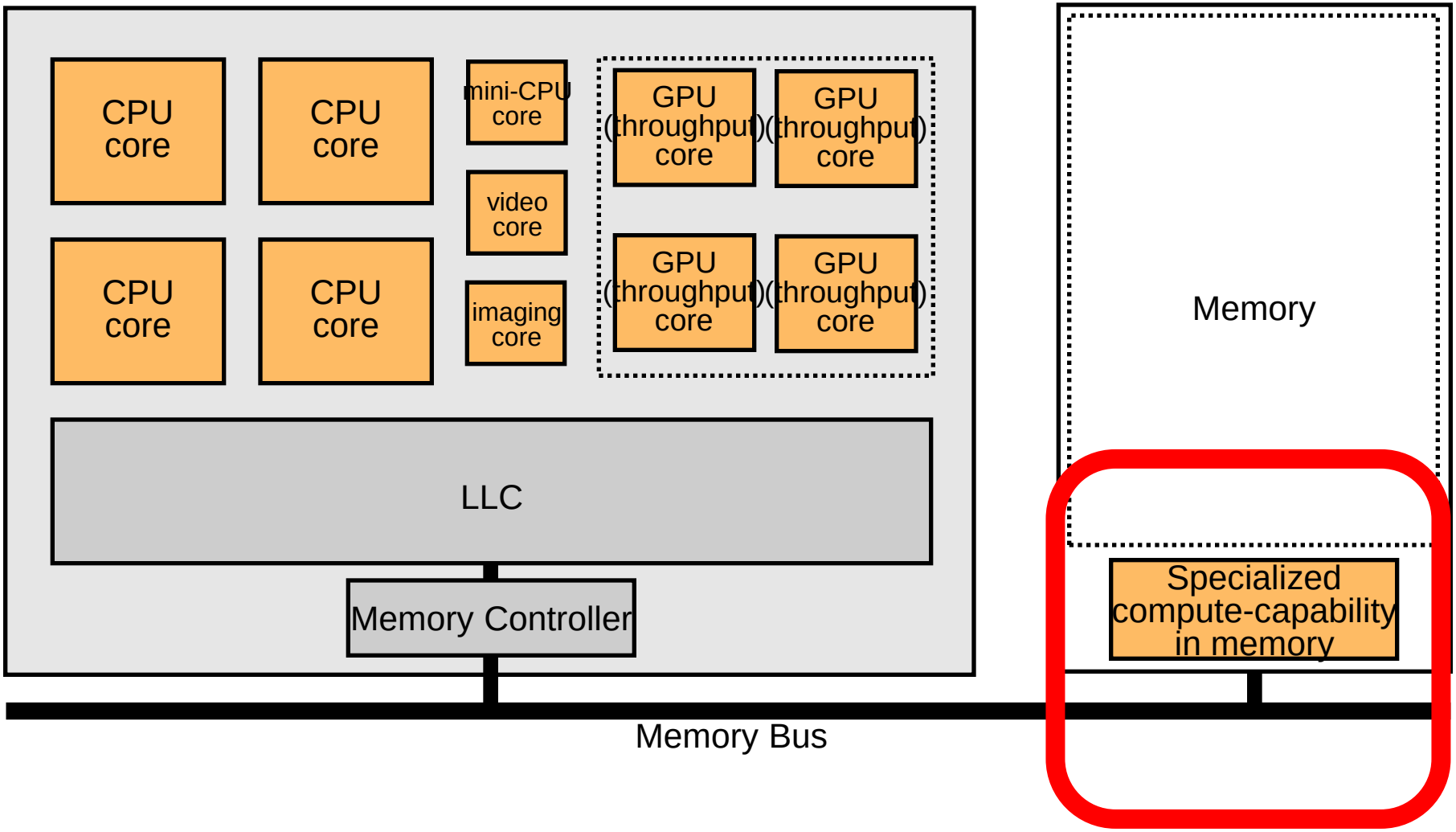

How to communicate occurrences of bulk copy/initialization across layers?

How to ensure cache coherence?

How to maximize latency and energy savings?

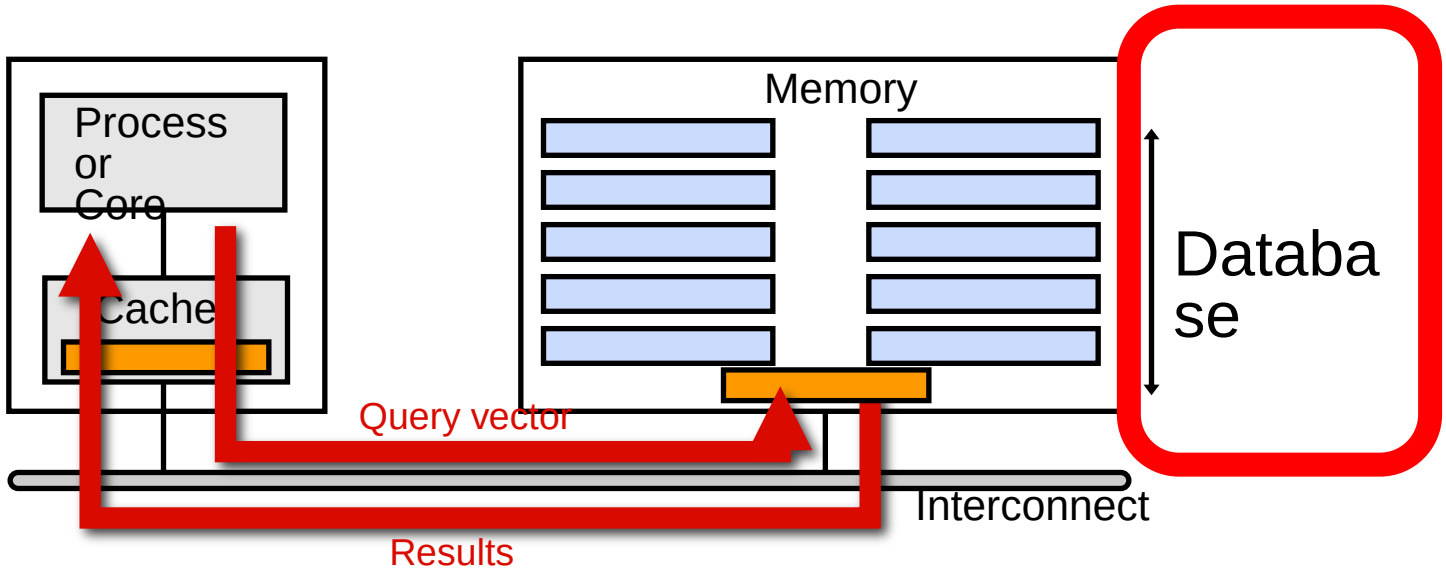
How to handle data reuse?

# Goal: Ultra-Efficient Processing Near Data




**Memory similar to a "conventional" accelera**

# Enabling In-Memory Search

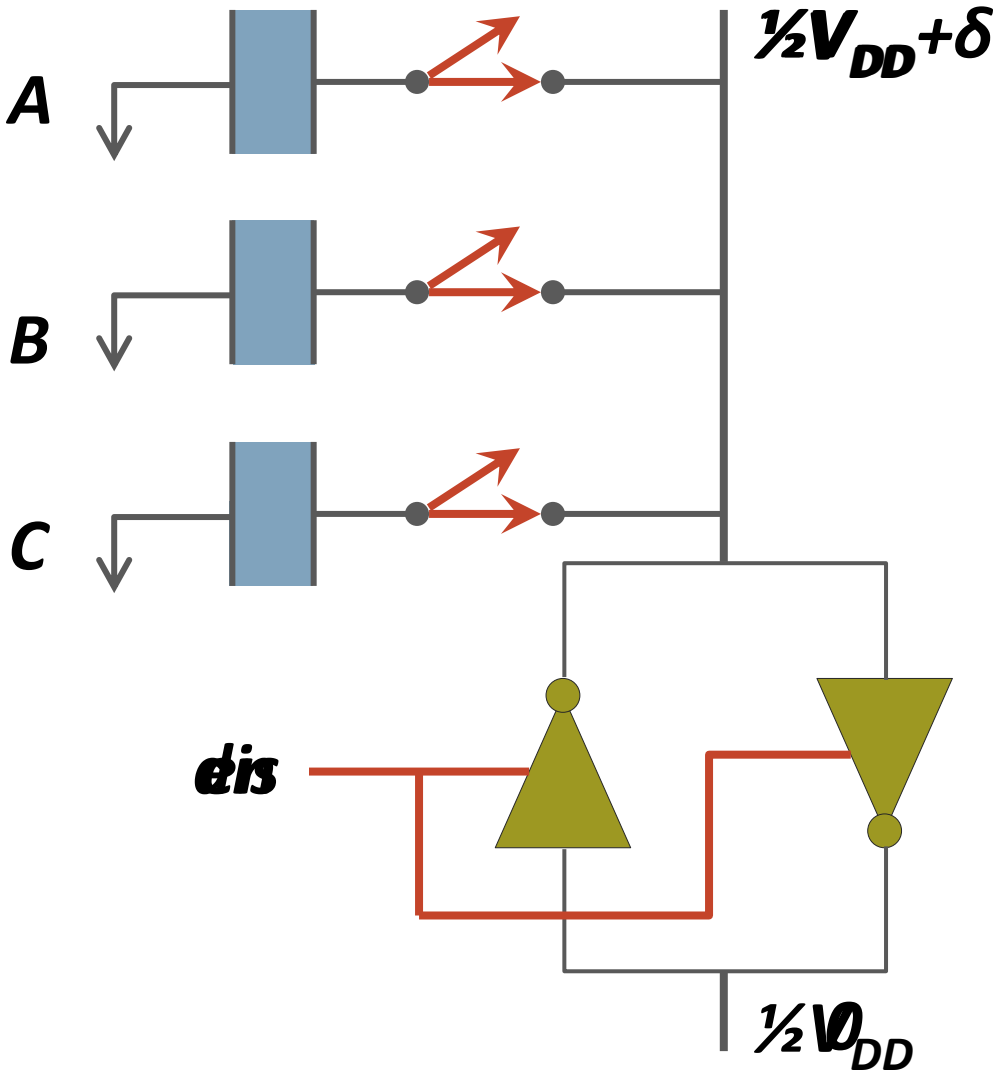



- What is a flexible and scalable memory interface?
- What is the right partitioning of computation capability?
- What is the right low-cost memory substrate?
- What memory technologies are the best

# Enabling In-Memory Computation

| DRAM Support | Cache Coherence | Virtual Memory Support |
| --- | --- | --- |
| RowClone (MICRO 2013) | Dirty-Block Index (ISCA 2014) | Page Overlays (ISCA 2015) |
| In-DRAM Gather Scatter | Non-contiguous Cache lines | Gathered Pages |
| In-DRAM Bitwise Operations (IEEE CAL 2015) | ? | ? |

# In-DRAM AND/OR: Triple Row Activation




**Final State**
***AB + BC + AC***

***C(A + B) + ~C(AB)***



Seshadri+, "Fast Bulk Bitwise AND and OR in DRAM", IEEE CAL 2015.

# In-DRAM Bulk Bitwise AND/OR Operation

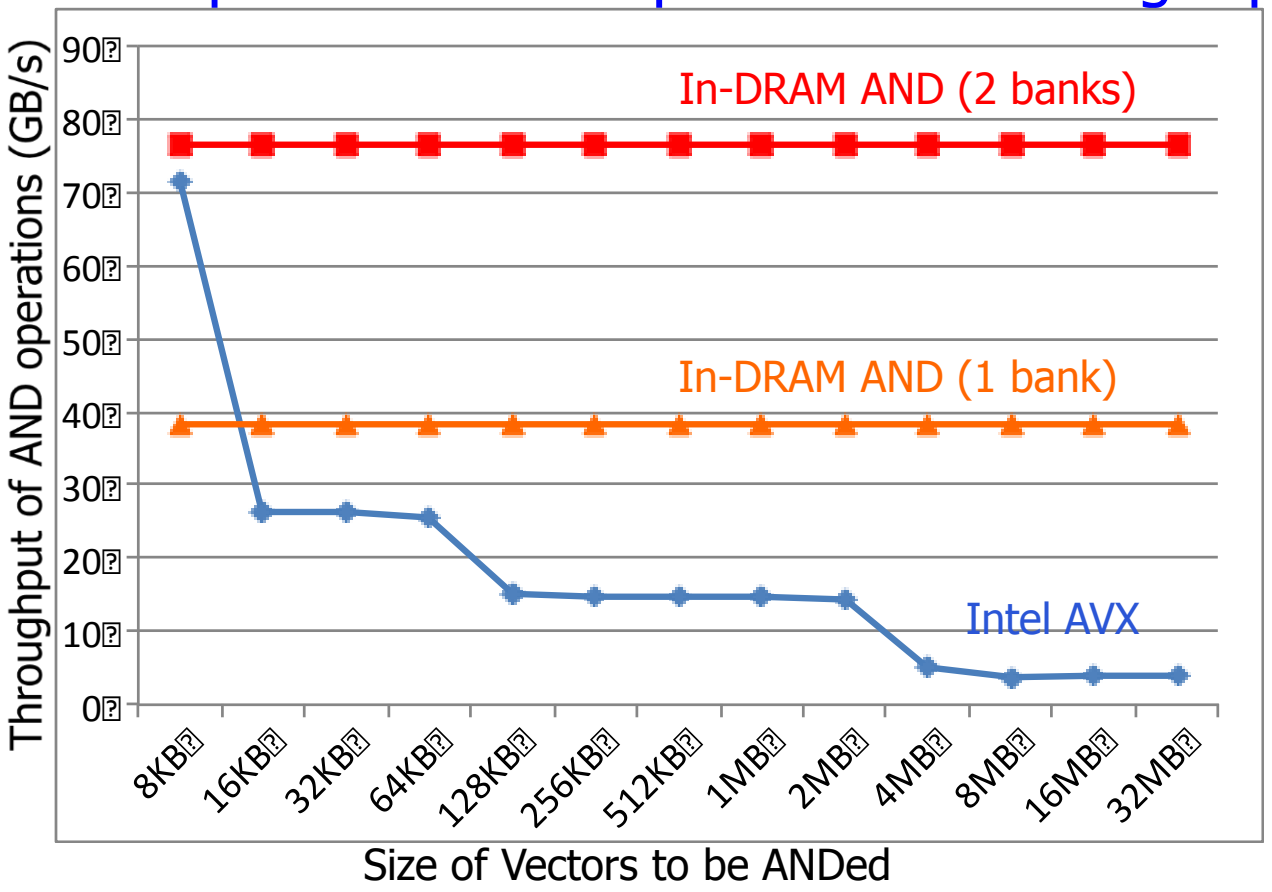
- BULKAND A, B → C
- Semantics: Perform a bitwise AND of two rows A and B and store the result in row C

