ISCA 2015

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, Kiyoung Choi

PIM-Enabled Instructions

A Low-Overhead, Locality-Aware Processing-in-Memory Architecture

Matthias Möhr

Problem sketch

- Contemporary data-intense applications
- Increasingly fast processors
- Memory Wall

Processing in memory

- Higher memory bandwidth
- Lower memory latency
- Better energy efficiency

Processing in memory: A short history

- Idea not new
- Practicality concerns in earlier studies
- Expensive hardware, new programming models
- Not taking advantage of on-chip caches
- New memory technologies today: Hybrid memory cube

Key challenges for processing in memory

- Cost
- Programming model
- Cache Coherence/Virtual memory use

Key challenges for processing in memories

- Cost HMC
- Programming model
- Coherence/Virtual memory use

Key challenges for processing in memories

- Cost HMC
- Programming model ?
- Coherence/Virtual memory use ?

Goals of the paper

- Provide an intuitive programming model for PIM
- Support cache coherence and virtual memory
- Reduce the implementation overhead of PIM units

Idea

- Exploit new stacking technologies to improve system performance and energy efficiency
- Use only minimal extension of the host processor’s ISA
- Only allow simple in-memory computation

- Careful with PIM! Usage may as well lead to performance degradation
- Benefit of PIM heavily depends on data locality

Major Contribution

- Introduce PIM-enabled instructions (=: PEI)
- PEI are instructions that *can* be executed in memory but they don’t have to; they can also be executed in the host processor
- PIM := programming in memory

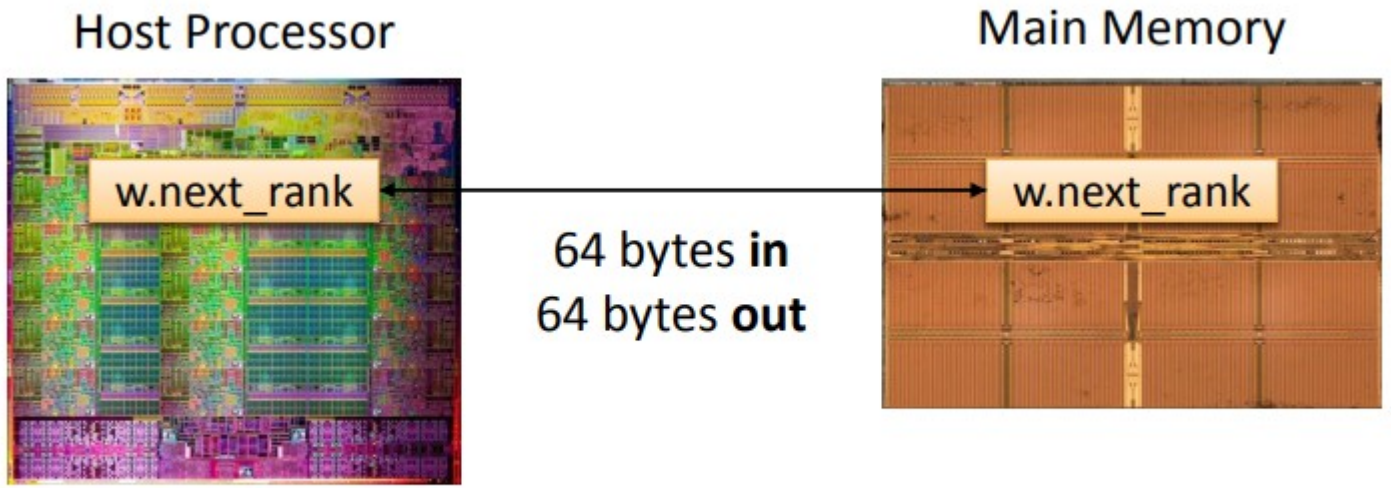
- Introduction of single cache block restriction:
- Memory region that can be accessed during the execution of a PEI is limited to one last-level cache block

Example use case: Parallel page-rank alg.

(Excerpt)

```
7     parallel_for (v: graph.vertices) {
8       delta = 0.85 * v.pagerank / v.out_degree;
9       for (w: v.successors) {
10        atomic w.next_pagerank += delta;
11      }
12    }
```

Bottleneck!

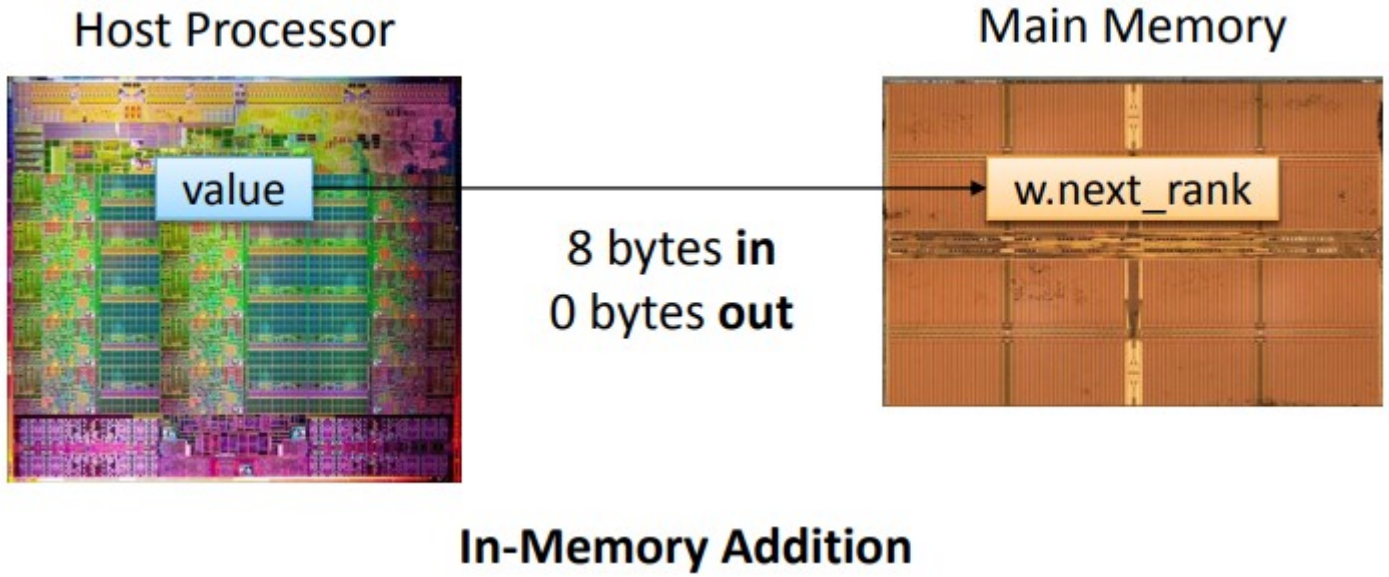


Conventional Architecture

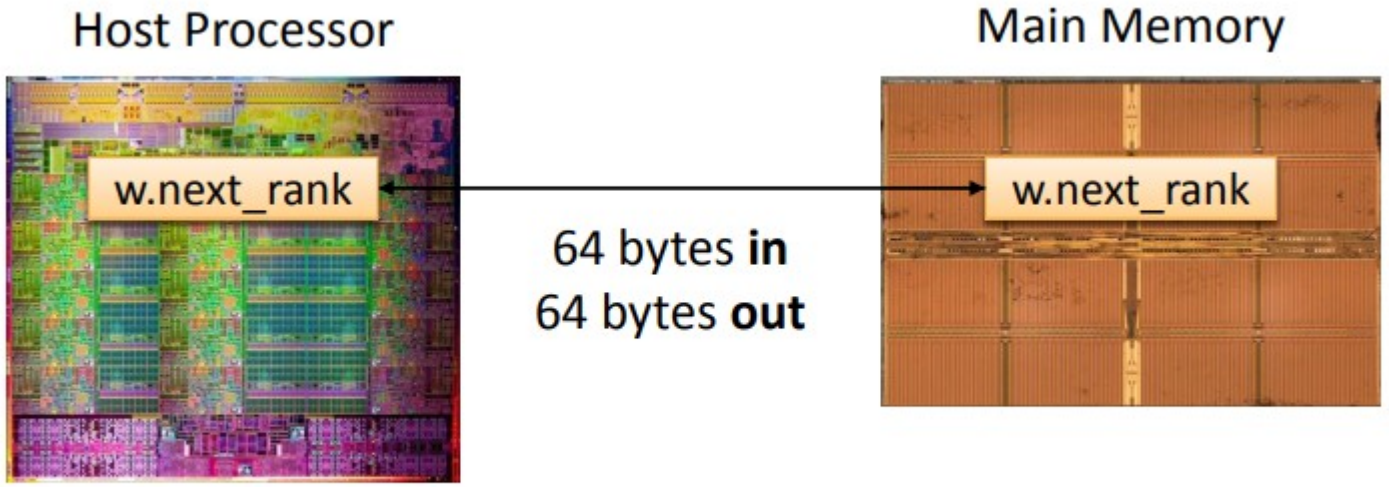
Example use case: Parallel page-rank alg.

