PEI: Processing-in-Memory-Enabled Instruction

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

Seoul National University [+]Carnegie Mellon University

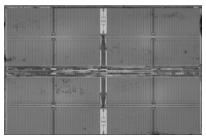
Processing-in-Memory (PIM)

- Move computations 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*

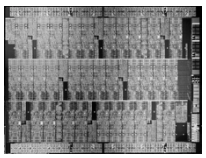
Why was PIM unsuccessful in its first attempt?

Challenges in PIM

Cost-effectiveness



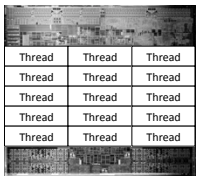
DRAM die

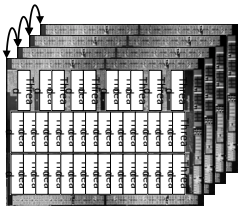


Complex Logic

Programming Model

Host Processor

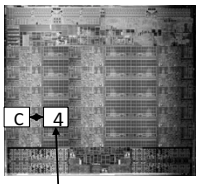


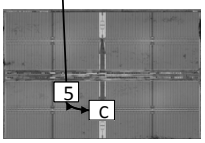


In-Memory Processors

Coherence & VM

Host Processor

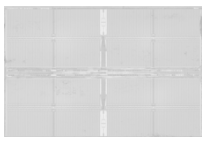




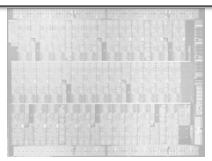
DRAM die

Challenges in PIM

Cost-effectiveness



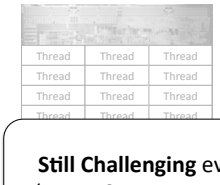
(Partially) Solved by 3D-Stacked DRAM



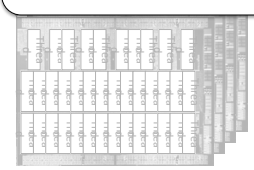
Complex Logic

Programming Model

Host Processor



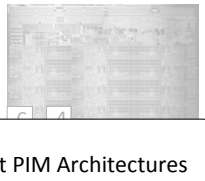
Still Challenging even in Recent PIM Architectures (e.g., AC-DIMM, NDA, NDC, TOP-PIM, Tesseract, …)

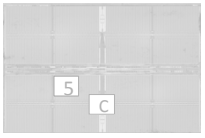


In-Memory Processors

Coherence & VM

Host Processor





DRAM die

A New Direction of PIM

- Objectives
  - Reduce the implementation overhead of PIM units
  - Provide an intuitive programming model for PIM
  - Full support for cache coherence and virtual memory
- Our solution: simple PIM operations as ISA extension
  - Simple: low-overhead implementation
  - ISA extension: intuitive programming model

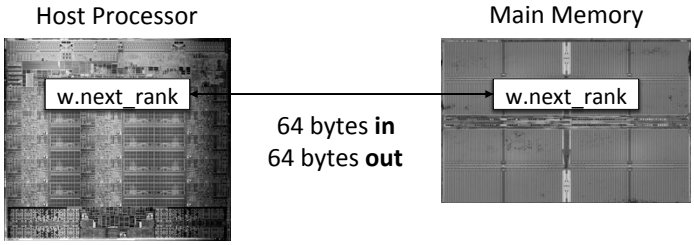
Potential of ISA Extension as the PIM Interface

- Example: Parallel PageRank (PR) computation

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

Potential of ISA Extension as the PIM Interface

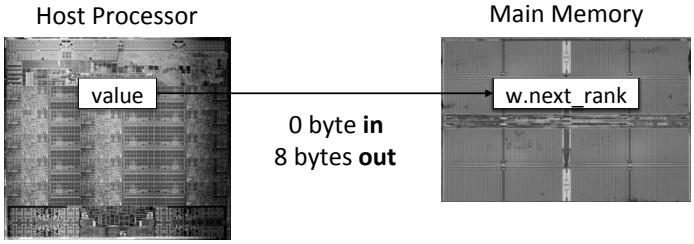
```
for (v: graph.vertices) {
  for (w: v.successors) {
    w.next_rank += weight * v.rank;
  }
}
```