- R0 – reserved zero row, R1 – reserved one row
- D1, D2, D3 – Designated rows for triple activation

1. RowClone A into D1
2. RowClone B into D2
3. RowClone R0 into D3
4. ACTIVATE D1,D2,D3
5. RowClone Result into C

# In-DRAM AND/OR Results

- 20X improvement in AND/OR throughput vs. Intel AVX
- 50.5X reduction in memory energy consumption
- At least 30% performance improvement in range queries

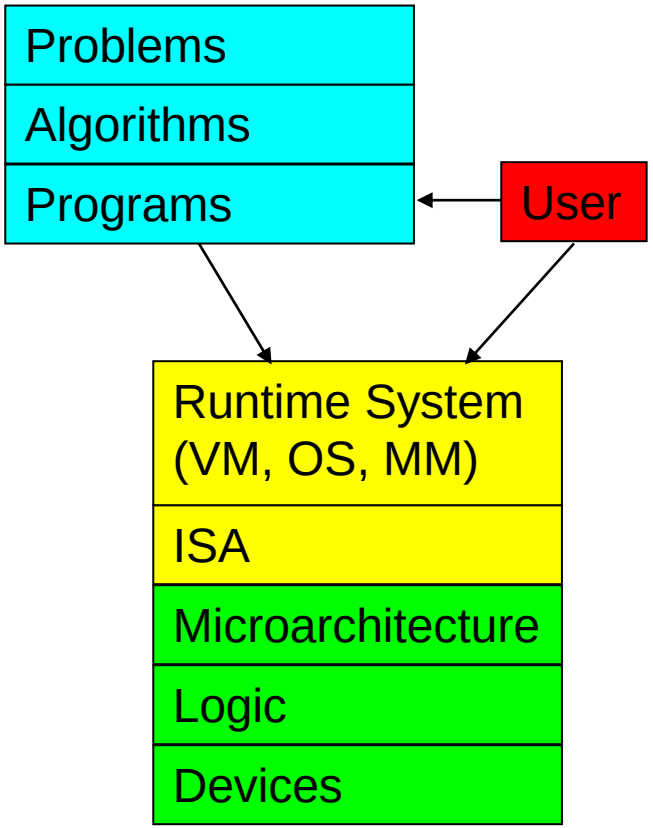





Seshadri+, "Fast Bulk Bitwise AND and OR in DRAM", IEEE CAL 2015.

# Going Forward

- A bulk computation model in memory
- New memory & software interfaces to enable bulk in-memory computation
- New programming models, algorithms, compilers, and system designs that can take advantage of the model



# Two Approaches to In-Memory Processing

- 1. Minimally change DRAM to enable simple yet powerful computation primitives
  - RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data (Seshadri et al., MICRO 2013)
  - Fast Bulk Bitwise AND and OR in DRAM (Seshadri et al., IEEE CAL 2015)

- 2. Exploit the control logic in 3D-stacked memory to enable more comprehensive computation near memory
  - PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture (Ahn et al., ISCA 2015)
  - A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing (Ahn et al., ISCA 2015)

# Two Key Questions in 3D Stacked PIM

- What is the minimal processing-in-memory support we can provide ?
  - without changing the system significantly
  - while achieving significant benefits of processing in 3D-stacked memory

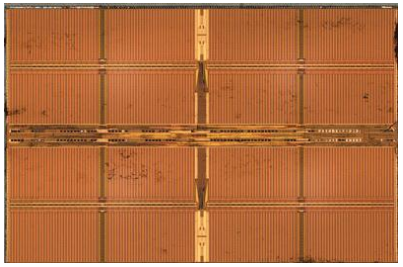
- How can we accelerate important applications if we use 3D-stacked memory as a coarse-grained accelerator?
  - what is the architecture and programming model?
  - what are the mechanisms for acceleration?

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

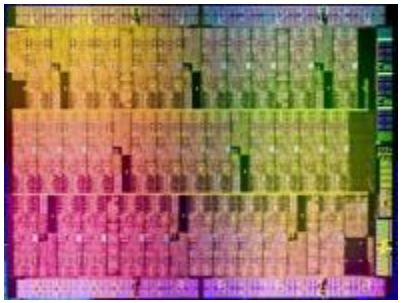
PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture
(Ahn et al., ISCA 2015)

# Challenges in Processing-in-Memory

## Cost-effectiveness



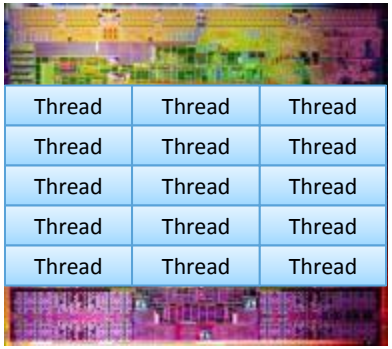
DRAM die

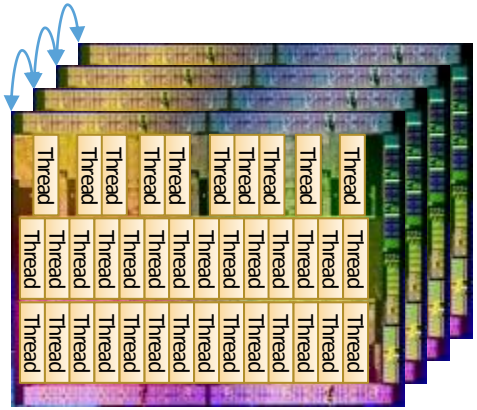


Complex Logic

## Programming Model

Host Processor

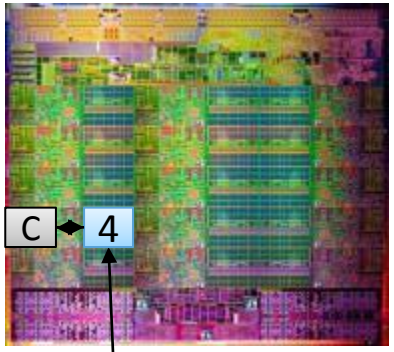


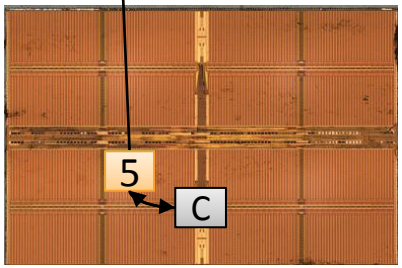




In-Memory Processors

## Coherence & VM

Host Processor

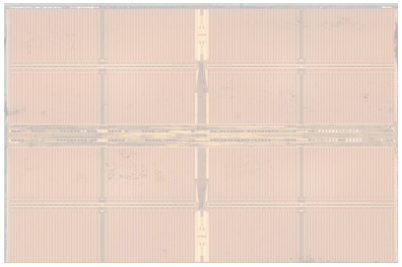


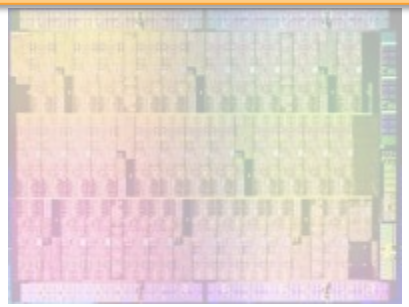

DRAM die

# Challenges in Processing-in-Memory

## Cost-effectiveness



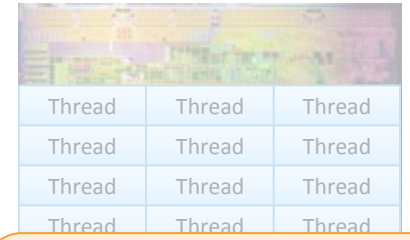
(Partially) Solved by 3D-Stacked DRAM



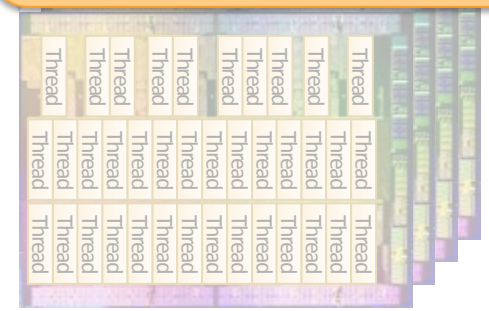
Complex Logic

## Programming Model

Host Processor



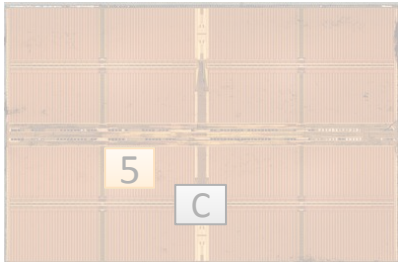

**Still Challenging** even in Recent PIM Architectures (e.g., AC-DIMM, NDA, NDC, TOP-PIM, Tesseract, …)




In-Memory Processors

## Coherence & VM

Host Processor




DRAM die

# Simple PIM Programming, Coherence, VM

- Objectives
  - Provide an intuitive programming model for PIM
  - Full support for cache coherence and virtual memory
  - Minimize implementation overhead of PIM units

- Solution: simple PIM operations as ISA extensions
  - PIM operations as host processor instructions: intuitive
    - Preserves sequential programming model
    - Avoids the need for virtual memory support in memory
  - Leads to low-overhead implementation
  - PIM-enabled instructions can be executed on the host-side or the memory side (locality-aware execution)

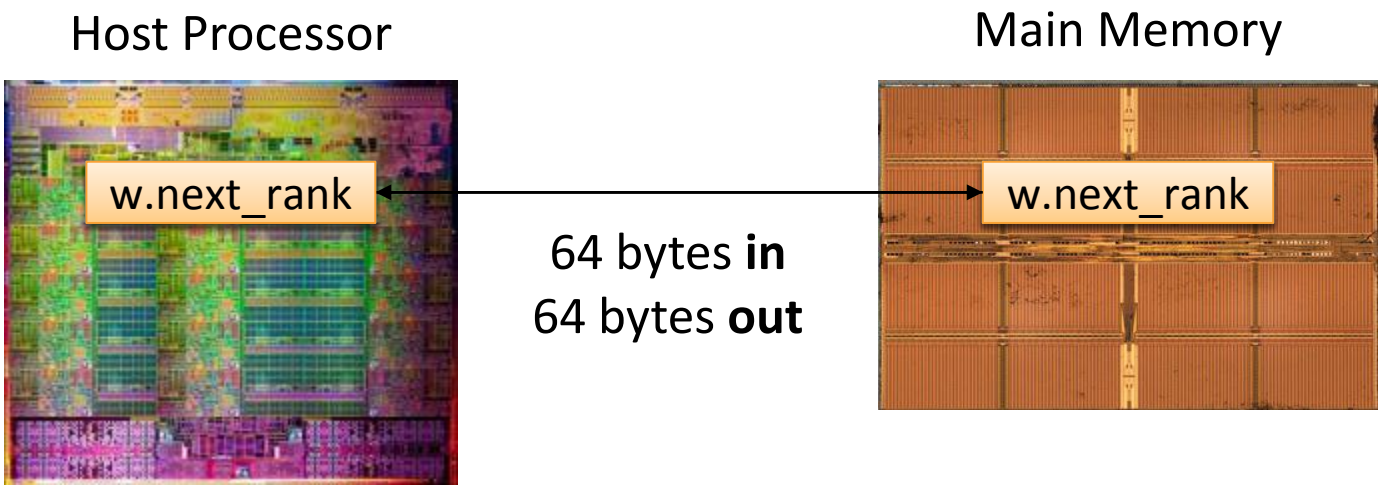
# Simple PIM Operations as ISA Extensions (I)