(Excerpt)

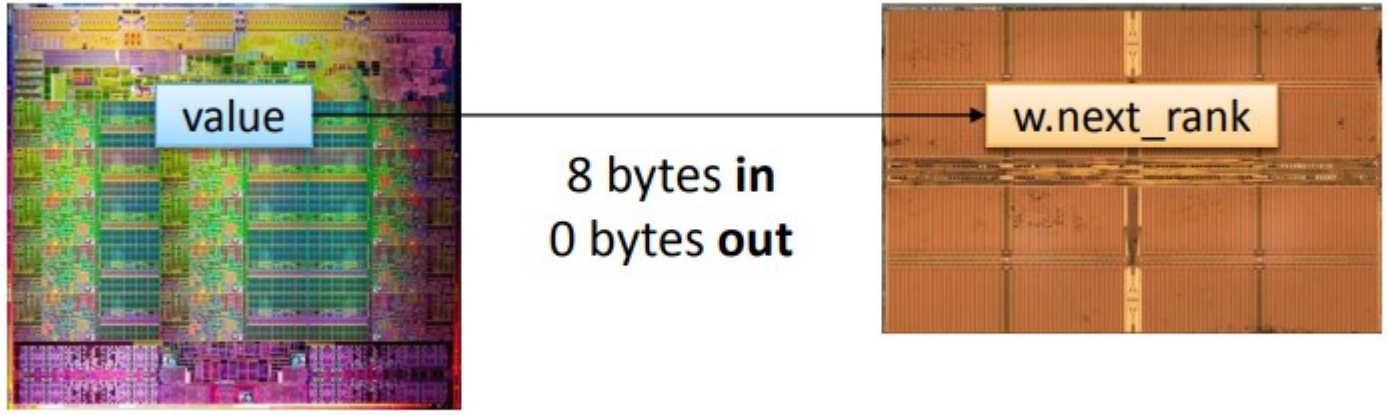
```
7     parallel_for (v: graph.vertices) {
8       delta = 0.85 * v.pagerank / v.out_degree;
9       for (w: v.successors) {
10        PIM  w.next_pagerank += delta;
11      }
12    }
```



Example use case: Parallel page-rank alg

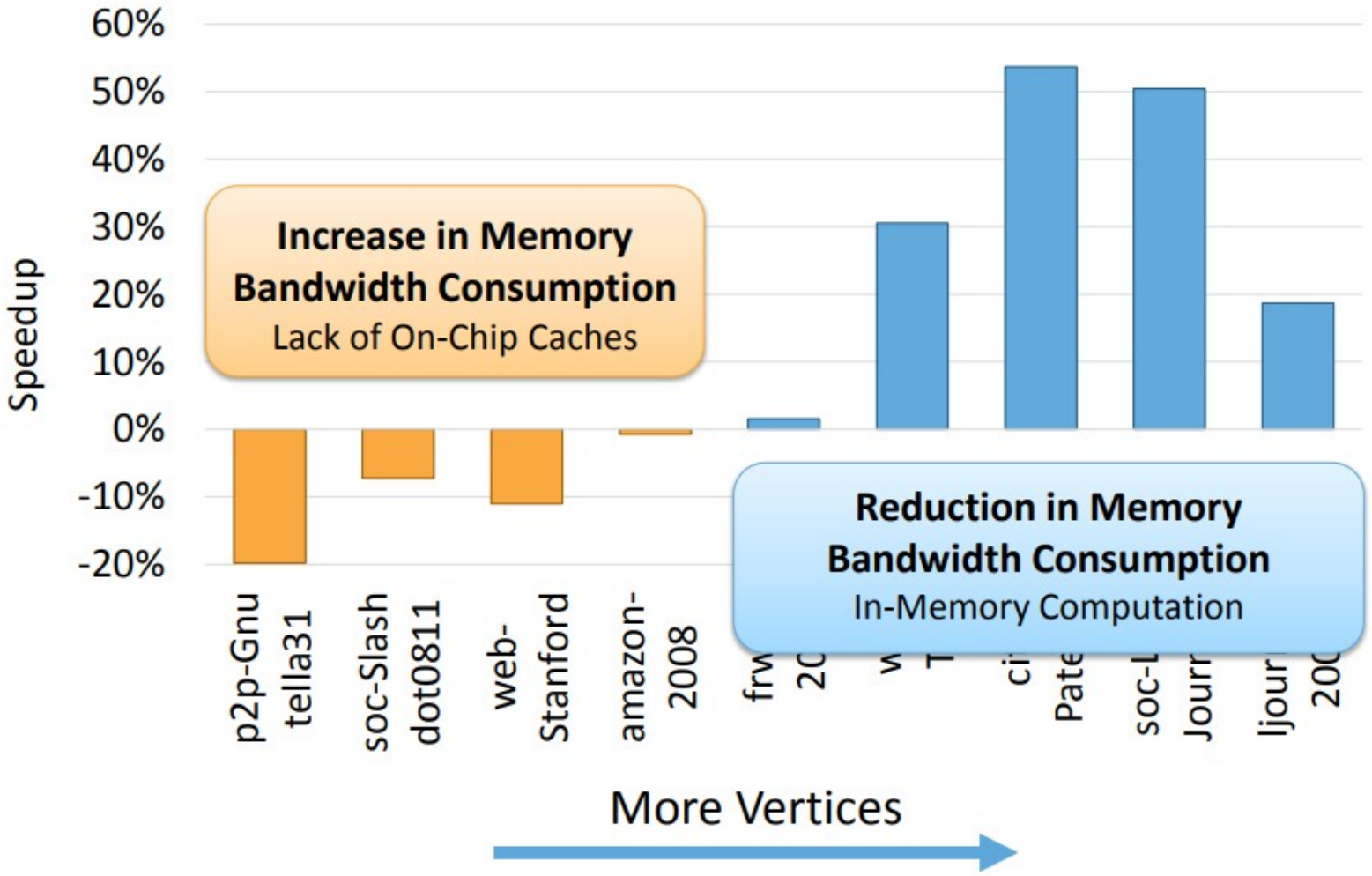


Conventional Architecture



In-Memory Addition

Result: Use of PIM on different Graphs



PEI

Processing-in-memory-enabled instructions

Mechanism

- PEI are expressed as specialized instructions of the host processor
- For any supported PIM instruction of the memory, the host’s ISA is extended by a corresponding PEI
  - For page rank: If the memory supports an atomic add, host has to provide a PIM-enabled atomic add to the user

Mechanism: Page-rank

```
7     parallel_for (v: graph.vertices) {
8       delta = 0.85 * v.pagerank / v.out_degree;
9       for (w: v.successors) {
10          PEI   w.next_pagerank += delta;
11      }
12    }
```

- Can be executed in memory OR host processor
- Cache-coherent
- Programmer can use virtual address space
- What about atomicity? -> We well see in a second

Mechanism

- PEI are expressed as specialized instructions of the host processor
- For any supported PIM instruction of the memory, the host’s ISA is extended by a corresponding PEI
  - For page rank: If the memory supports an atomic add, host has to provide a PIM-enabled atomic add to the user
- Integration into existing software is done through simple instruction replacement
- Hardware mechanism decides dynamically and per operation where to execute a command
- Software is unaware of that decision

Atomicity

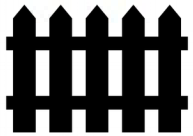
- Atomicity of a PEI between any PEI given by the architecture
- Atomicity of PEI with respect to normal instructions not guaranteed
- To prevent this, every memory access would have to be checked
  -> introduces overhead for regular instructions
- Host processor is supposed to provide a PIM memory fence instruction to enforce memory ordering

Memory fence

- Blocks all host processor execution until all PEI before the fence are completed
- Generally introduces an overhead
- Very notable in academic examples
- Paper claims that it usually can be amortized in real-world application
- E.g. page rank issues millions or even more PEI at the given line, equivalent to the number of edges in the graph

Memory fence in page-rank

```
7     parallel_for (v: graph.vertices) {
8       delta = 0.85 * v.pagerank / v.out_degree;
9       for (w: v.successors) {
10                PEI   w.next_pagerank += delta;
11      }
12    }
13  pfence();
```



Single cache block restriction revisited

- Restrict data access of one PEI to a single last-level cache block
- Why?
- Ensures that accessed data is bounded to a single DRAM module
- Cache coherence and virtual memory management for PIM can use the same hardware that supports those for last-level cache
- Locality of data can easily be identified using the tag array of the last-level cache