Conventional Architecture

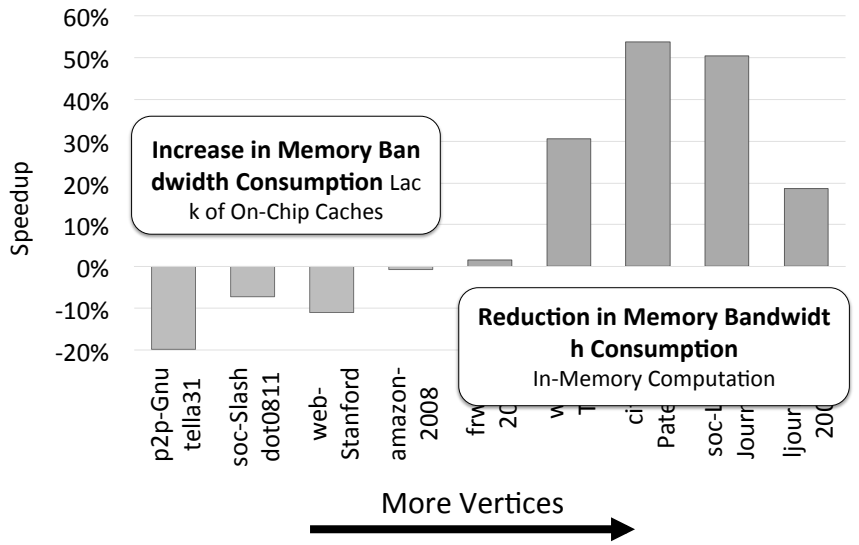
Potential of ISA Extension as the PIM Interface