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;
}
```

# Simple PIM Operations as ISA Extensions (II)

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

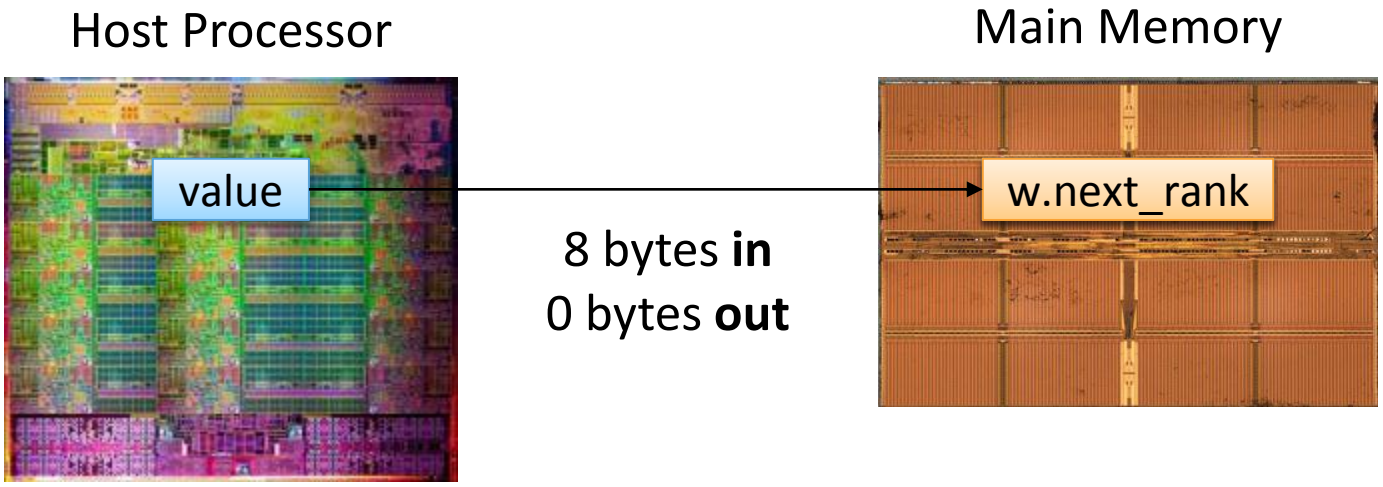



**Conventional Architecture**

# Simple PIM Operations as ISA Extensions (III)

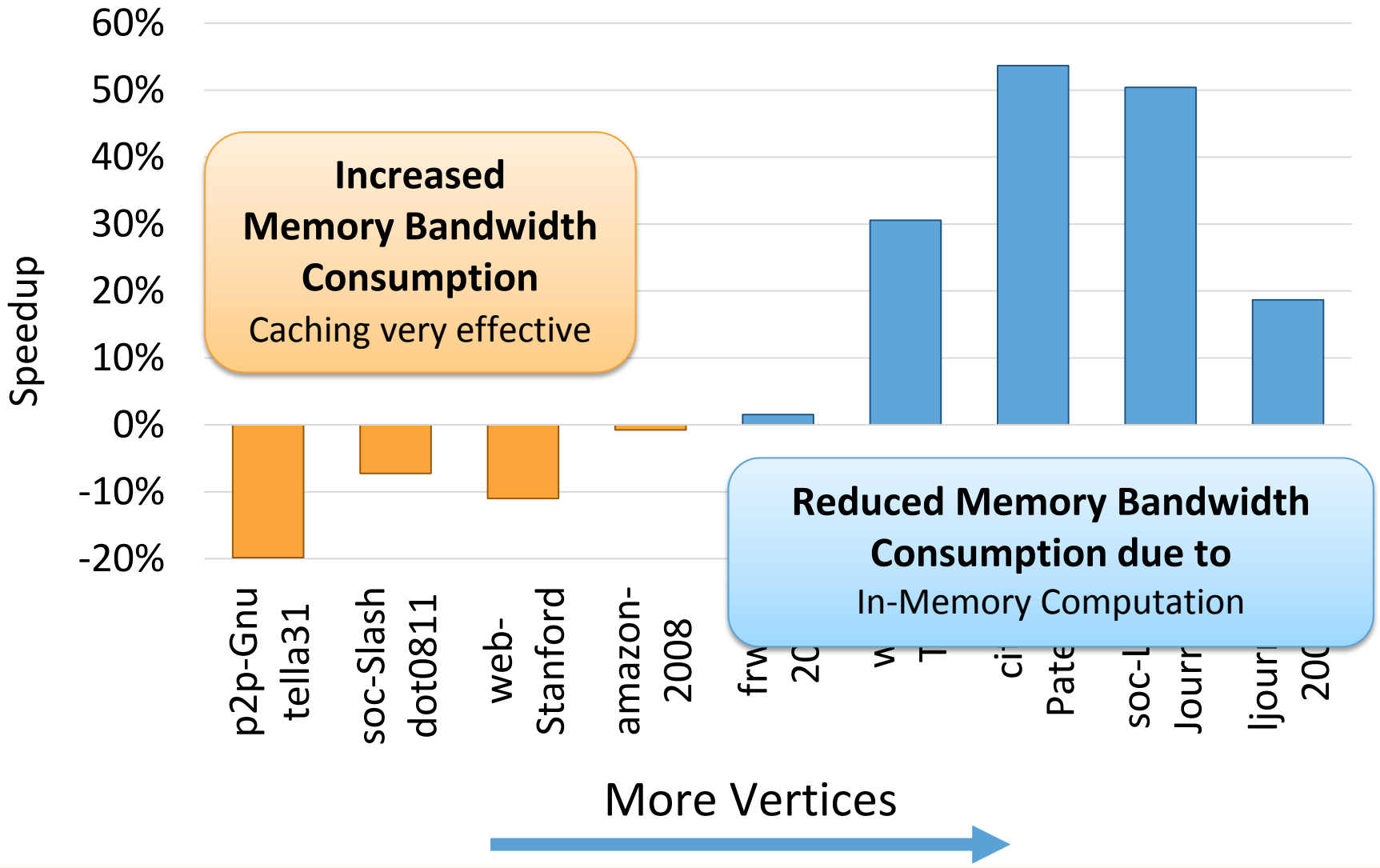
```
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        __pim_add(&w.next_rank, value);
    }
}
```

pim.add r1, (r2)




**In-Memory Addition**

# Always Executing in Memory? Not A Good Idea



# Two Key Questions for Simple PIM

- How should simple PIM operations be interfaced to conventional systems?
  - PIM-enabled Instructions (PEIs): Expose PIM operations as *cache-coherent, virtually-addressed host processor instructions*
  - No changes to the existing sequential programming model
- What is the most efficient way of exploiting such simple PIM operations?