Hardware Architecture

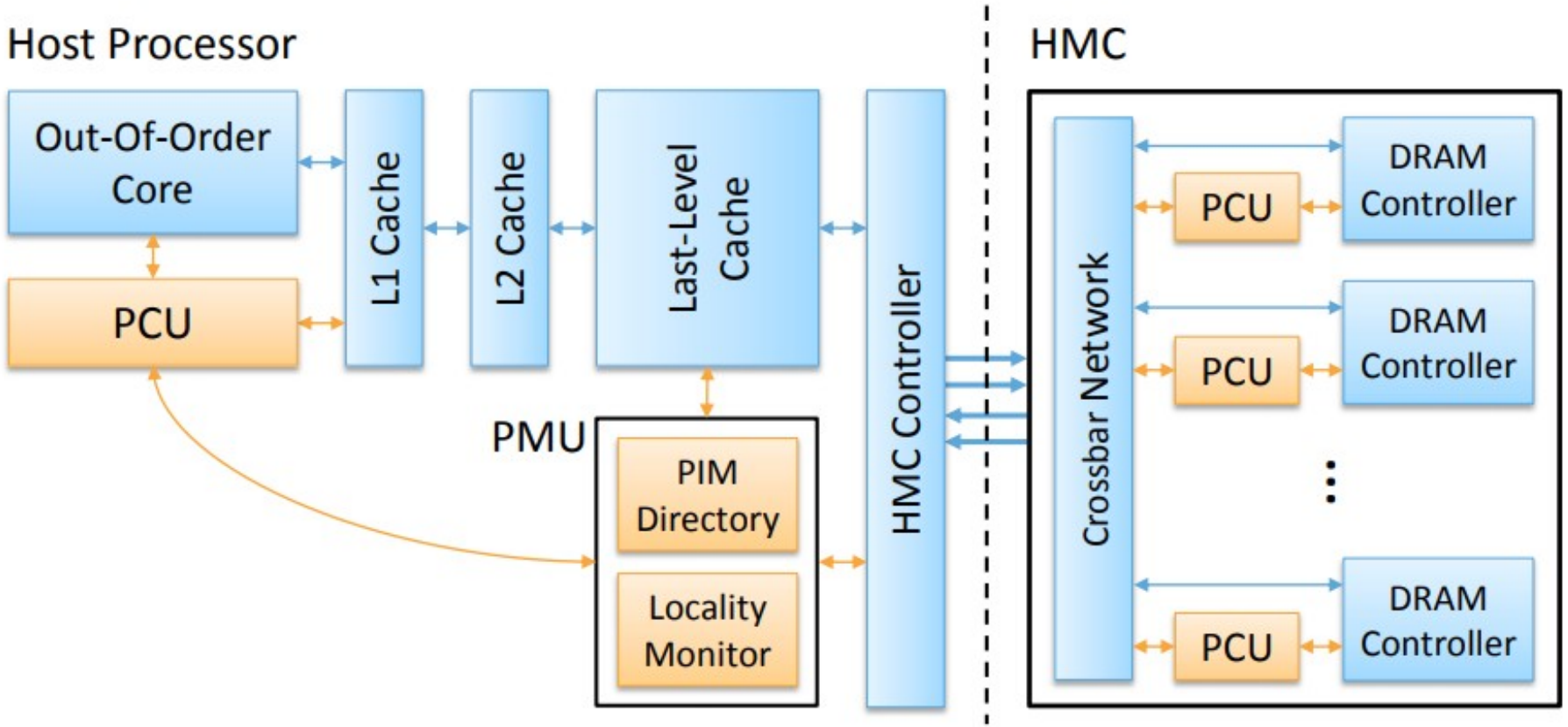
- Not limited to specific types of instructions
- Can easily implement any simple general purpose PIM operation
- Assumes to use Hyper Memory Cube
  - Multiple vertical DRAM partitions
  - Every DRAM partition has separate DRAM controllers on the logic unit
- Easily transferable design, does not depend on very specific memory properties

Hardware Architecture

- Key features:
- Support PIM operations as part of the host processor’s instruction set
- Identify PEI with high data locality and let them be executed on host

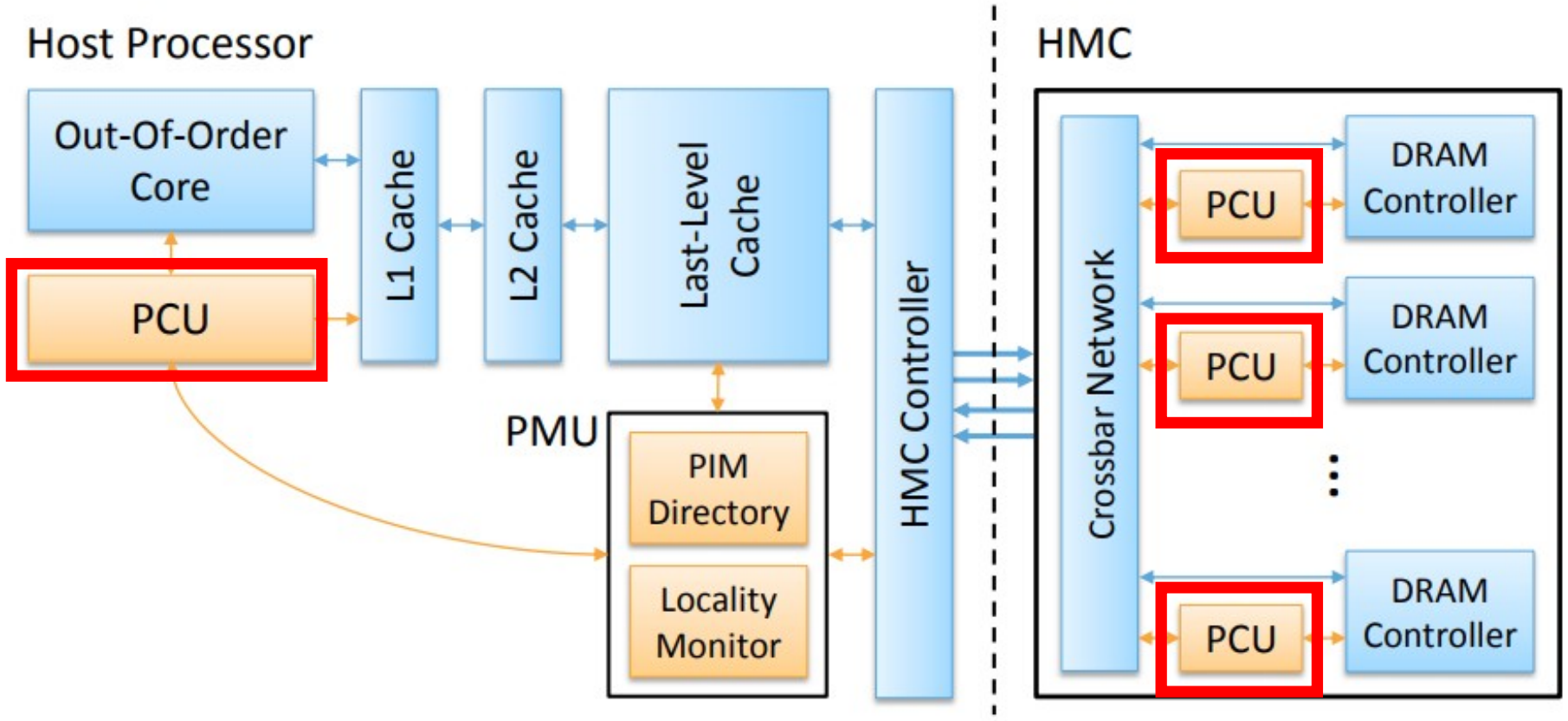
- Modifications:
- Add PEI computation unit (=: PCU) to each memory unit and also each host processor for design simplicity
- Add a PEI management unit (=: PMU) shared between all host processors near last-level cache

Hardware Architecture



Proposed PEI Architecture

Hardware Architecture



Proposed PEI Architecture

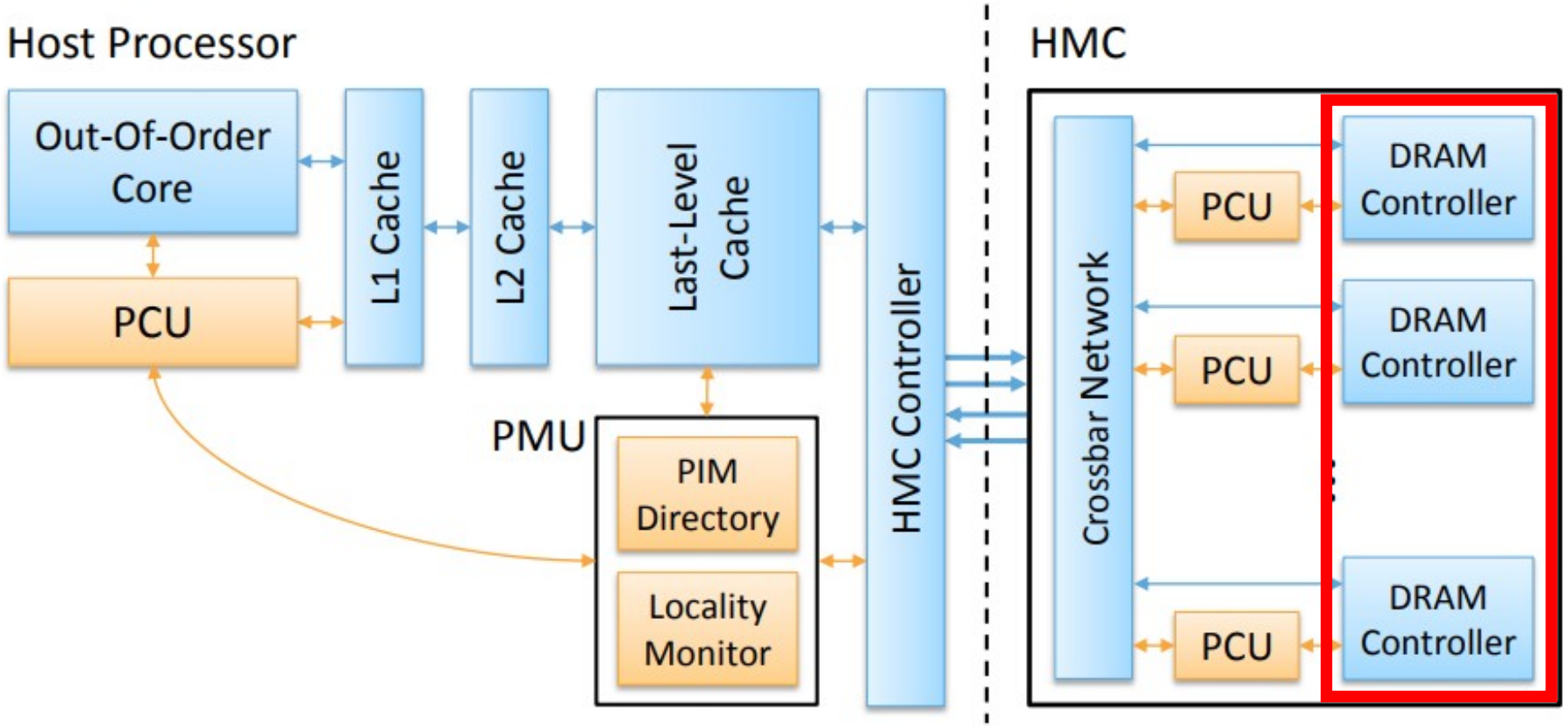
PCU Architecture

- Hardware unit that executes PEI
- PEI can be executed by any PCU equally since they all are the same
- An operand buffer stores in-flight PEI information
  - Implemented as a small SRAM
- When full, PEI are stalled until space becomes available