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



In-Memory Addition

Potential of ISA Extension as the PIM Interface



Overview

1. How should simple PIM operations be interfaced to conventional systems?
   - Expose PIM operations as *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) {
   for (w: v.successors) {
      w.next_rank += weight * v.rank;
   }
}
```

PIM-Enabled Instructions

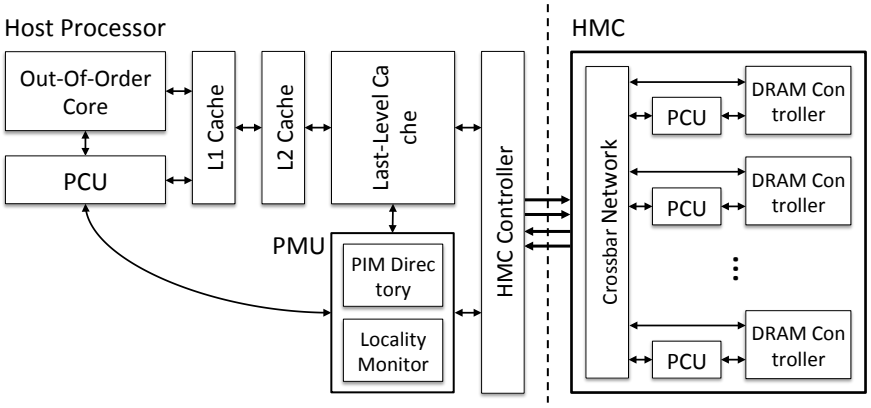
```
for (v: graph.vertices) {
   for (w: v.successors) {
      PIM_add(&w.next_rank, weight * v.rank);
   }
}
pfence();
```

- 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

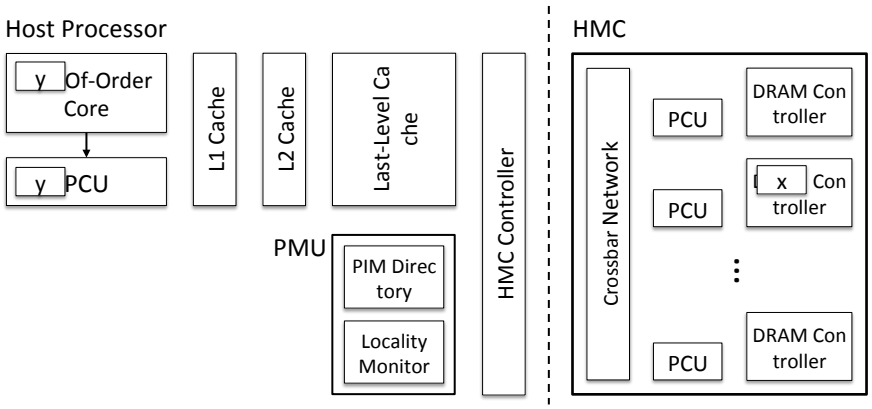
- Single-cache-block restriction
  - Each PEI can access *at most one last-level cache block*
  - Similar restrictions exist in atomic instructions
- Benefits