  - Locality-aware PEIs: Dynamically determine the location of PEI 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

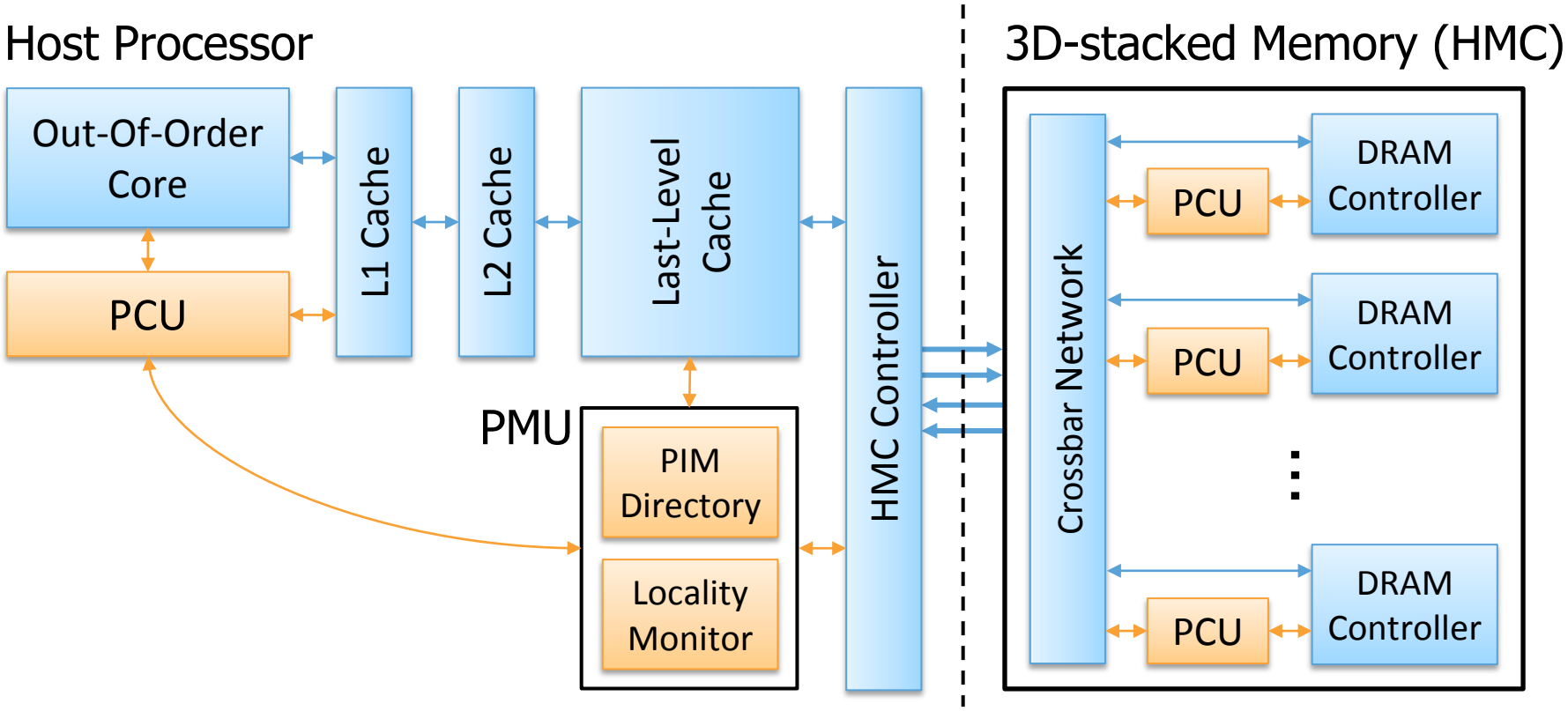
- Executed either in memory or in the host processor
- Cache-coherent, virtually-addressed
- Atomic between different PEIs
- *Not* atomic with normal instructions (use *pfence*)

# 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

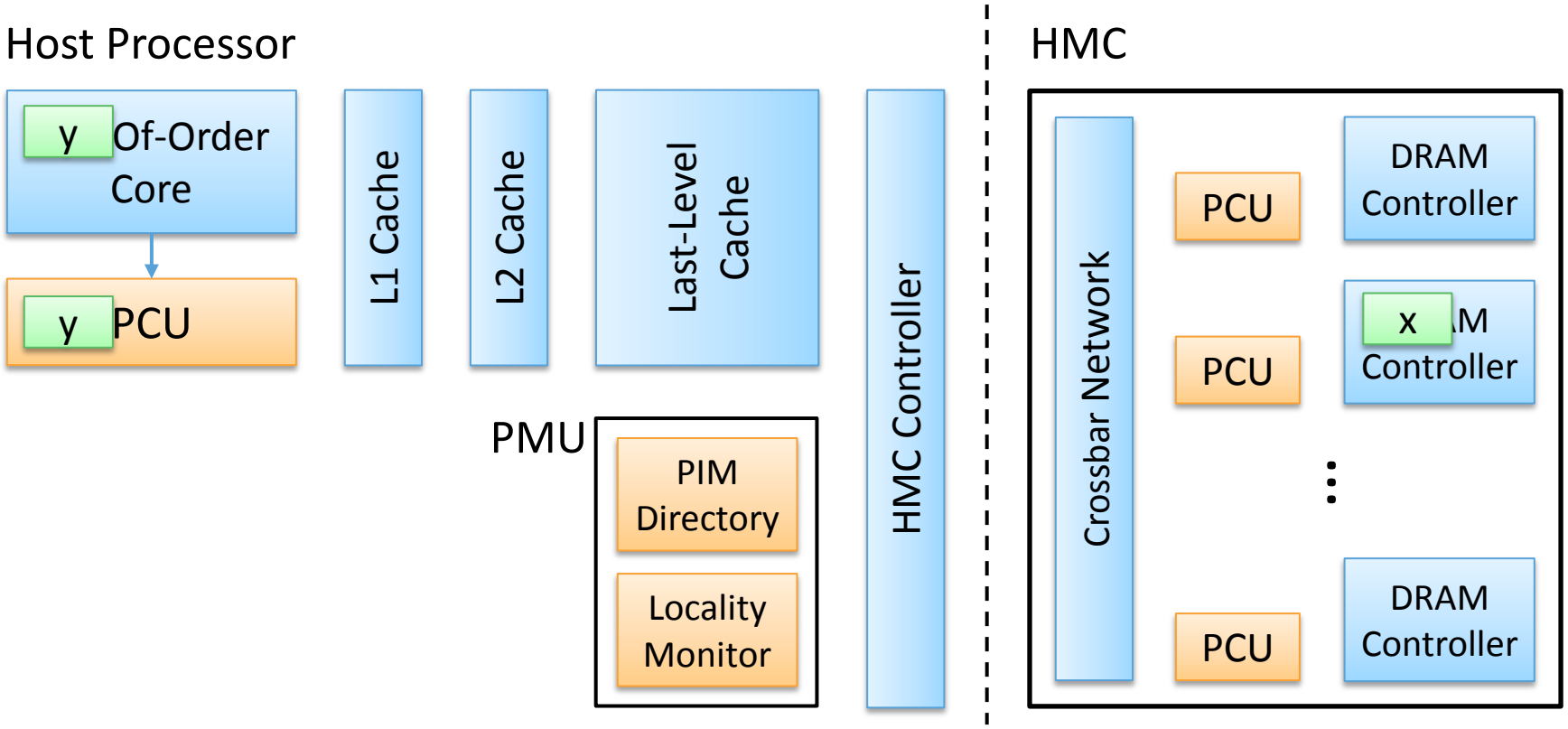
- Benefits
  - **Localization**: each PEI is bounded to one memory module
  - **Interoperability**: easier support for cache coherence and virtual memory
  - **Simplified locality monitoring**: data locality of PEIs can be identified simply by the cache control logic

# PEI Architecture



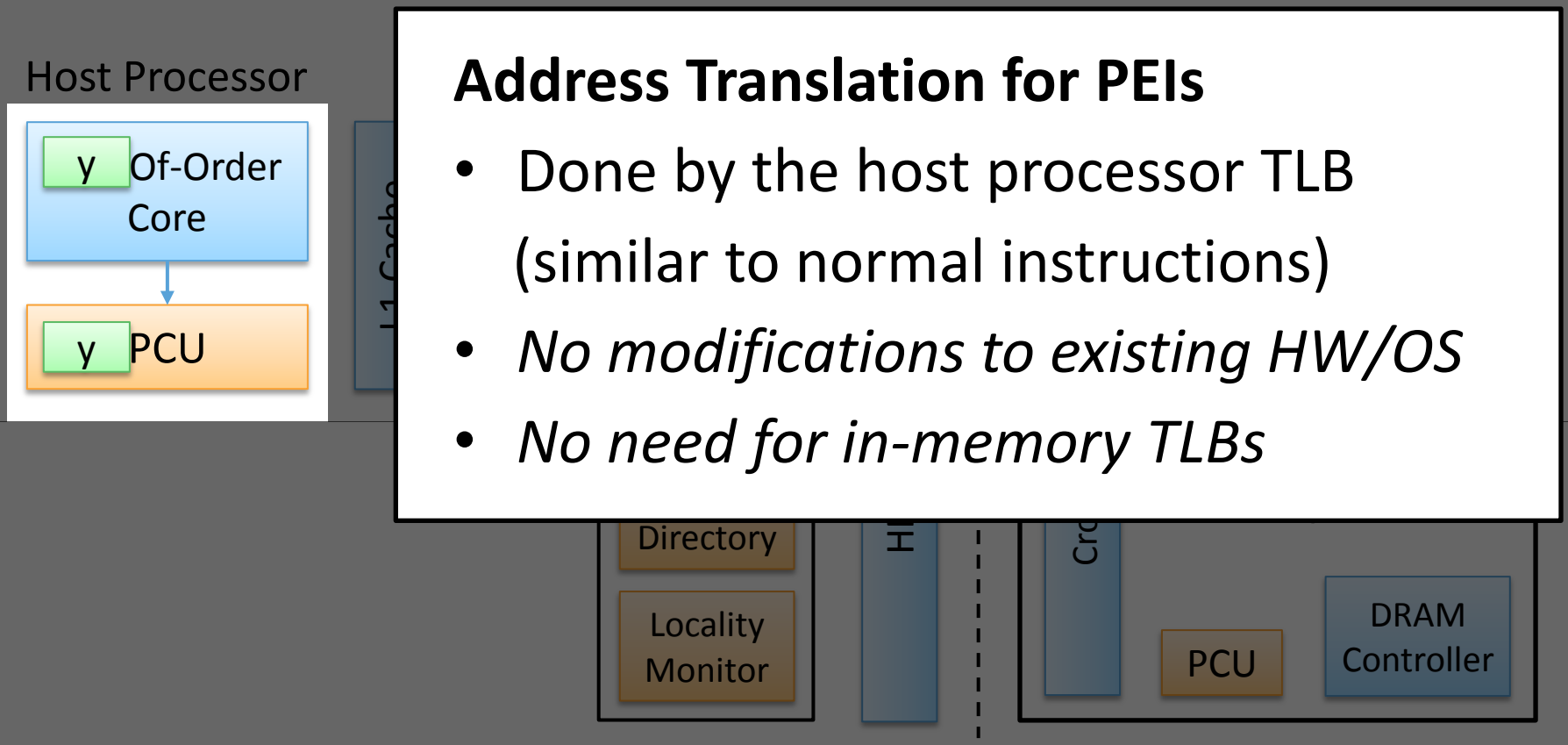

Proposed PEI Architecture

# Memory-side PEI Execution




pim.add y, &x

# Memory-side PEI Execution

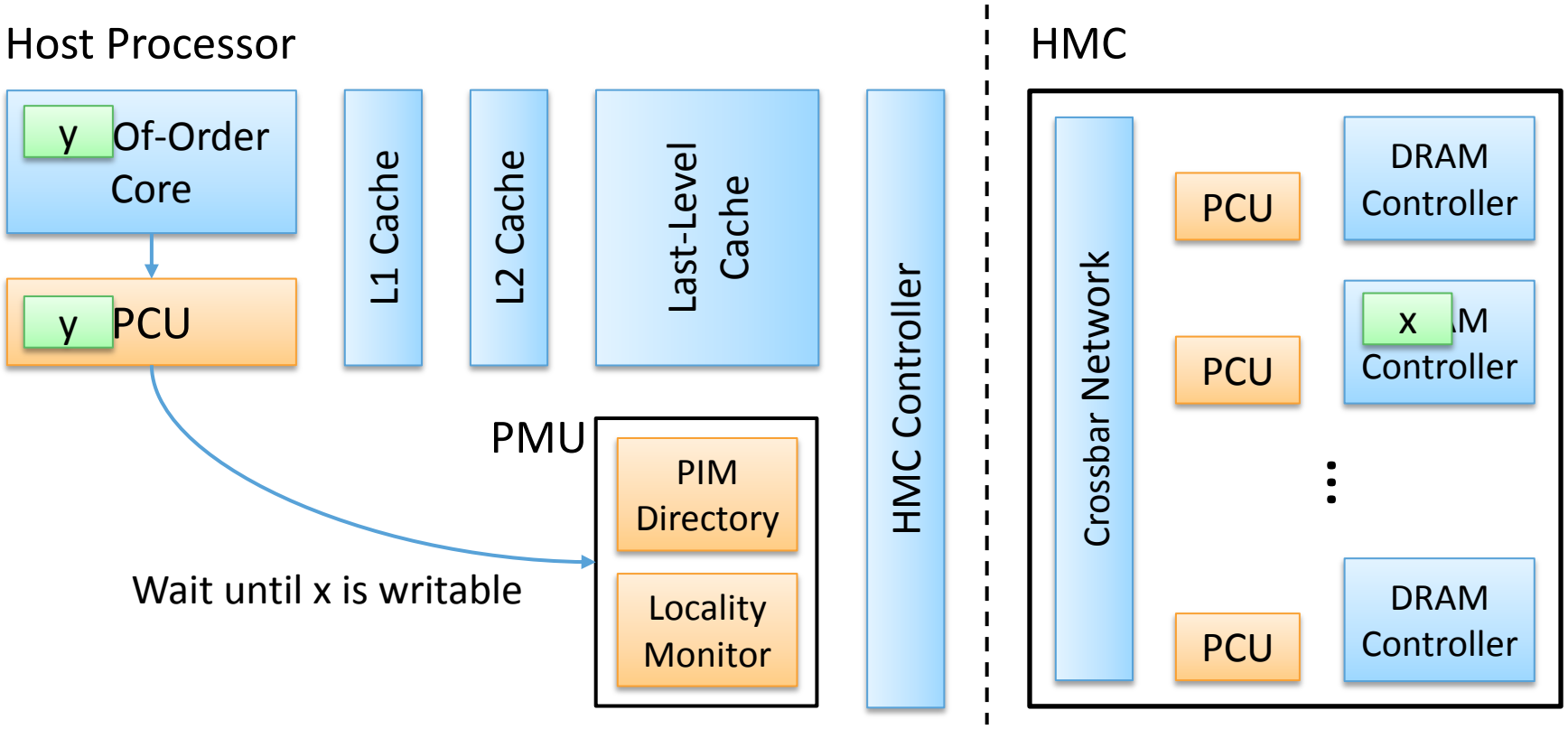



**Address Translation for PEIs**

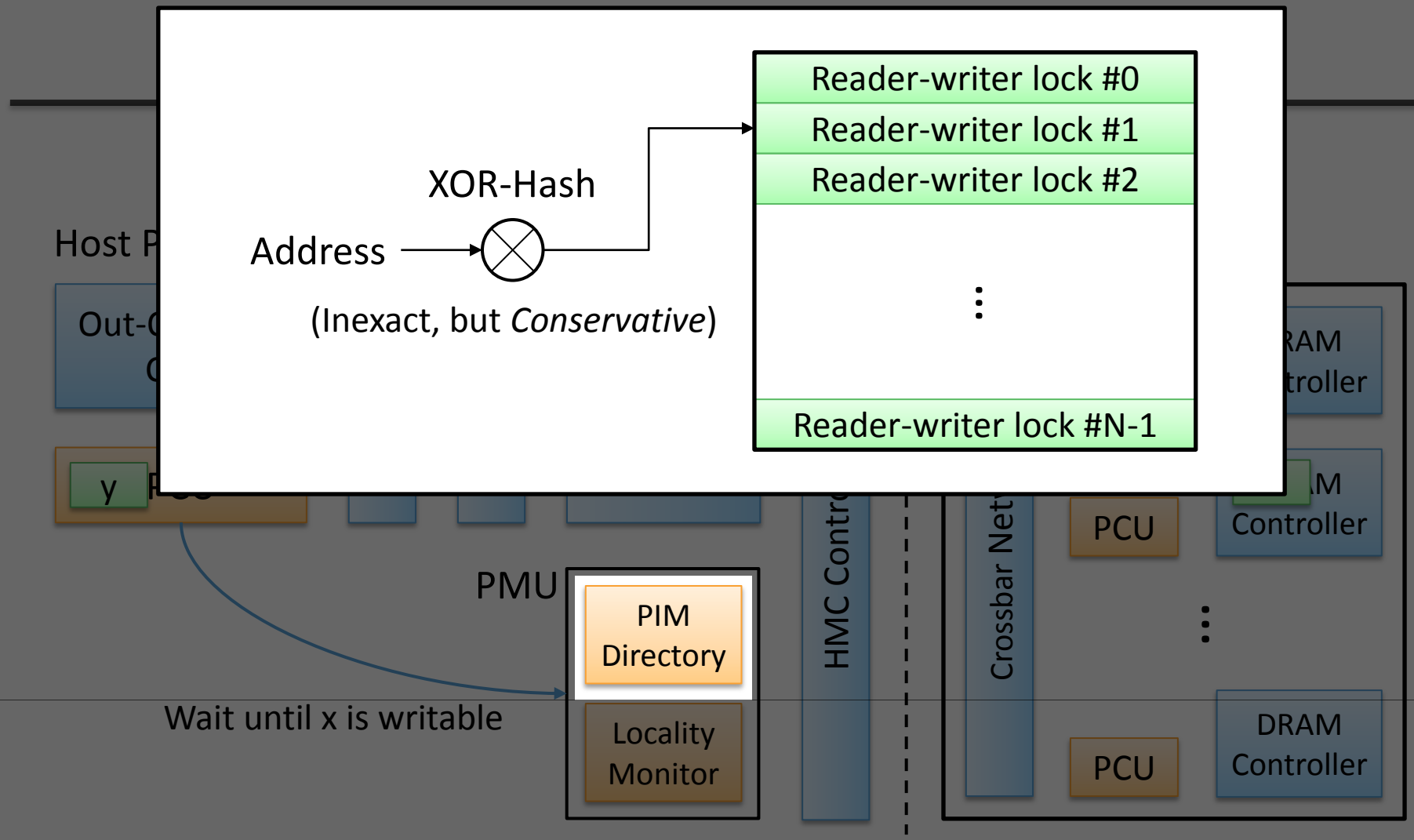