PCU Interface

- Host controls host-side PCU by manipulating memory-mapped registers inside it. For that, assemblers provide pseudo-instructions that are translated into accesses of those registers as an abstraction
- Memory-side PCUs are controlled by memory (HMC) controllers
- Uses special memory command that can easily be added to the packet-based communication (as claimed)

Hardware Architecture

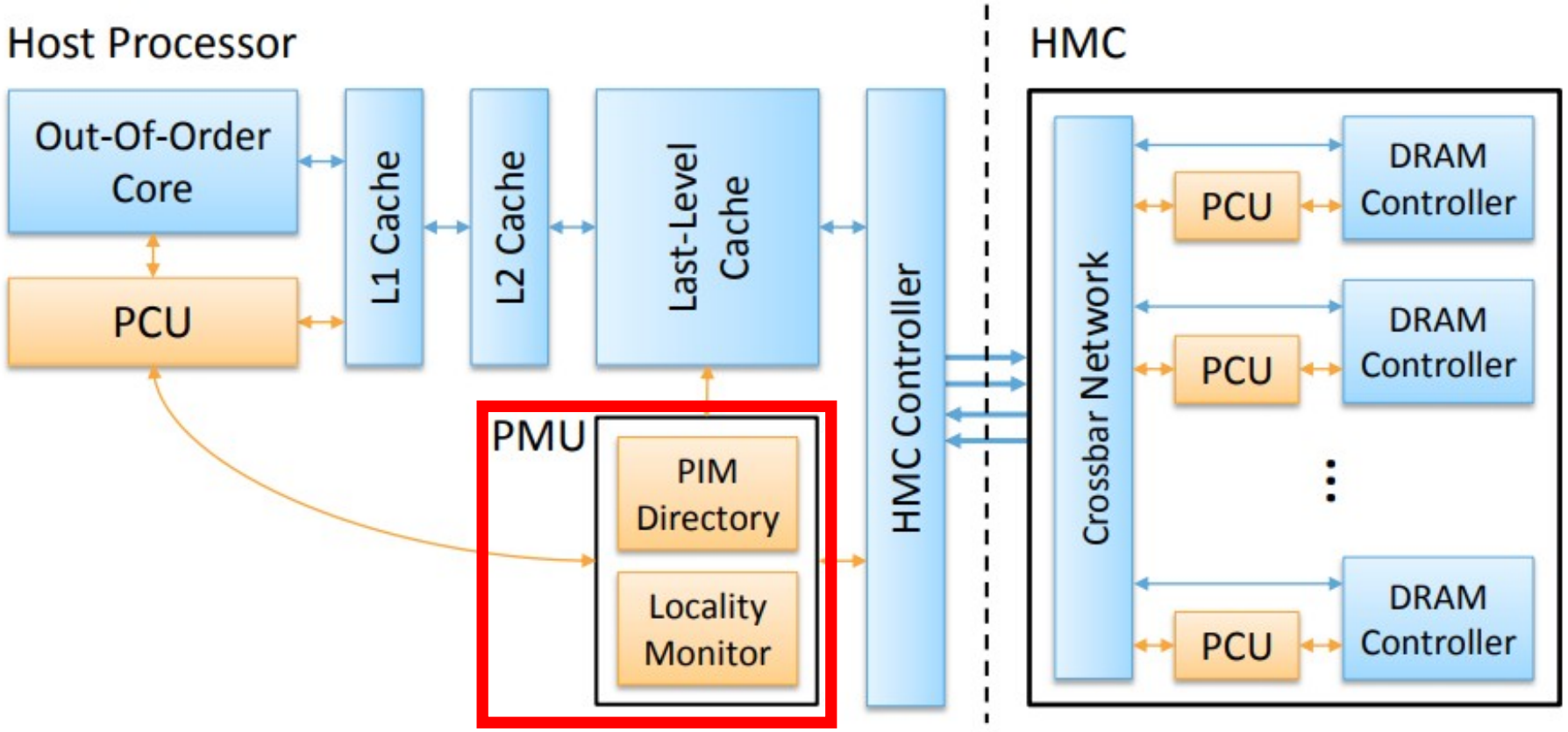


Proposed PEI Architecture

Hardware Architecture

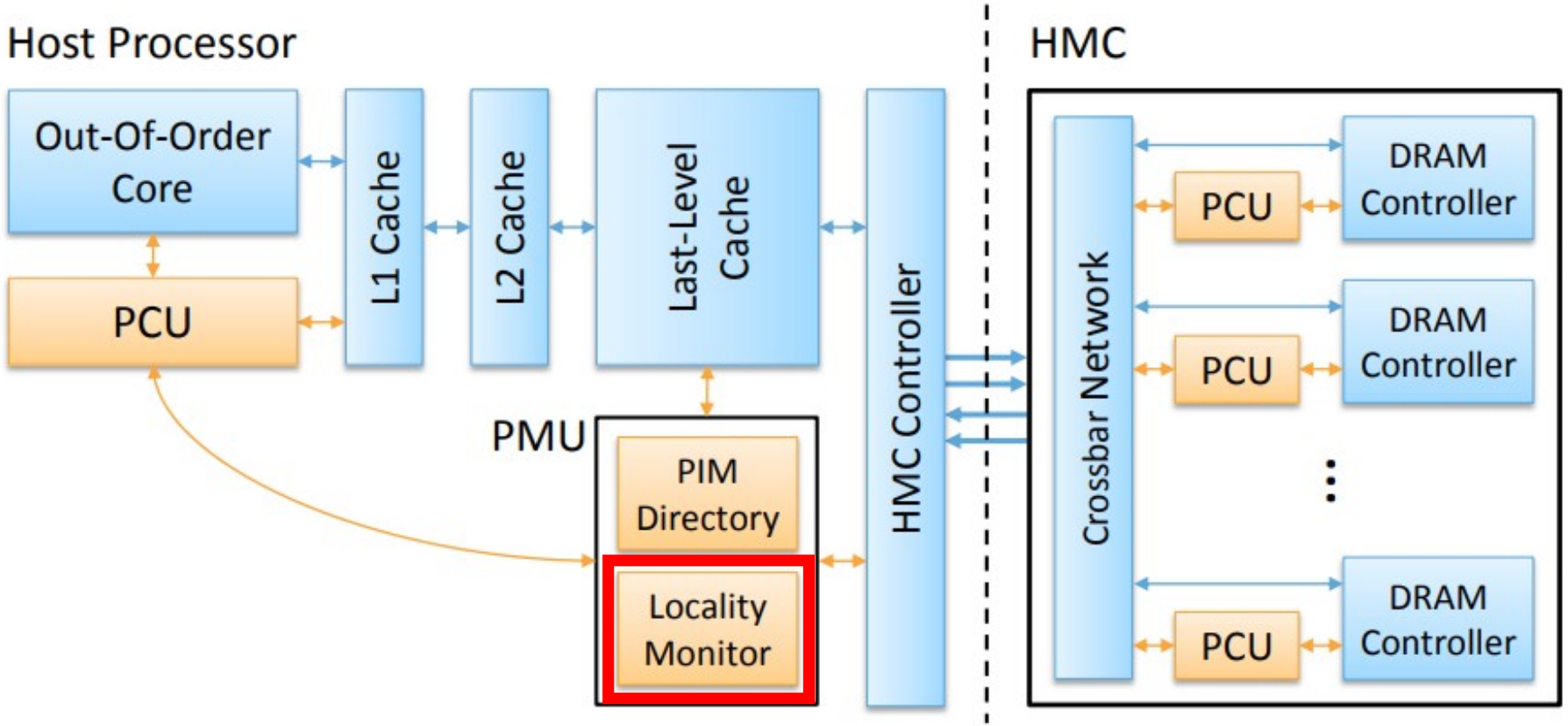
- Atomicity of memory-side execution can be achieved by simple modifications to the DRAM controllers
- To guarantee atomicity for host-side execution, a new hardware structure is introduced: the PIM directory

Hardware Architecture



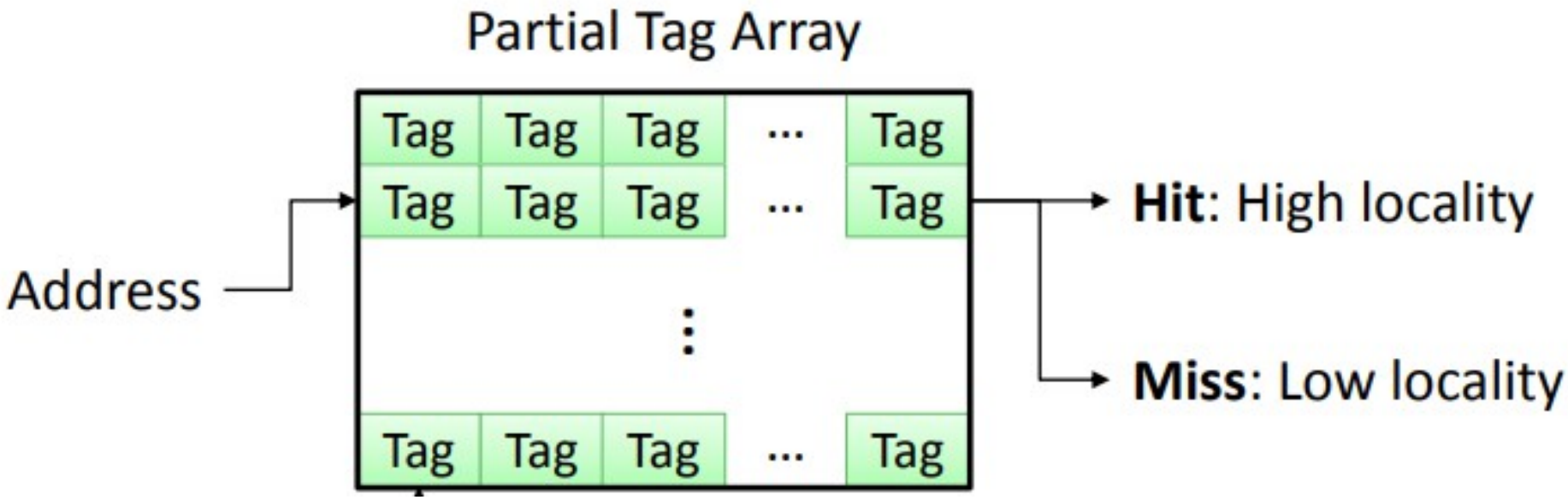
Proposed PEI Architecture

Hardware Architecture



Proposed PEI Architecture

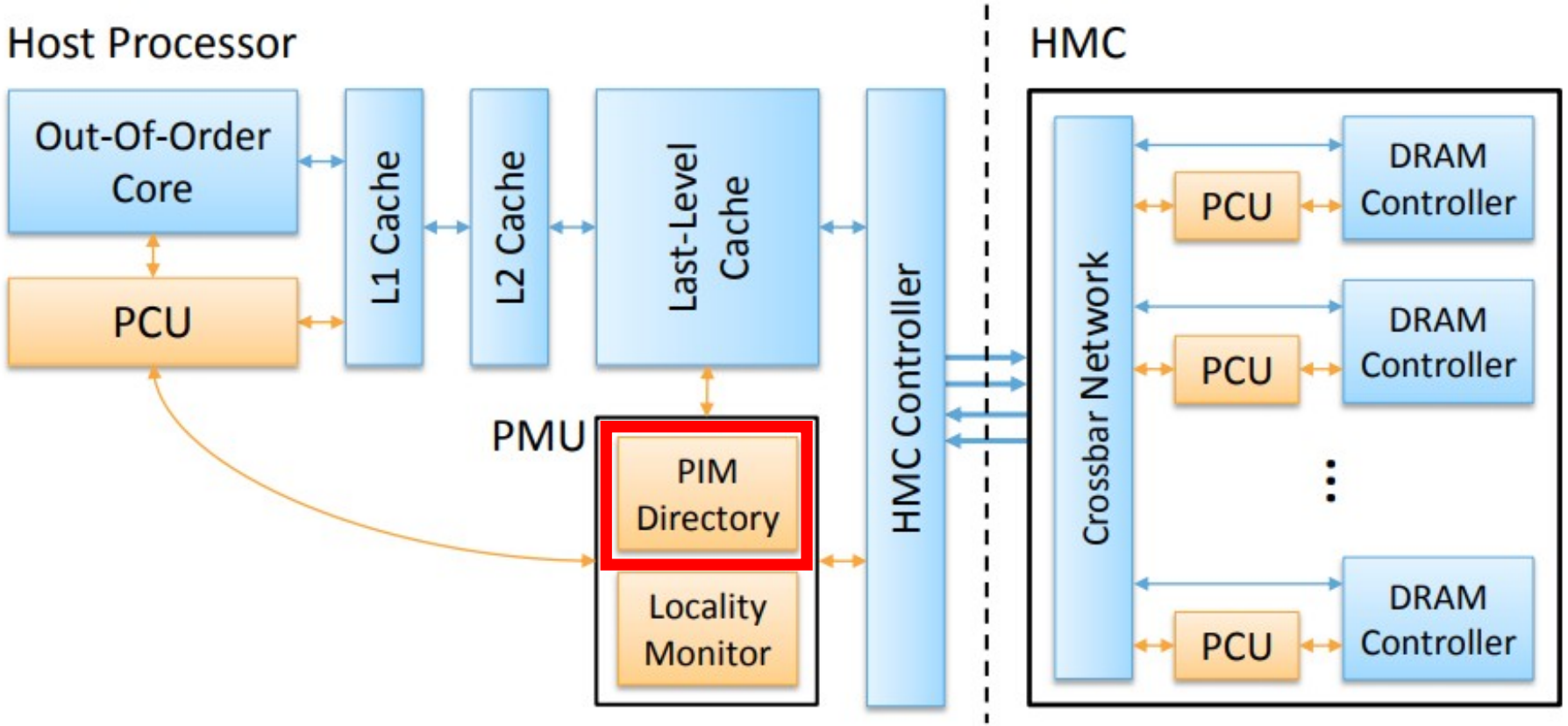
PMU: Locality Monitor



PMU: Locality Monitor

- Simple because of single cache block restriction
- Similar structure to last-level cache tag array
- Unless tag array updates as well when a PEI goes to memory as if it had accessed the corresponding last-level cache block in cache

Hardware Architecture

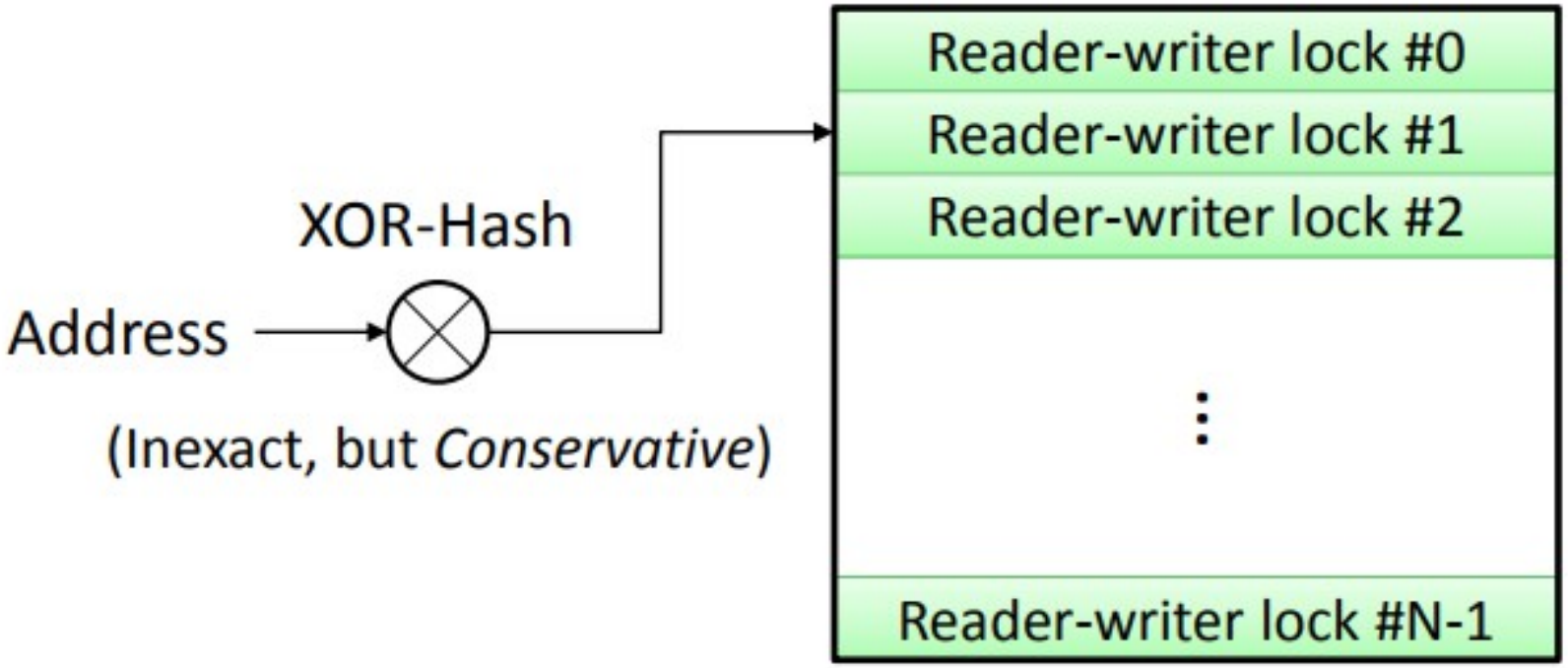


Proposed PEI Architecture

PMU: PIM directory

- Atomicity of memory-side execution can be achieved by simple modifications to the DRAM controllers
- To guarantee atomicity for host-side execution, a new hardware structure is introduced: the PIM directory
- Ideally, would track all in-flight PEI for reader/writer access safety
  - Very large overhead, requires fully associative table with as many entries as the added total entry number of all PCUs' operand buffers in the system
- To avoid this, rare false positives (unnecessary serialization) shall be allowed
  - Leads to overhead as well but paper claims it to be neglectable