- Done by the host processor TLB (similar to normal instructions)
- *No modifications to existing HW/OS*
- *No need for in-memory TLBs*

pim.add y, &x

# Memory-side PEI Execution

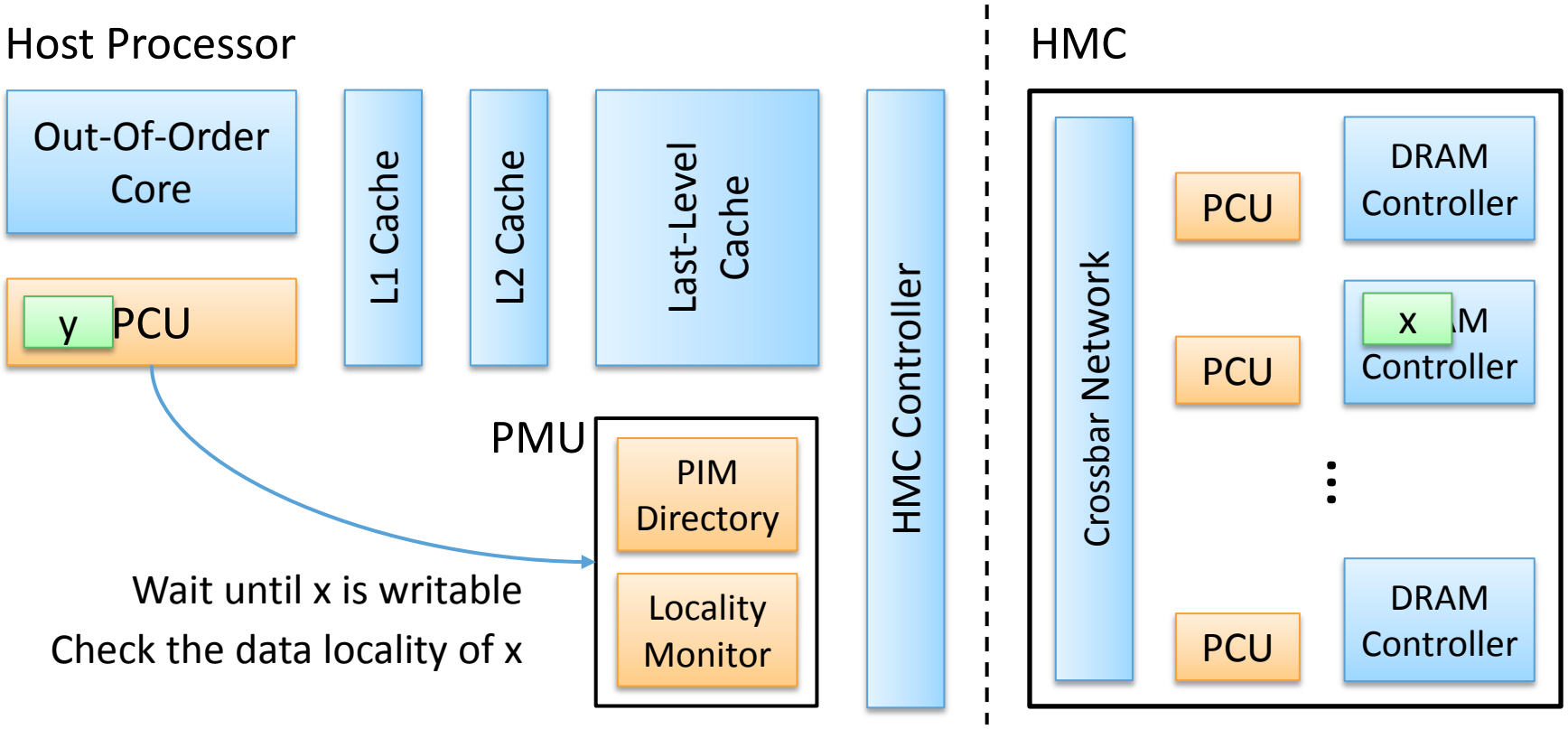



pim.add y, &x

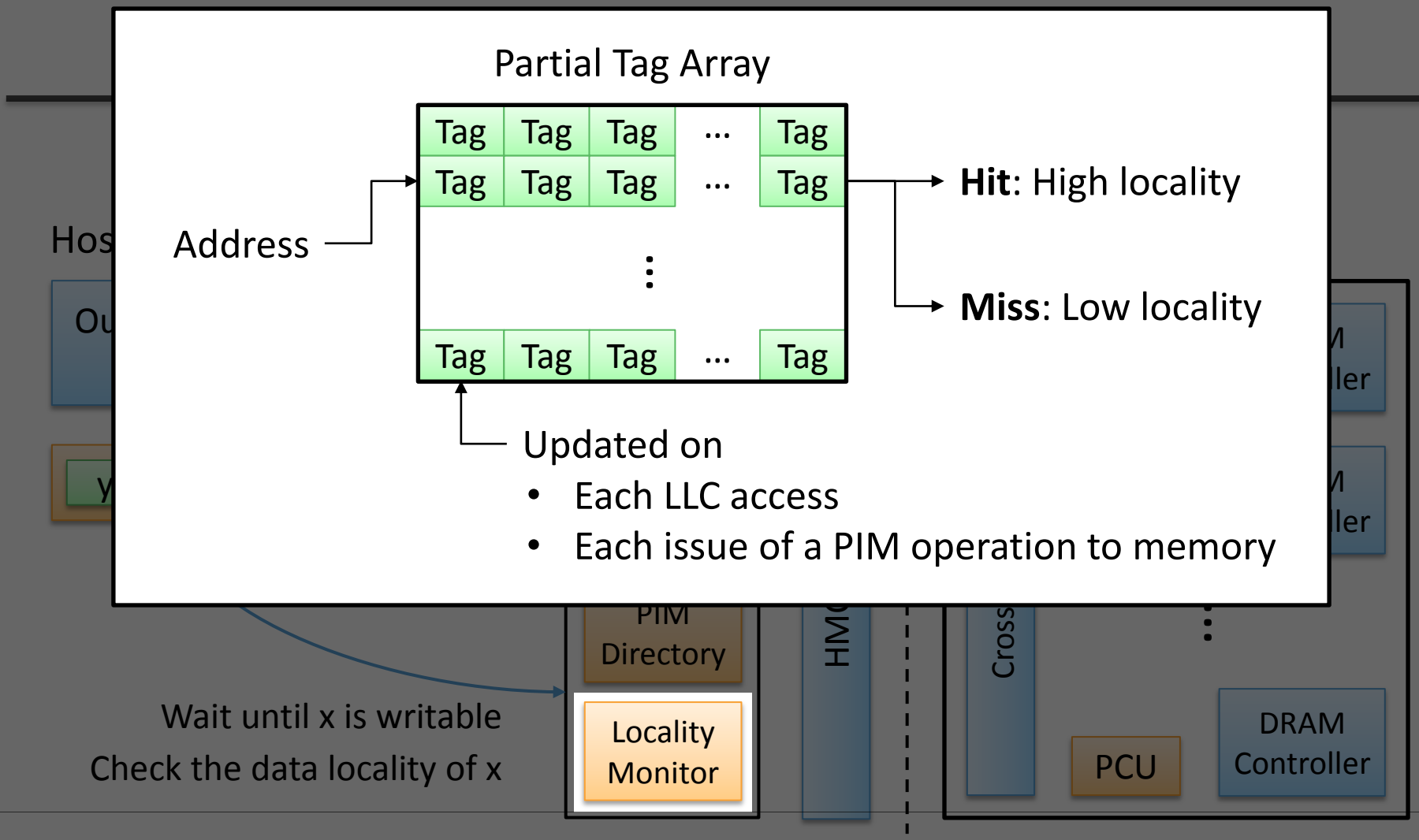

Reader-writer lock #0
Reader-writer lock #1
Reader-writer lock #2
Reader-writer lock #N-1
XOR-Hash
Address
(Inexact, but *Conservative*)
Host P
Out-C
y
PMU
PIM Directory
Locality Monitor
Wait until x is writable
HMC Contro
Crossbar Net
PCU
PCU
DRAM Controller
Controller
RAM troller

pim.add y, &x

# Memory-side PEI Execution

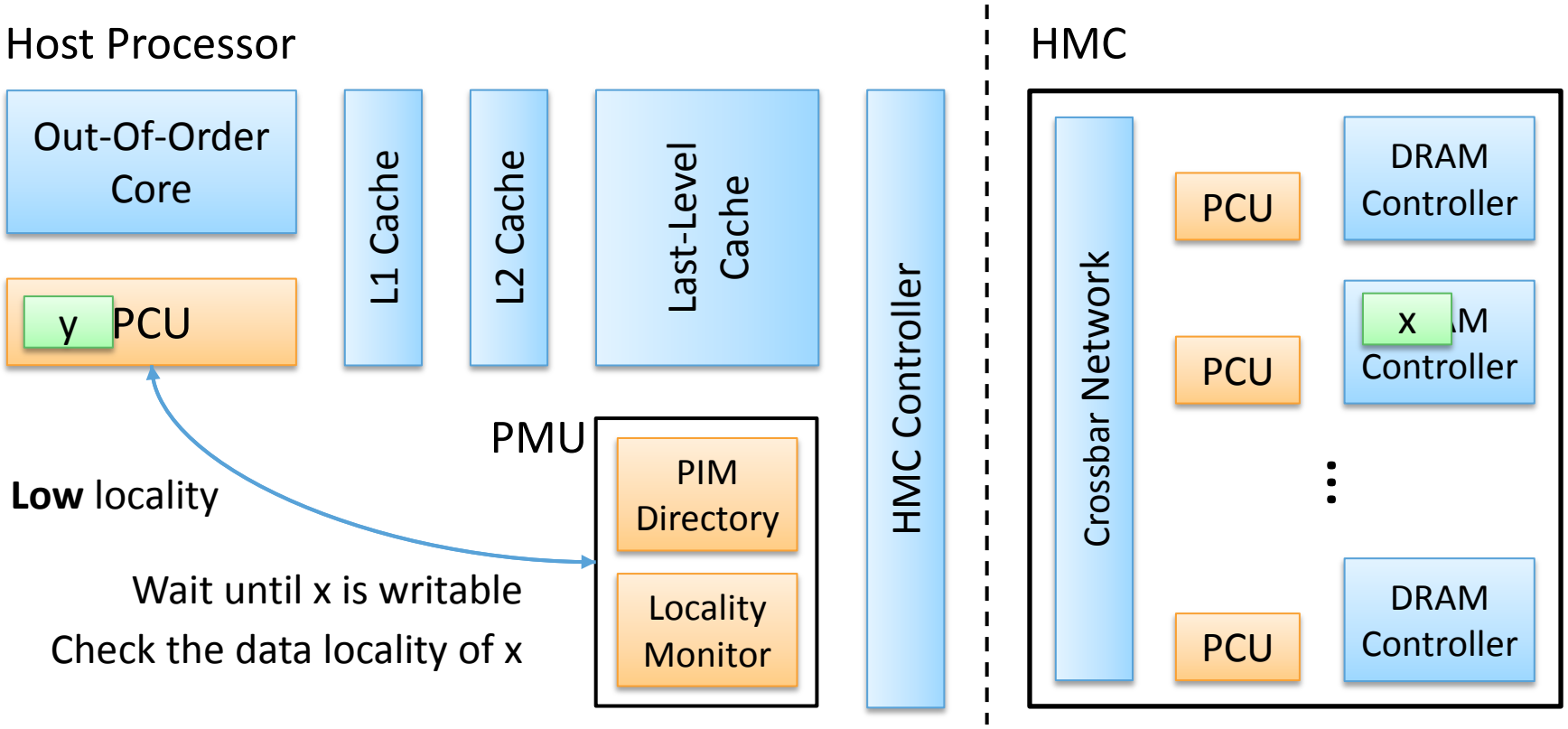



pim.add y, &x


Partial Tag Array
Tag Tag Tag ... Tag
Tag Tag Tag ... Tag
Tag Tag Tag ... Tag
Address
Hit: High locality
Miss: Low locality
Updated on
• Each LLC access
• Each issue of a PIM operation to memory
PIM Directory
Locality Monitor
Wait until x is writable
Check the data locality of x
PCU
DRAM Controller