PMU: PIM directory



PEI data access

- Always through PIM directory
- PIM directory maintains a writable bit, a readable bit and a reader counter

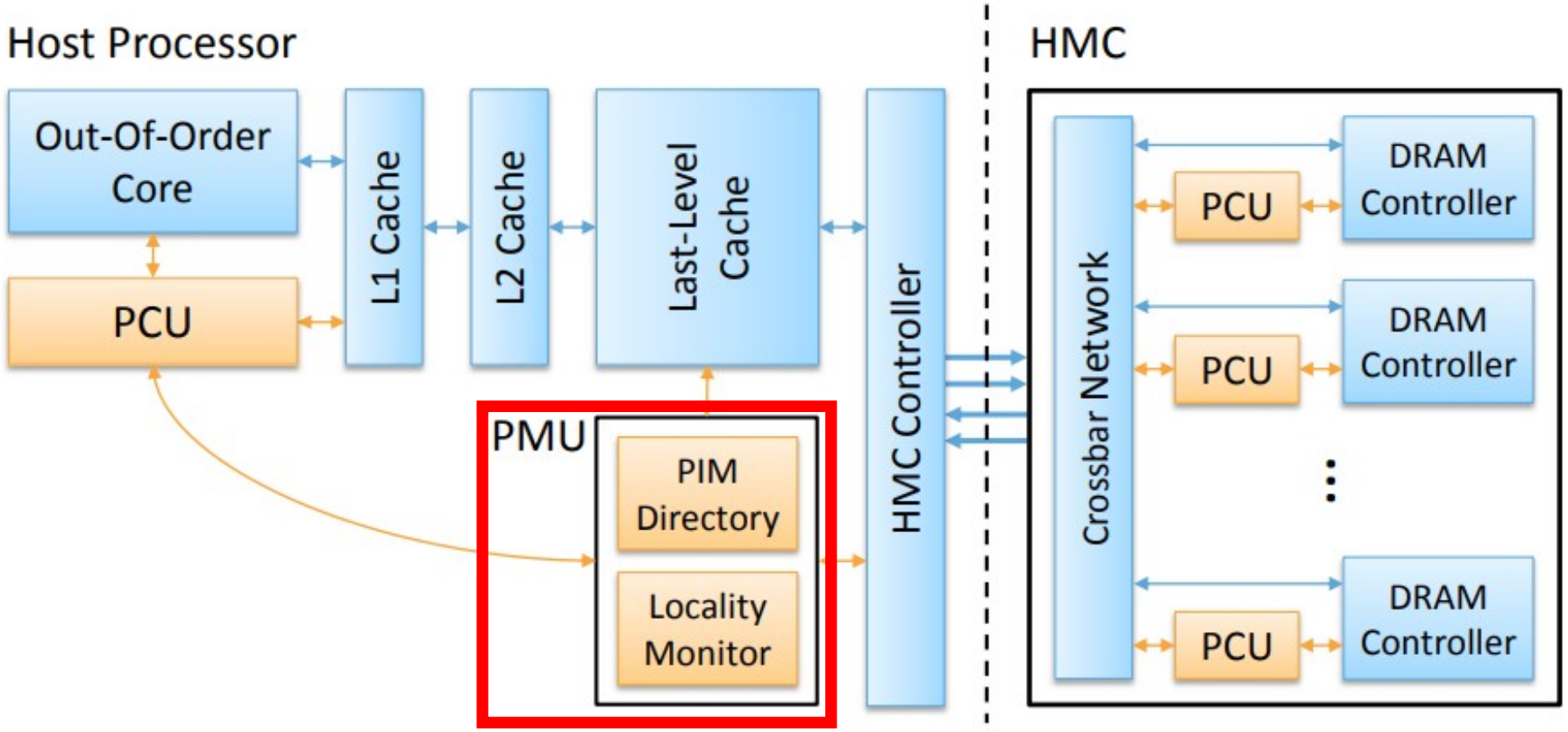
- Read PEI:
  - Wait until desired entry is readable
  - Mark entry as non-writable
  - Read entry
  - If no more read PEI to this entry are in-flight: mark as writable

PEI data access

- Always through PIM directory
- PIM directory maintains a writable bit, a readable bit and a reader counter

- Write PEI:
  - Wait until no other write PEI to the desired entry are in-flight
  - Mark entry as non-readable
  - Wait until desired entry is writable
  - Write entry
  - Mark as readable

Hardware Architecture



Proposed PEI Architecture

PMU: Cache coherence management

- Has to ensure that an old copy of data is never used
- PMU knows which cache block is accessed by a PEI it receives
  - Follows from single cache block restriction
- Therefore it can request back-invalidation or writeback for that cache block before any PIM operation
- Will rarely happen since PEI are only outsourced to memory if the target data is not expected to be present in on-chip caches

More on virtual memory

- Host processor always does address translation and hands over the corresponding physical address
- Thus PCU and PMU are only handling physical addresses
- No overhead from address translation capabilities in memory
- Page faults can be handled like usual by the host processor
- No more address translation needed for PEI than for any normal memory access operation

What kind of operation can be implemented as a PEI?

Operation	R	W	Input	Output	Applications
8-byte integer increment	O	O	0 bytes	0 bytes	AT
8-byte integer min	O	O	8 bytes	0 bytes	BFS, SP, WCC
Floating-point add	O	O	8 bytes	0 bytes	PR
Hash table probing	O	X	8 bytes	9 bytes	HJ
Histogram bin index	O	X	1 byte	16 bytes	HG, RP
Euclidean distance	O	X	64 bytes	4 bytes	SC
Dot product	O	X	32 bytes	8 bytes	SVM

O: operation modifies target cache block

Experimental Evaluation

Methodology

- 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)

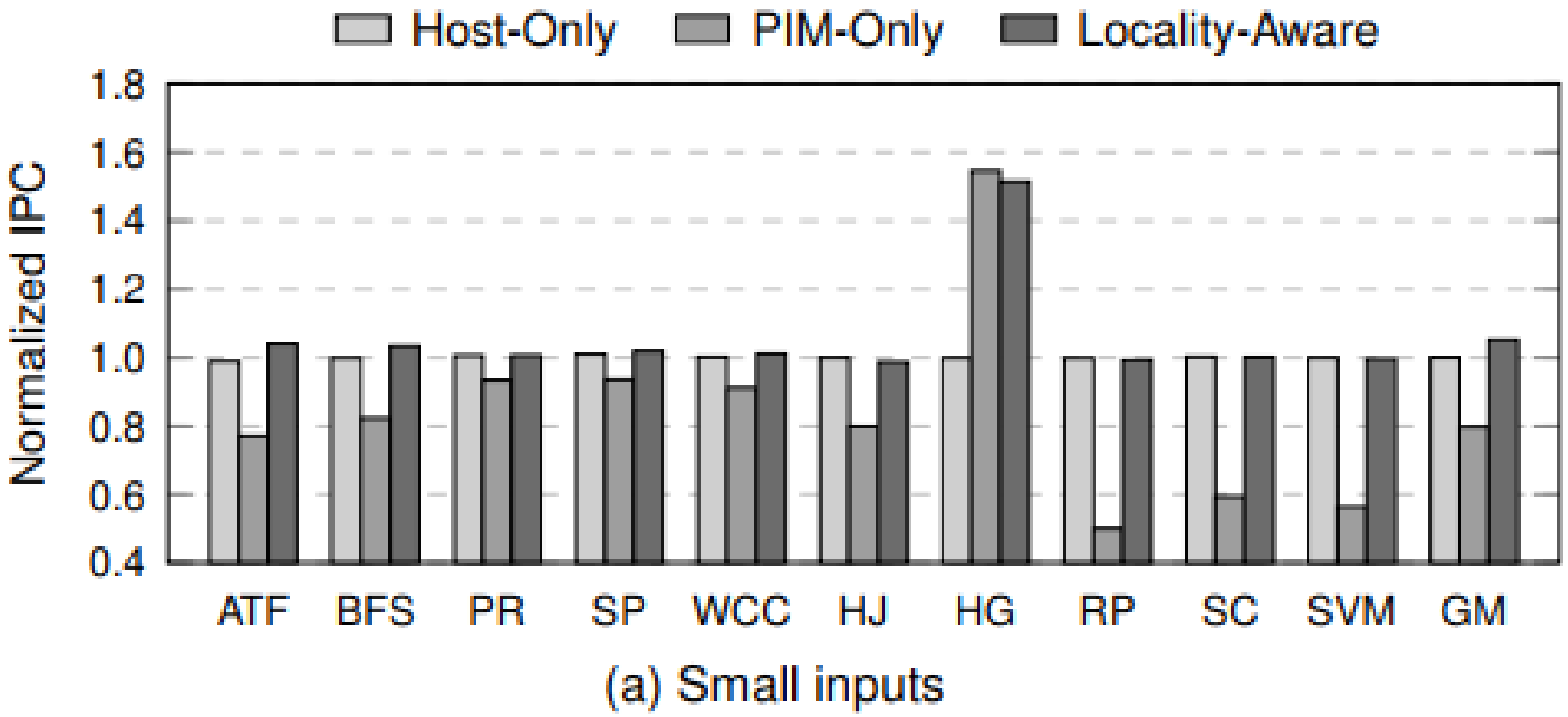
Applications

- 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

- What they all have in common: Data-intensive workloads

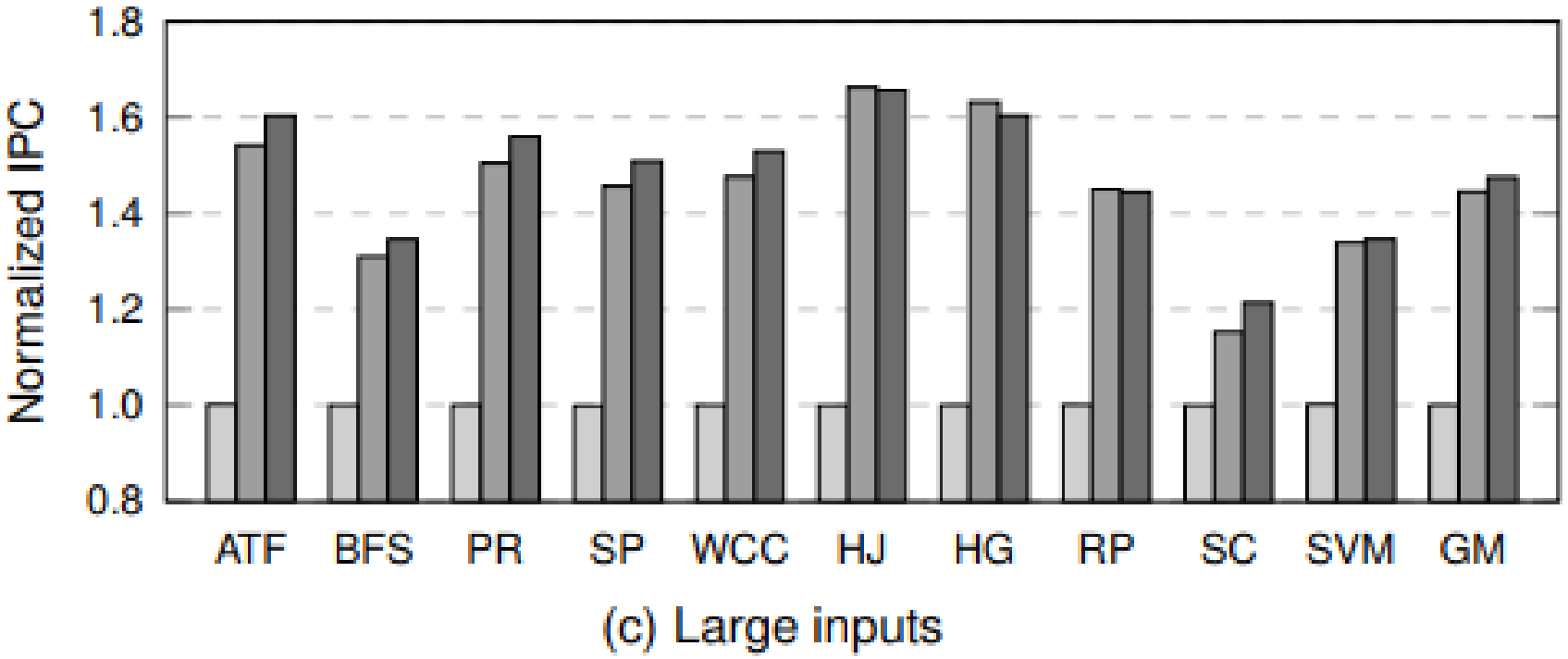
Results

Results: Performance



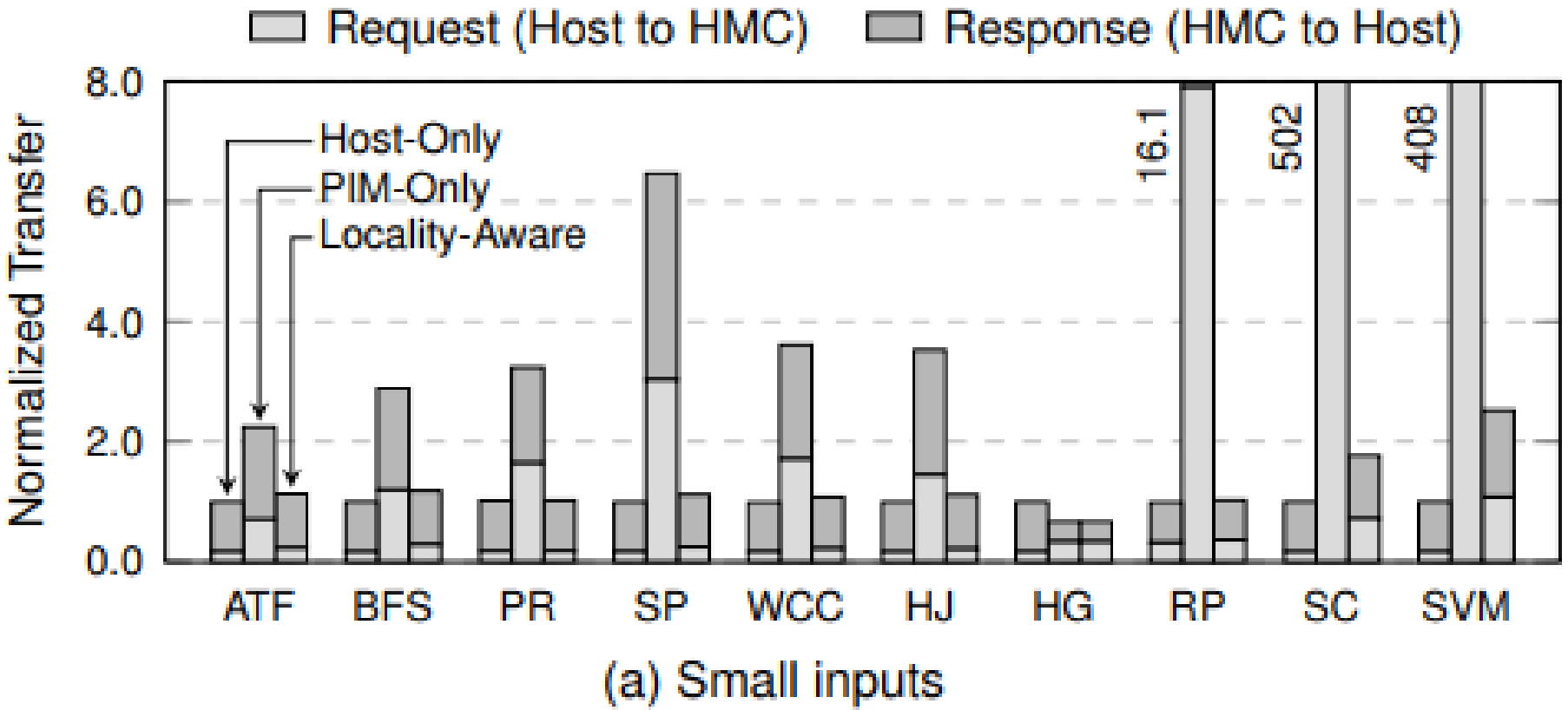
(a) Small inputs

Results: Performance

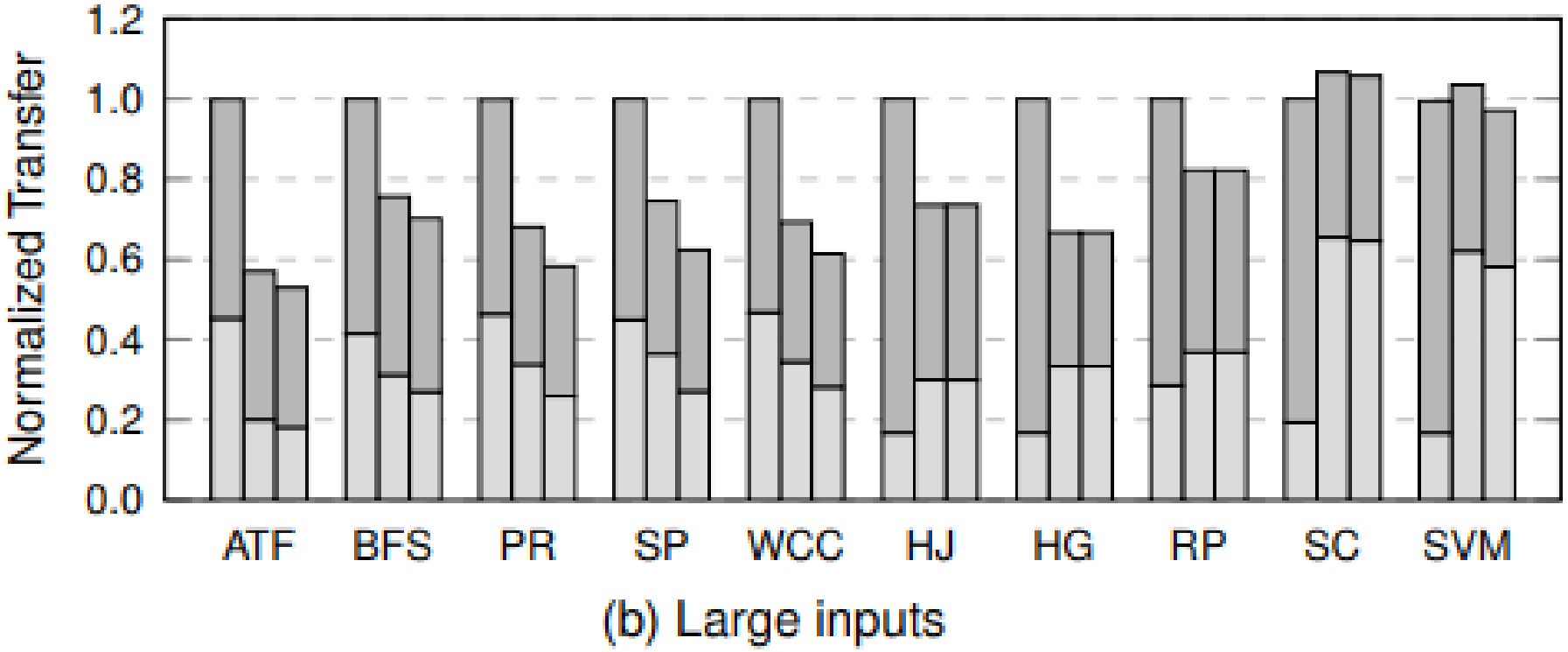


(c) Large inputs

Results: Off-chip transfers

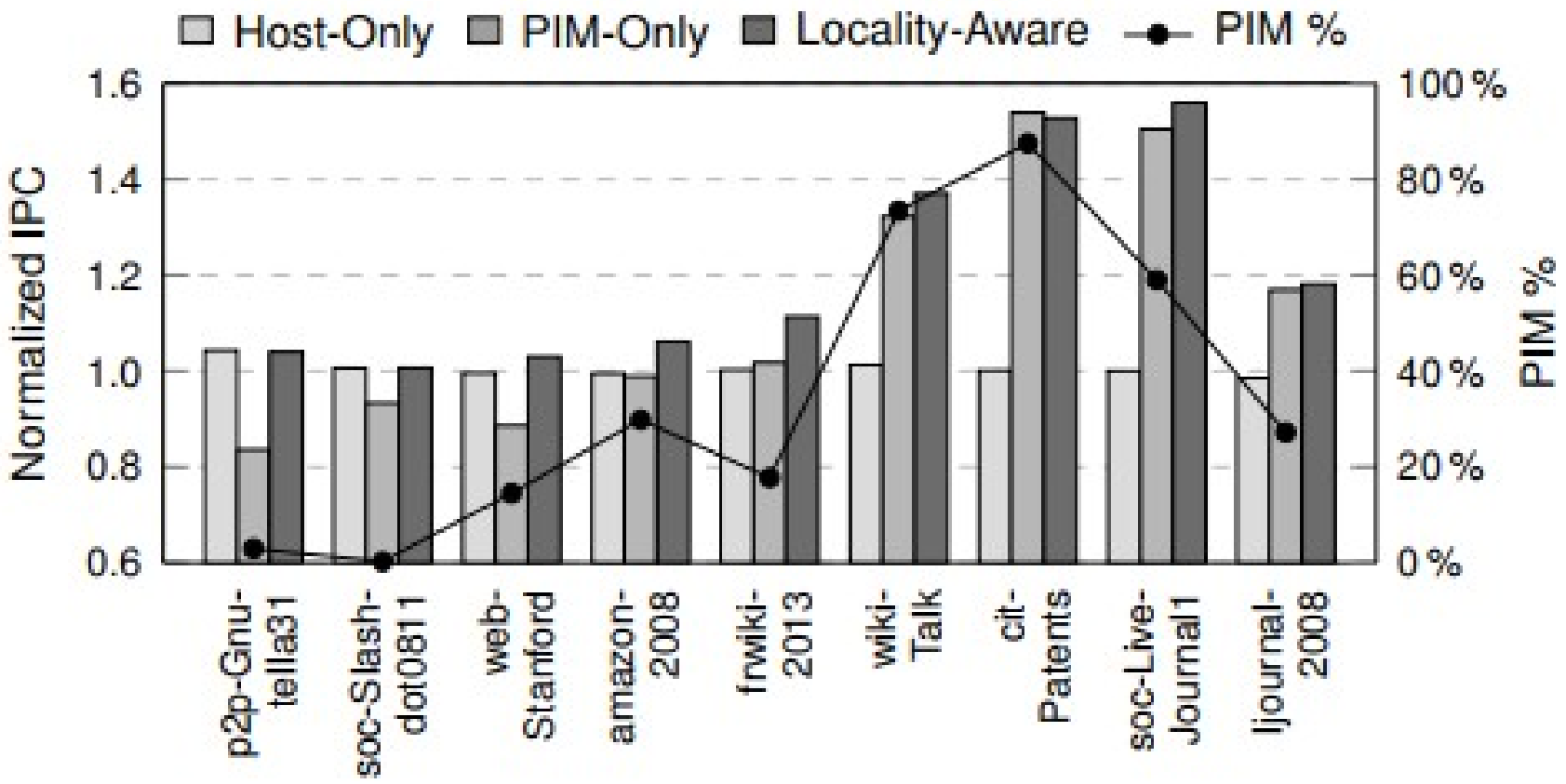


Results: Off-chip transfers



(b) Large inputs

Results: Page-rank with variable input size



Takeaways

Strengths

- Introduces a completely new idea for the application of processing in memory
- Doesn’t require as futuristic hardware as previous suggestions
- Not dependent on specific hardware architecture
- Native support for cache coherence
- Shows the possibly right direction to overcome current limitations of data-heavy computations

Weaknesses

- Ignores off-chip transfer of instructions
- Programs still have to be manually modified to take advantage of PIM
- The memory fence still adds a major change to the ISA
- Lack of compatibility
- Not enough details about simulation

Questions?

Thoughts and Discussion

- The paper provides an idea for the manual instruction replacement
- Instead of modifying source code, let compiler automatically find potential PEI and compile them accordingly
- The authors believe that their model is simple enough for a modern compiler to do this without complex program analysis
- However takes compilers out of too heavy responsibility since they don’t have to decide over PIM

- Do you think this is a viable solution?

Thoughts and Discussion

- Can you think of a way to avoid program changes?
- Can you make it fully compatible?

- New idea: Move decisions even further away from programmer
- Do not change ISA at all
- Decide on runtime which instructions could possibly be executed as PEI
- Then decide over PIM as proposed in the paper

- Do you think this would work?