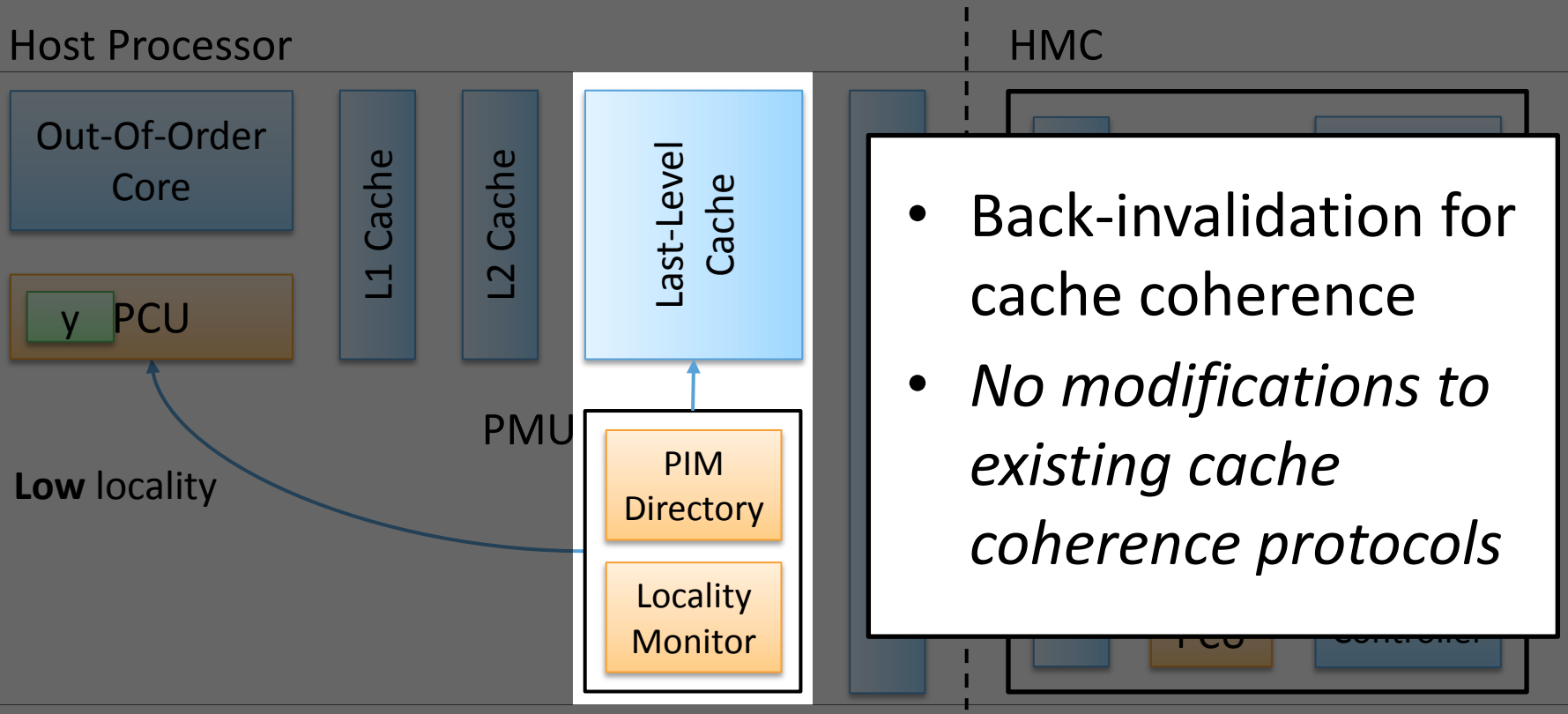
pim.add y, &x

# Memory-side PEI Execution



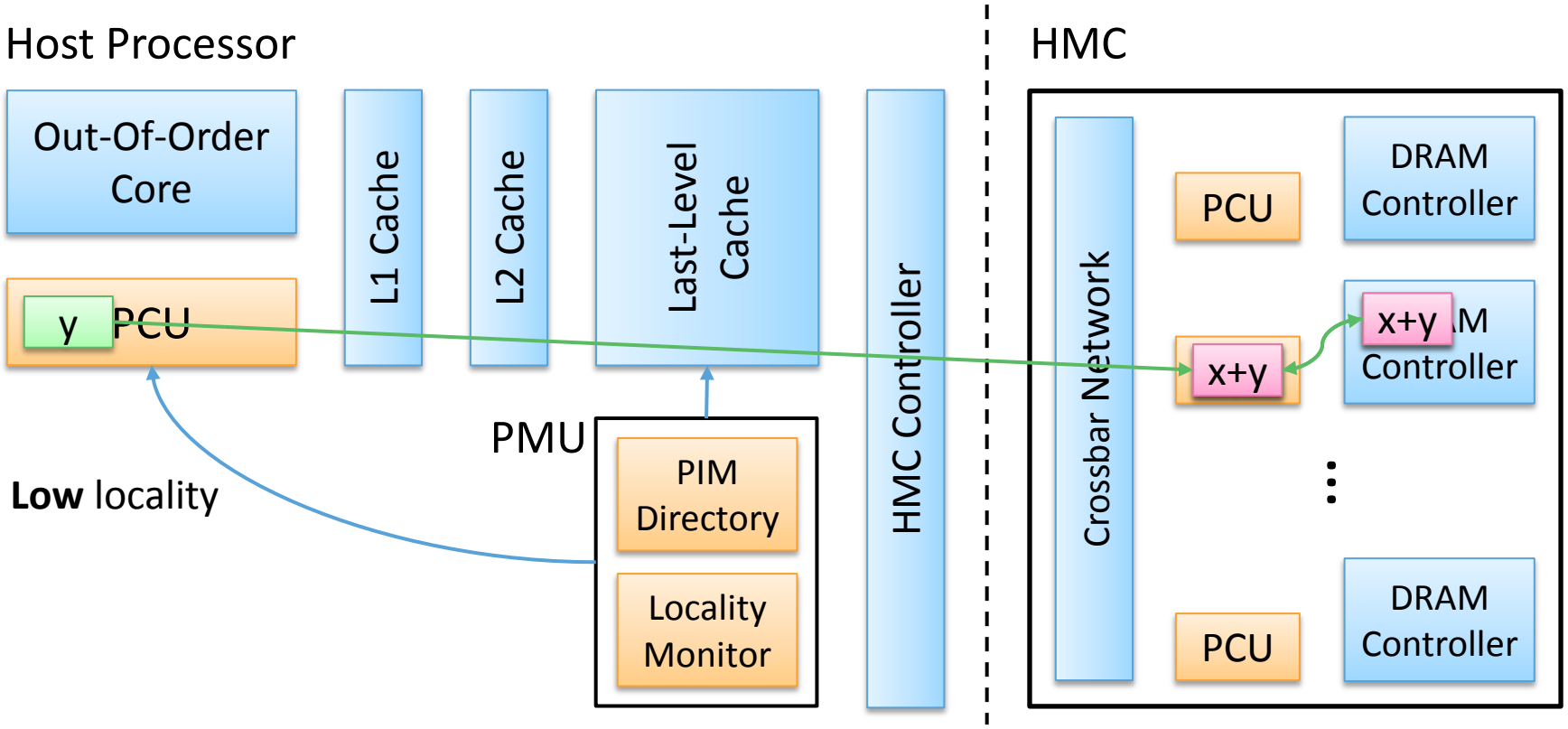

pim.add y, &x

# Memory-side PEI Execution



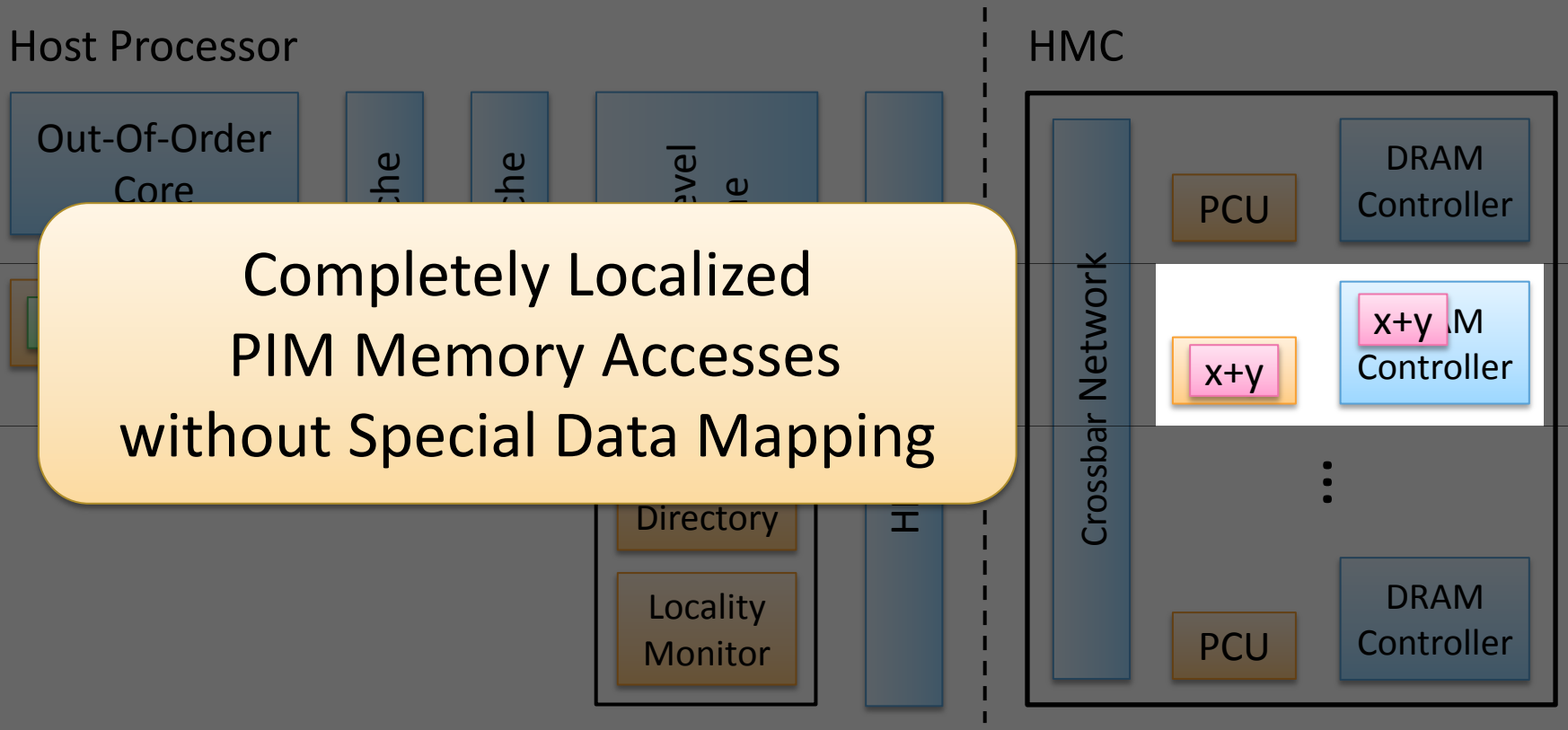

pim.add y, &x

# Memory-side PEI Execution



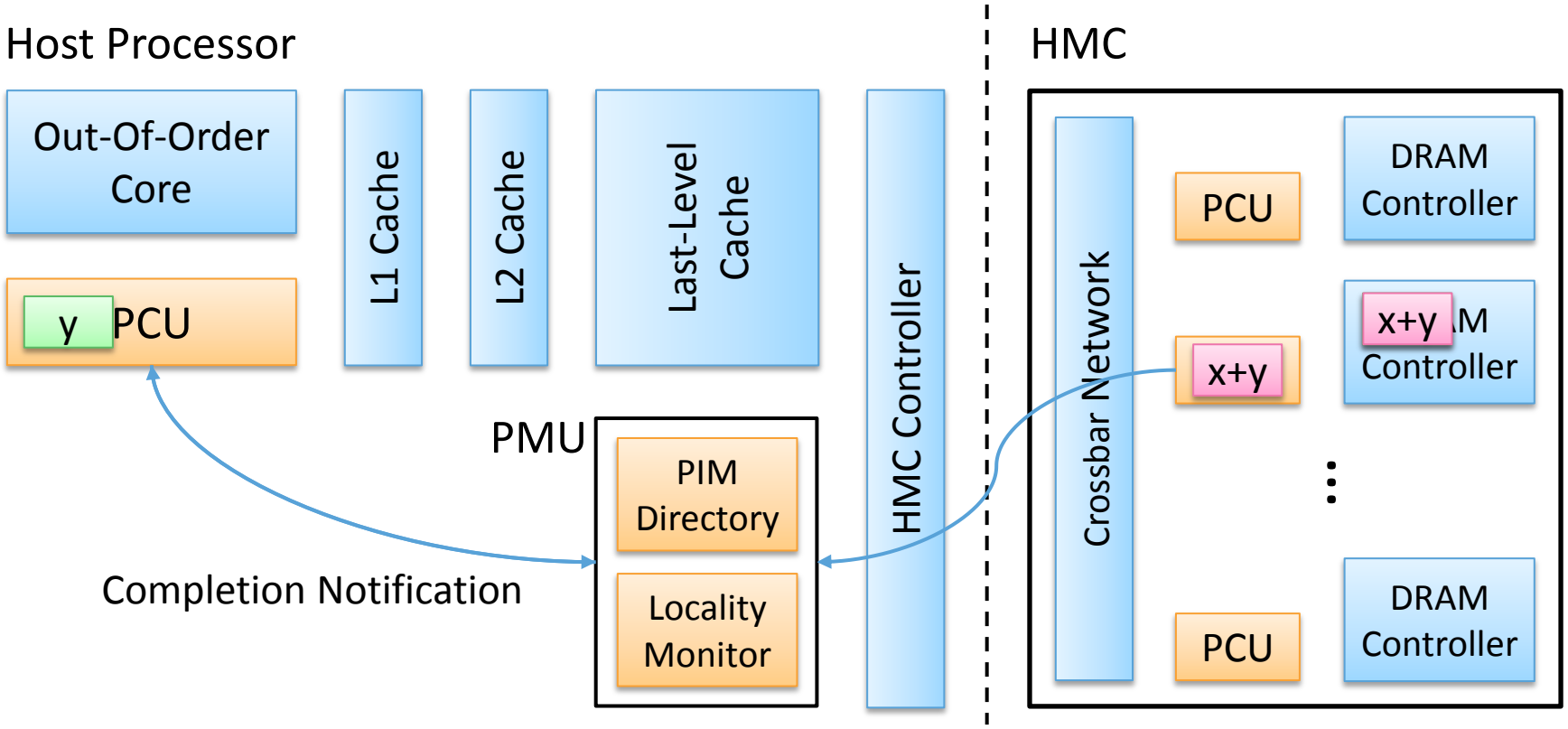

pim.add y, &x

# Memory-side PEI Execution



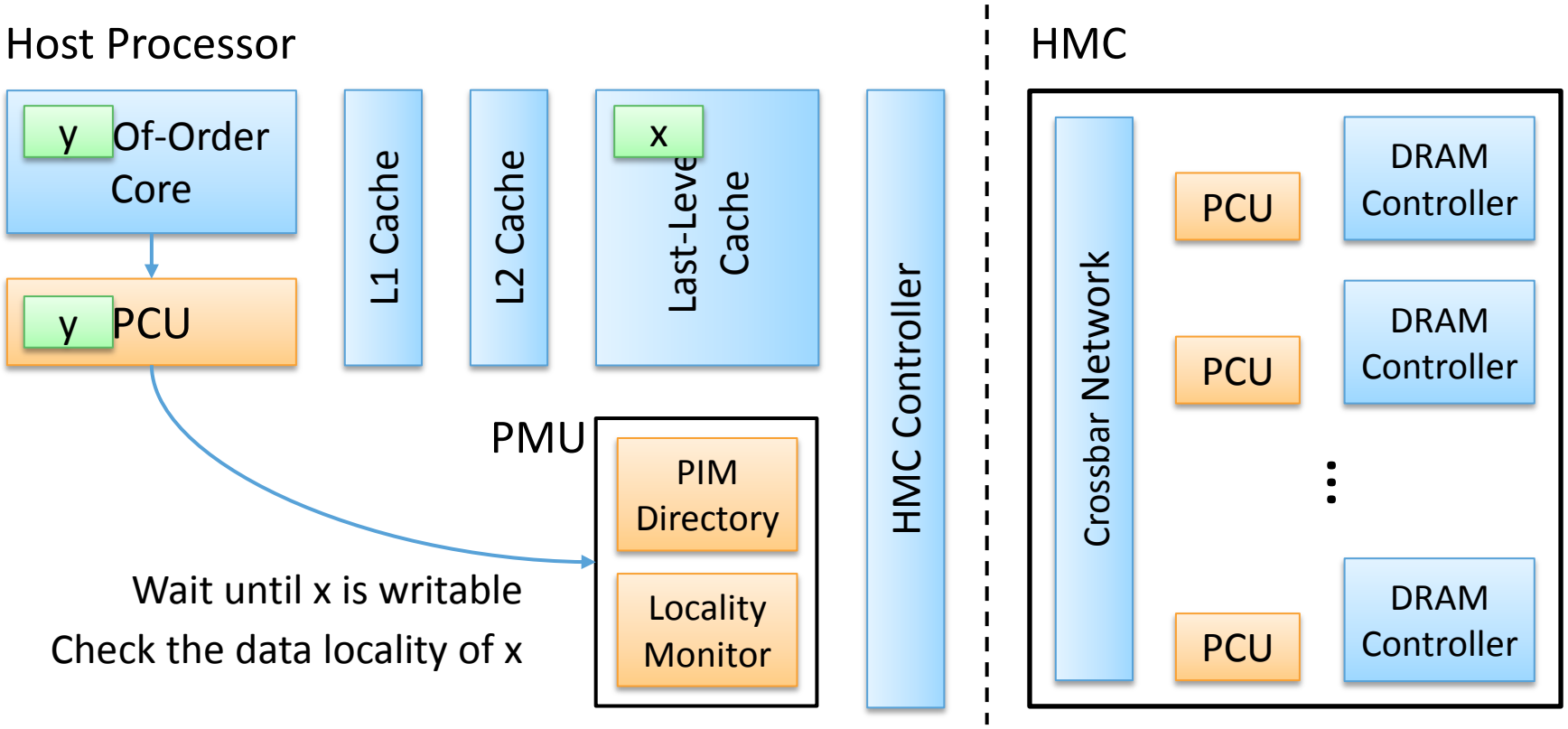

pim.add y, &x

# Memory-side PEI Execution



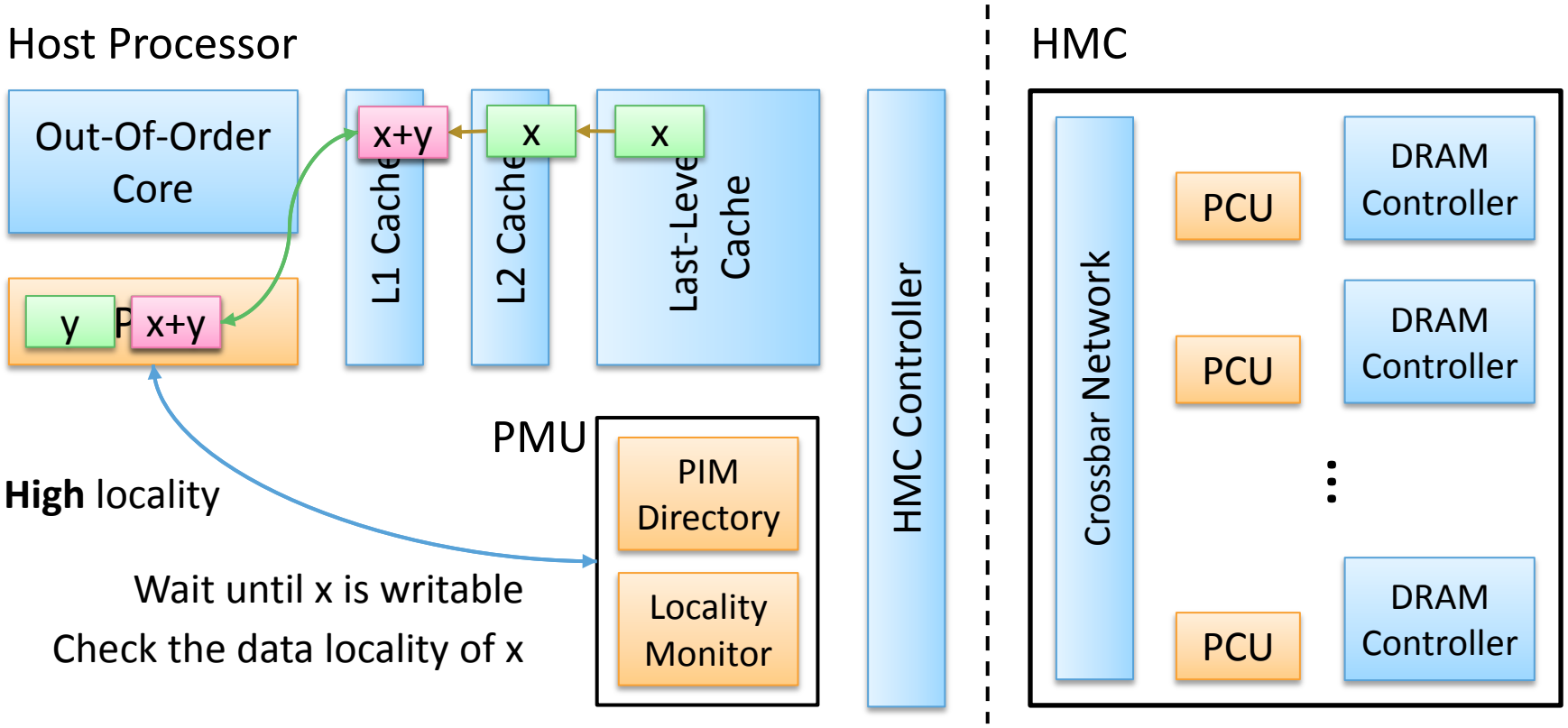

pim.add y, &x

# Host-side PEI Execution



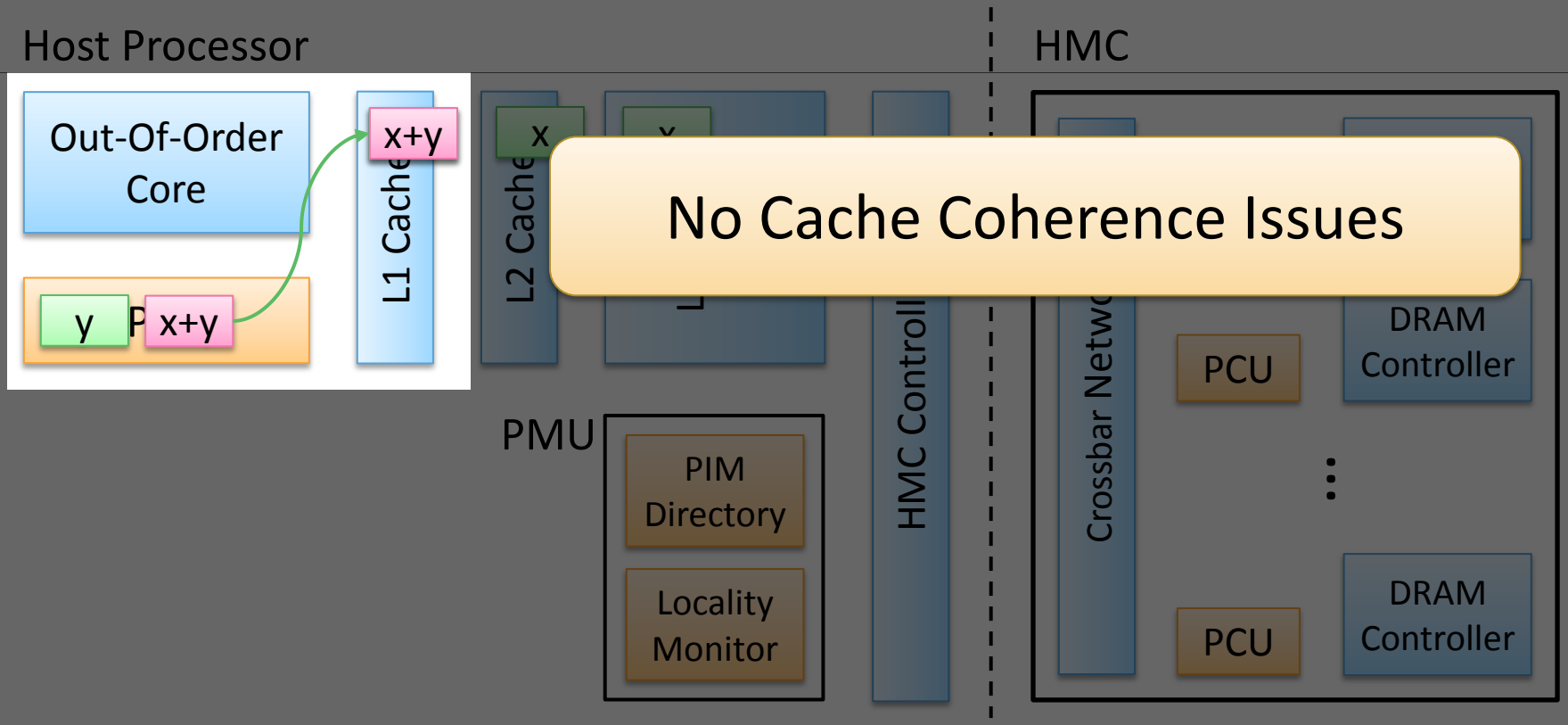

pim.add y, &x

# Host-side PEI Execution



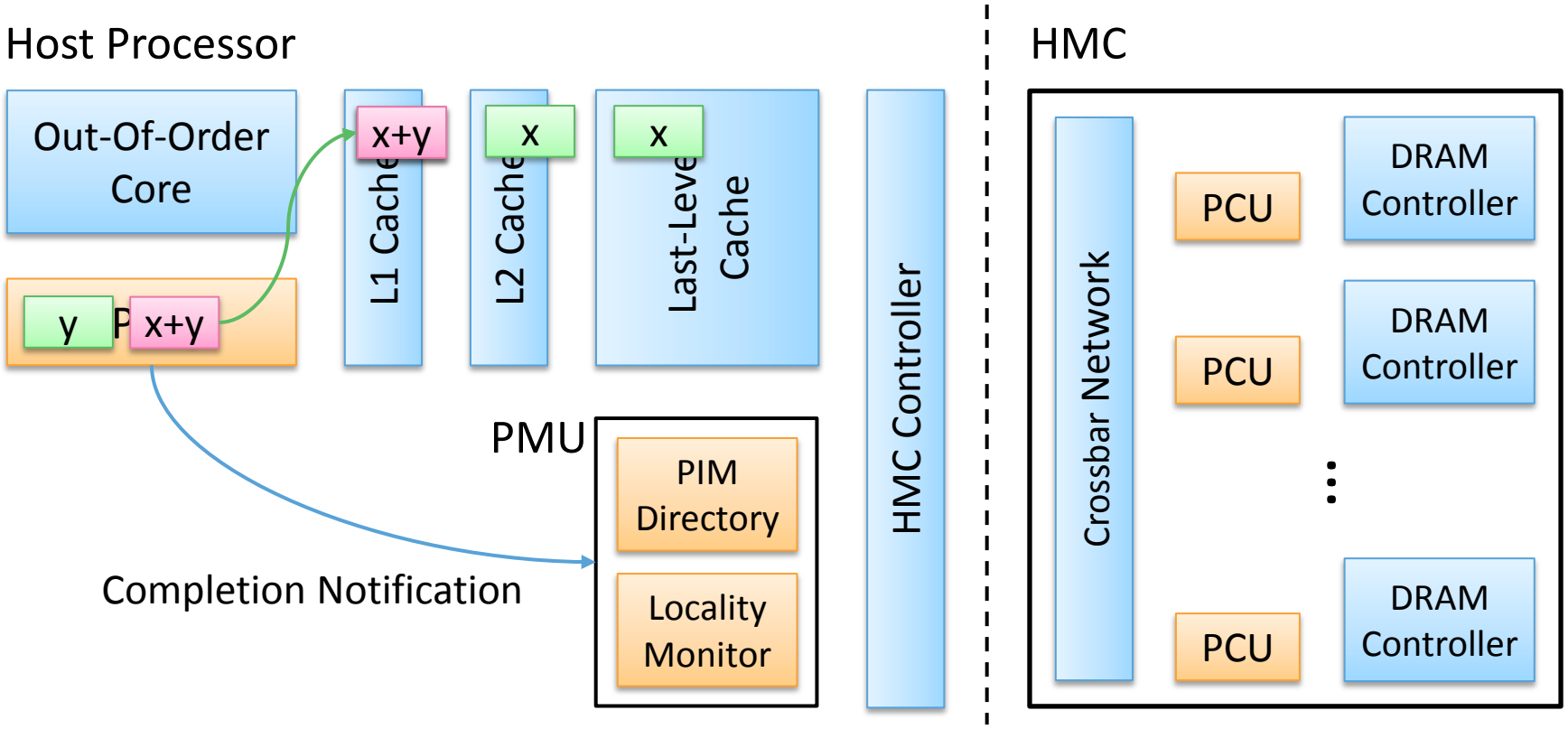

pim.add y, &x

# Host-side PEI Execution




pim.add y, &x

# Host-side PEI Execution




pim.add y, &x

# PEI Execution 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

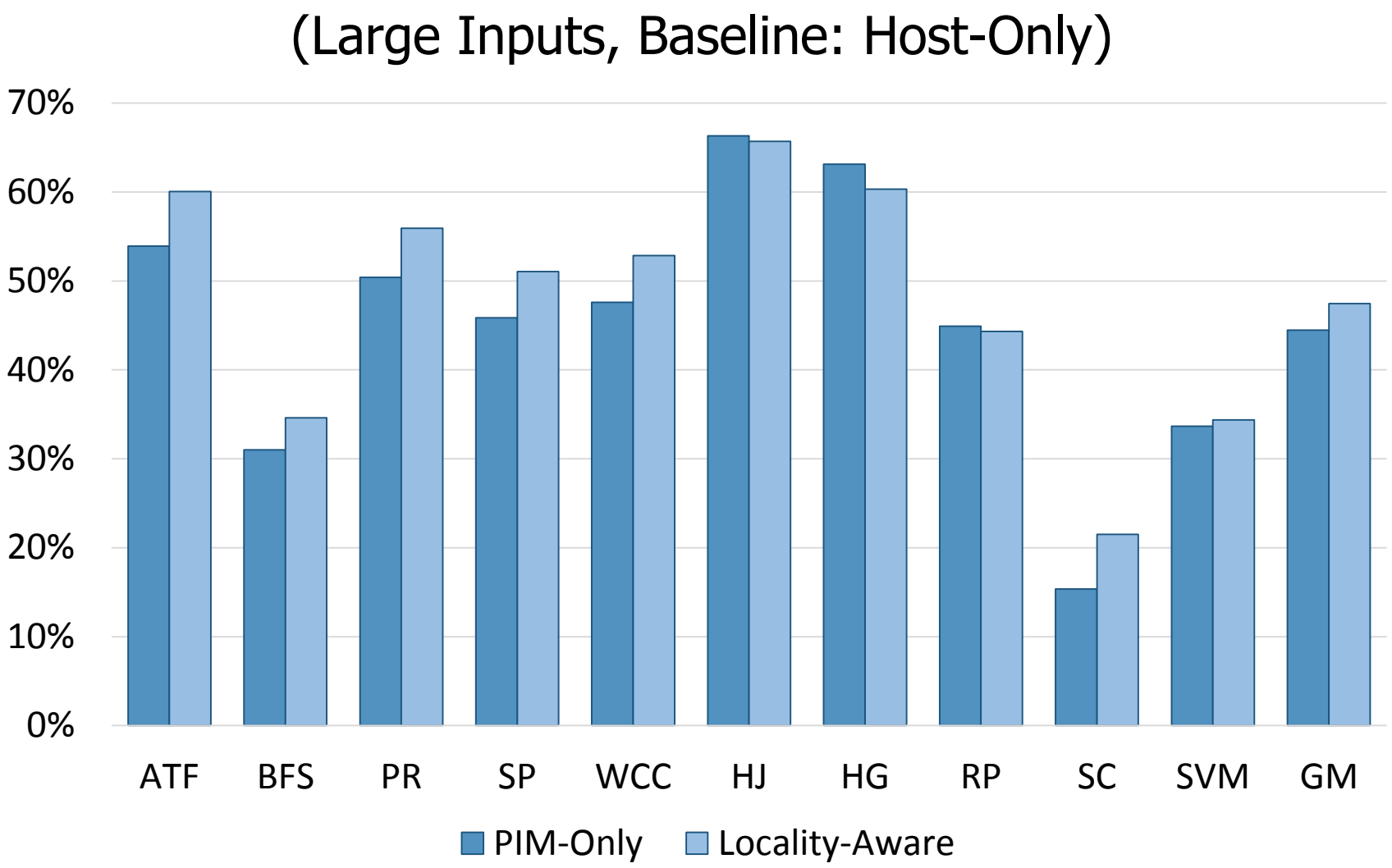
# Evaluation: Simulation Configuration

- 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 (PIM Computation Unit, In Memory)