Discussion

- How does the PEI model perform in comparison to GPGPUs? For which workloads?
- The single cache block restriction gives a very strong restriction to what can be performed in memory. Can’t this lack of generality become an issue e.g. for data-heavy single operations?
- Can’t the direct physical memory access by all PIM/PEI units and instructions cause a safety issue?
- What do you think about the overlapping pipeline stages problem? Is stalling a good solution? Can you think of a different one?

Related Papers for Discussion

GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks

Lifeng Nai[†], Ramyad Hadidi[†], Jaewoong Sim[*], Hyojong Kim[†], Pranith Kumar[†], Hyesoon Kim[†]

Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM

Vivek Seshadri[1], Donghyuk Lee[2], Thomas Mullins[3], Hasan Hassan[4], Amirali Boroumand[5], Jeremie Kim[5], Michael A. Kozuch[6], Onur Mutlu[7], Phillip B. Gibbons[5], Todd C. Mowry[5]

Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories

Shuangchen Li[1]; Cong Xu[2], Qiaosha Zou[1,5], Jishen Zhao[3], Yu Lu[4], and Yuan Xie[1]

A Case for Near Memory Computation Inside the Smart Memory Cube

Erfan Azarkhish,
Davide Rossi, and Igor Loi
DEI, University of Bologna, Bologna, Italy
Emails: erfan.azarkhish@unibo.it,
davide.rossi@unibo.it, and igor.loi@unibo.it

Luca Benini
ITET, Swiss Federal Institute of Technology,
Zurich, Switzerland,
DEI, University of Bologna, Bologna, Italy
Email: lbenini@iis.ee.ethz.ch

More interesting papers for discussion support

- Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions, Saugata Ghose et al.
- Performance Implications of Processing-in-Memory Designs on Data-Intensive Applications, Borui Wang et al.
- LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory, Amirali Boroumand et al.
- Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation, Kevin Hsieh et al.
- CoolPIM: Thermal-Aware Source Throttling for Efficient PIM Instruction Offloading, Lifeng Nai et al.