  - 1-issue computation logic, 4-entry operand buffer
  - 16 host-side PCUs at 4GHz, 128 memory-side PCUs at 2GHz
- PMU (PIM Management Unit, Host Side)
  - PIM directory: 2048 entries (3.25KB)
  - Locality monitor: similar to LLC tag array (512KB)

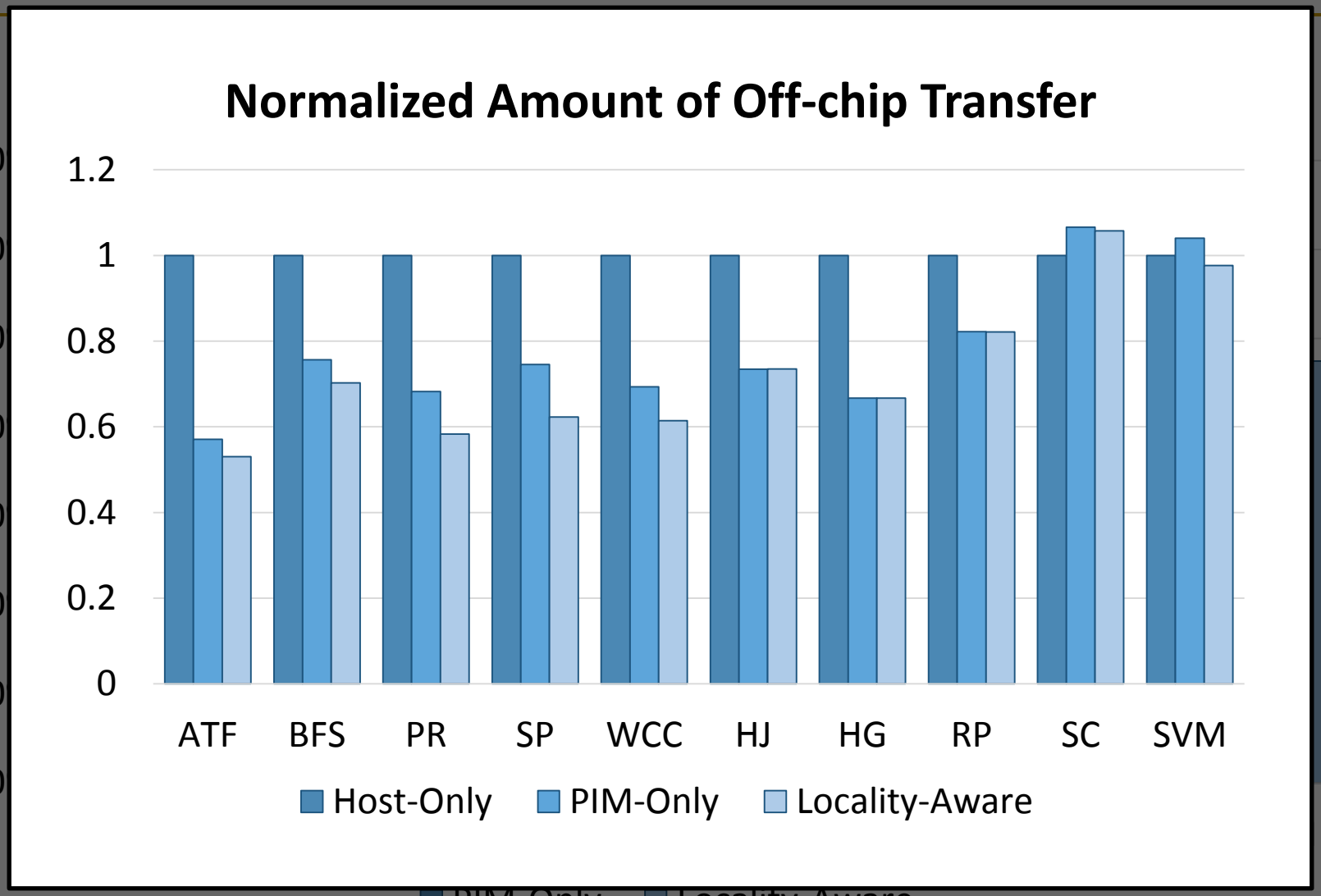
# Evaluated Data-Intensive Applications

- 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

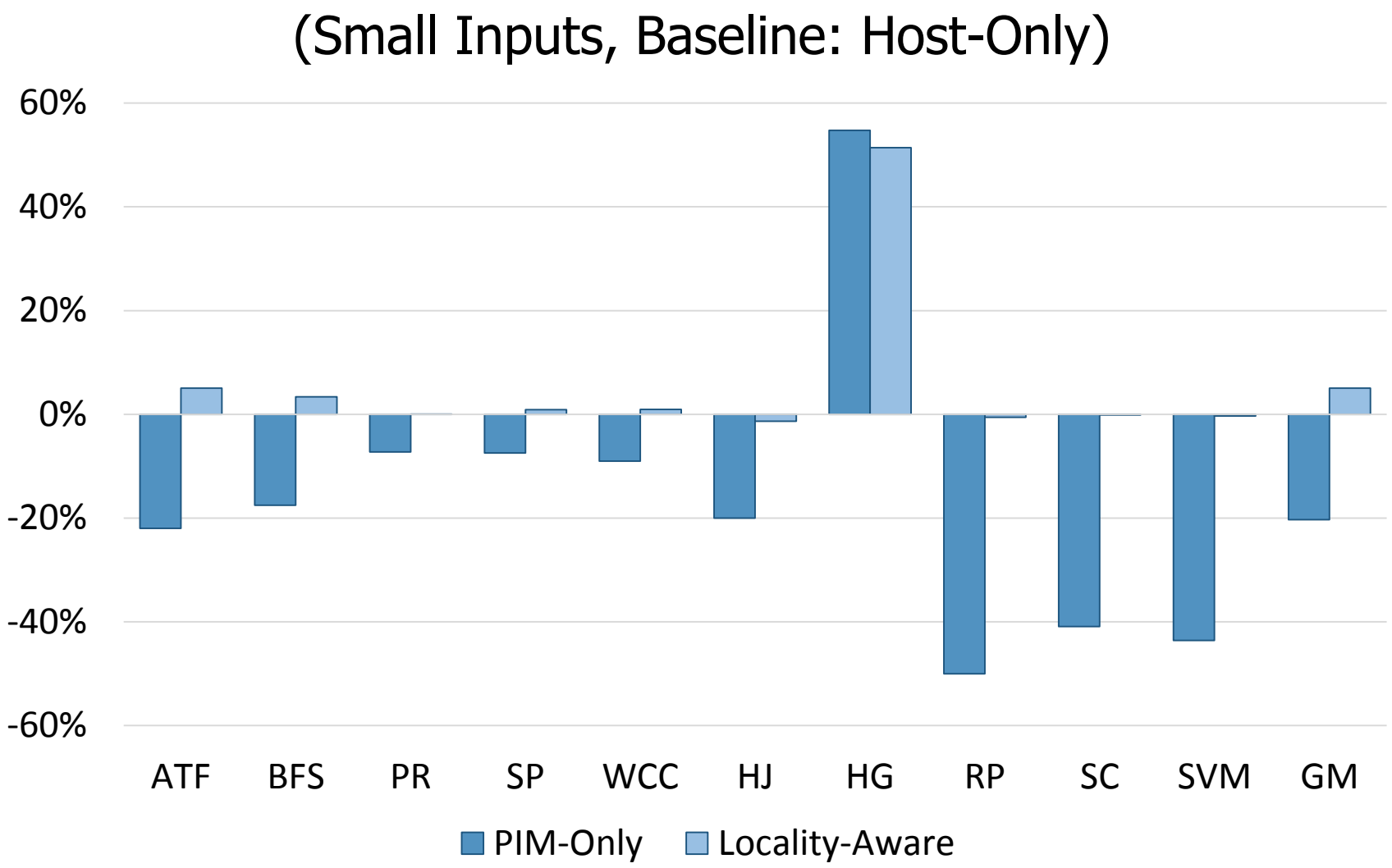
# PEI Performance Delta: Large Data Sets



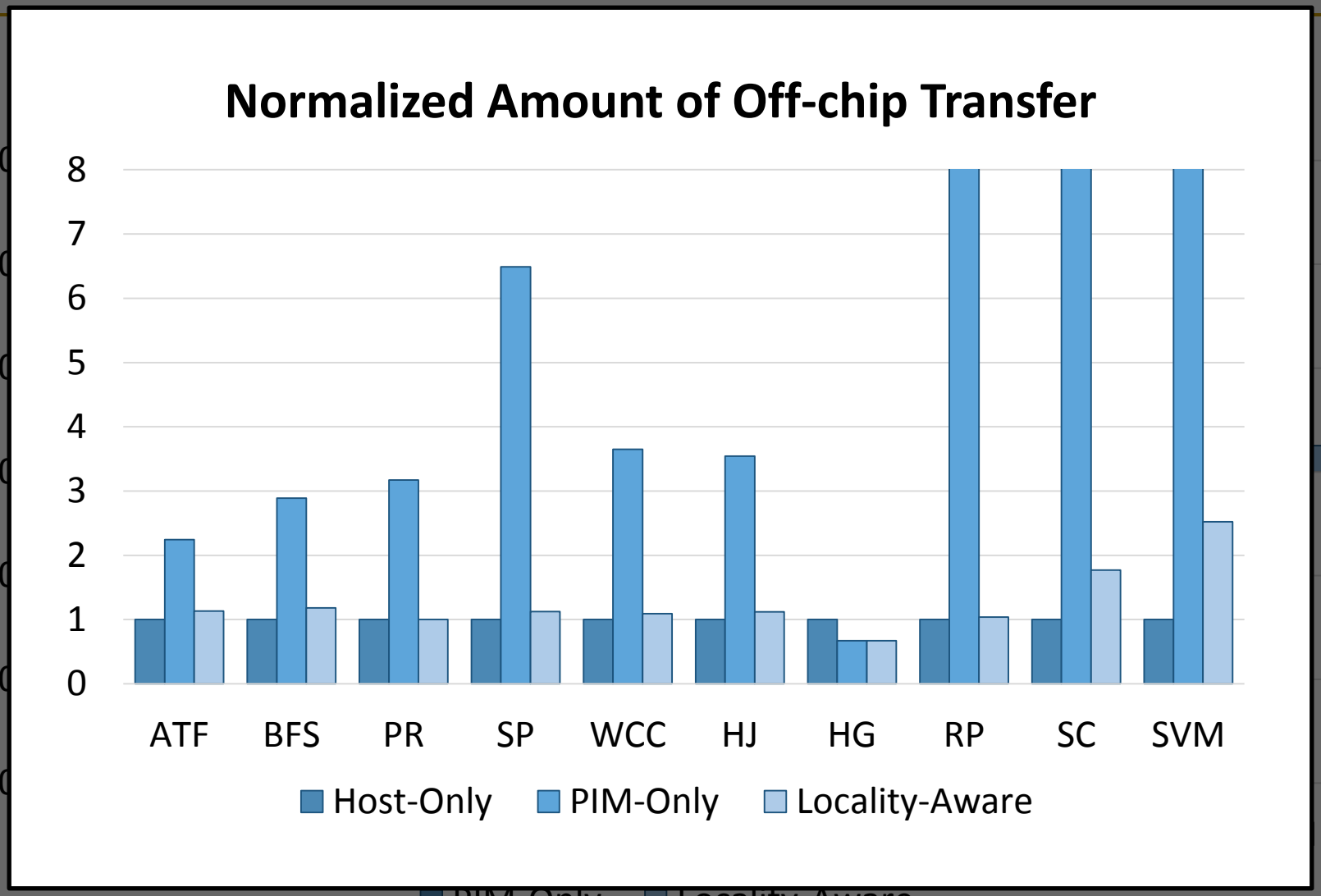
# PEI Performance: Large Data Sets



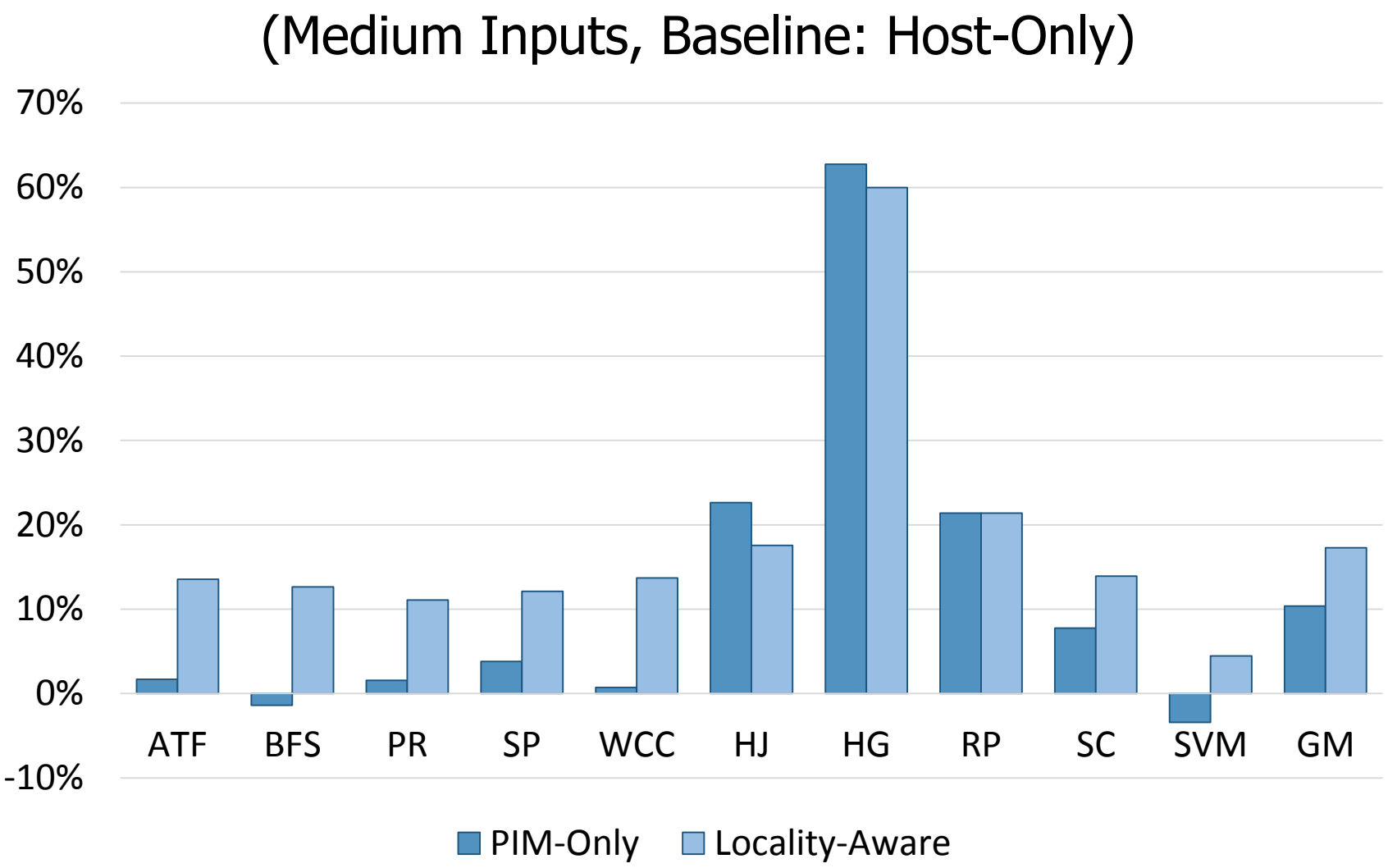
# PEI Performance Delta: Small Data Sets



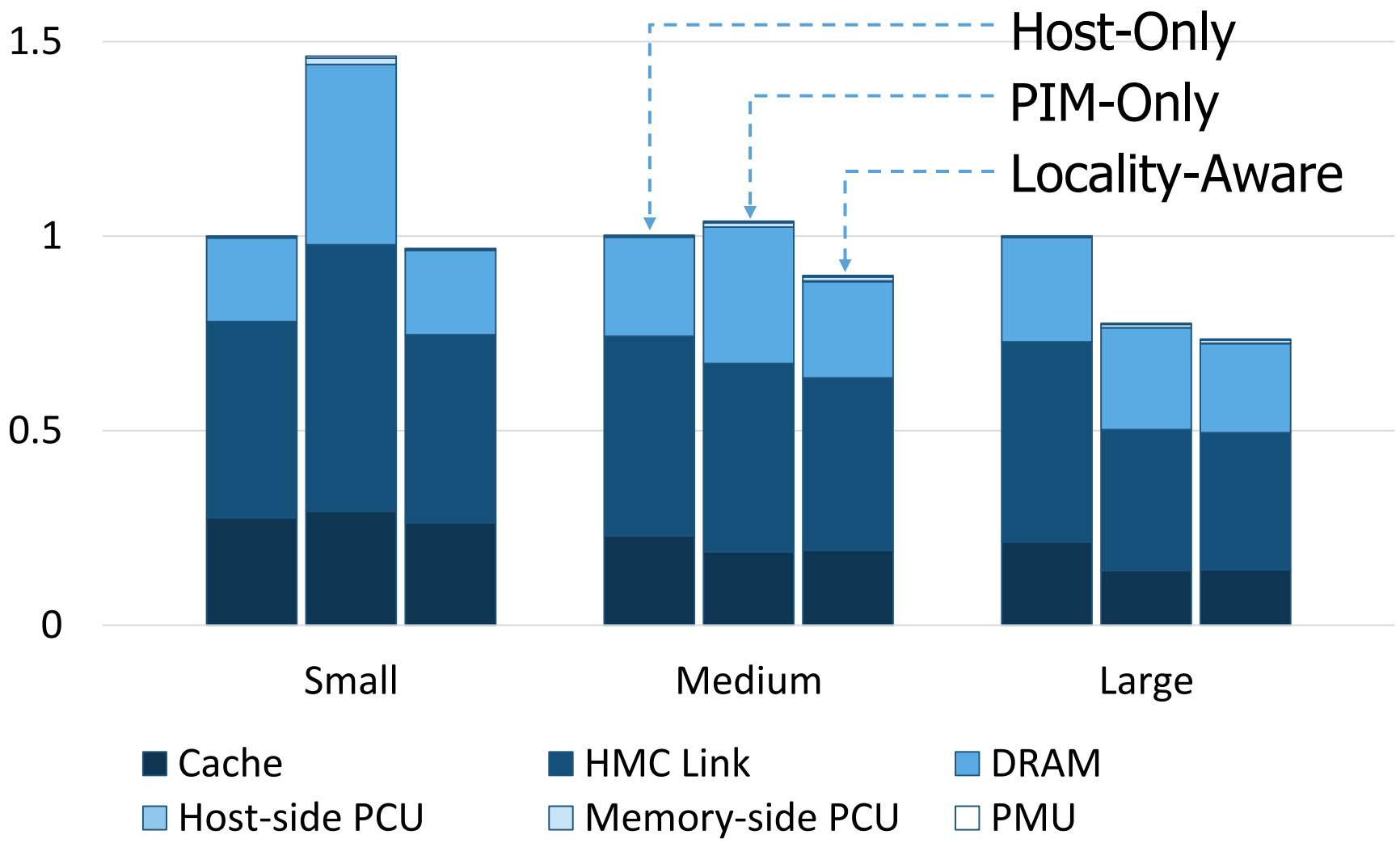
# PEI Performance: Small Data Sets



# PEI Performance Delta: Medium Data Sets



# PEI Energy Consumption



# Summary: Simple Processing-In-Memory

- PIM-enabled Instructions (PEIs): Expose PIM operations as *cache-coherent, virtually-addressed host processor instructions*
  - No changes to the existing sequential programming model
  - No changes to virtual memory
  - Minimal changes for cache coherence
- Locality-aware PEIs: Dynamically determine the location of PEI execution based on data locality without software hints
- PEI performance and energy results are promising
  - 47%/32% speedup over Host/PIM-Only in large/small inputs
  - 25% node energy reduction in large inputs
  - Good adaptivity across randomly generated workloads

# Two Key Questions in 3D Stacked PIM

- What is the minimal processing-in-memory support we can provide ?
  - without changing the system significantly
  - while achieving significant benefits of processing in 3D-stacked memory

- How can we accelerate important applications if we use 3D-stacked memory as a coarse-grained accelerator?
  - what is the architecture and programming model?
  - what are the mechanisms for acceleration?