  - **Localization**: each PEI is bounded to one memory module
  - **Interoperability**: easier support for cache coherence and v irtual memory
  - **Simplified locality monitoring**: data locality of PEIs can be i dentified by LLC tag checks or similar methods

Architecture



Proposed PEI Architecture

Memory-side PEI Execution

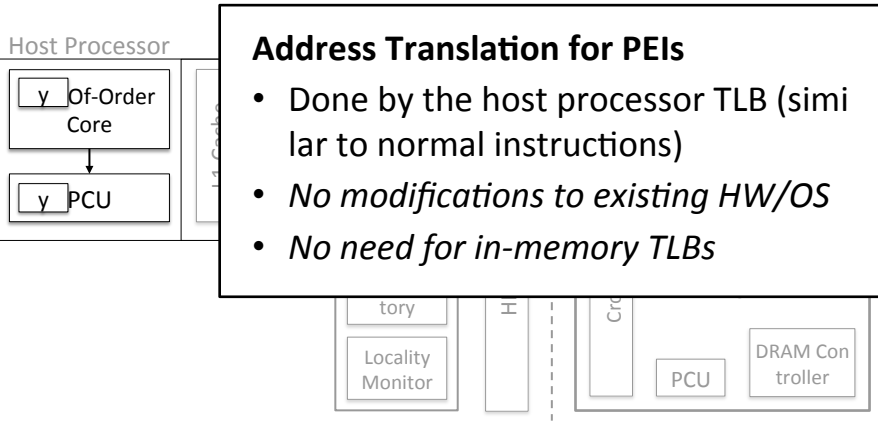


PIM_add(&x, y)

Memory-side PEI Execution

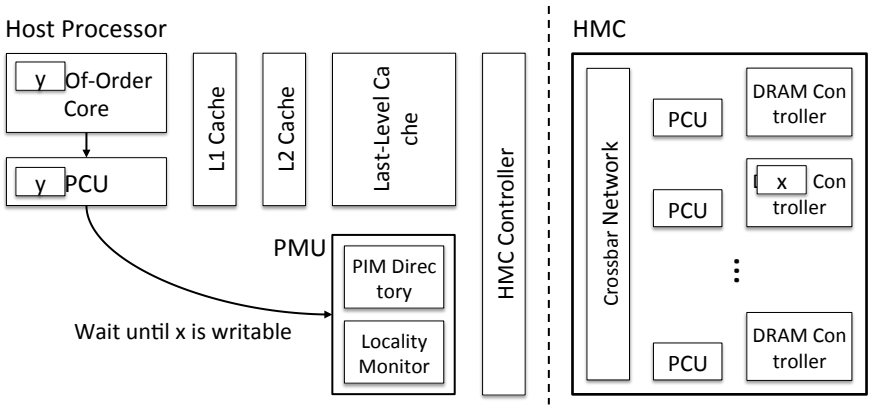
Address Translation for PEIs

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

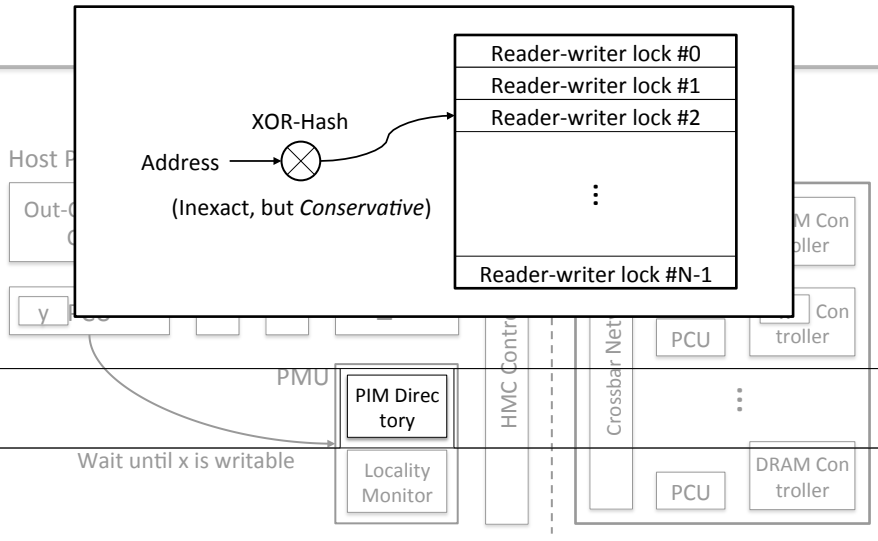


PIM_add(&x, y)

Memory-side PEI Execution

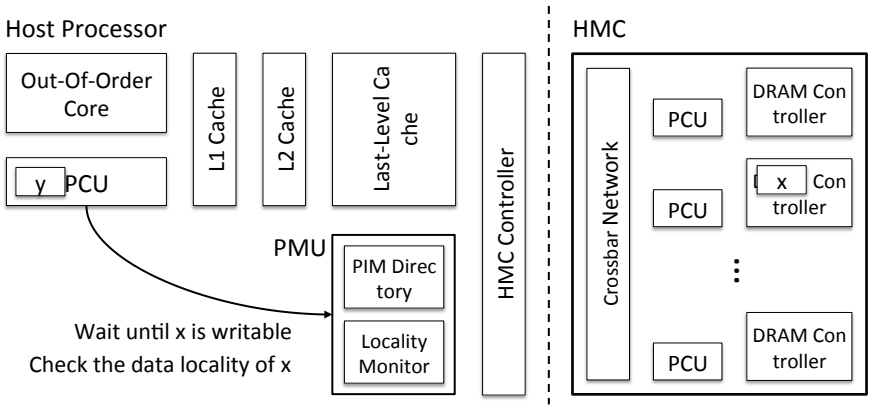


PIM_add(&x, y)



PIM_add(&x, y)

Memory-side PEI Execution

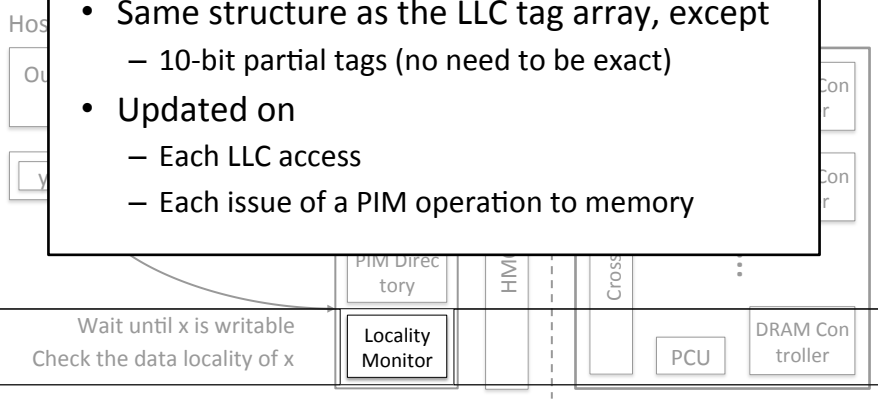


PIM_add(&x, y)

Memory-side PEI Execution

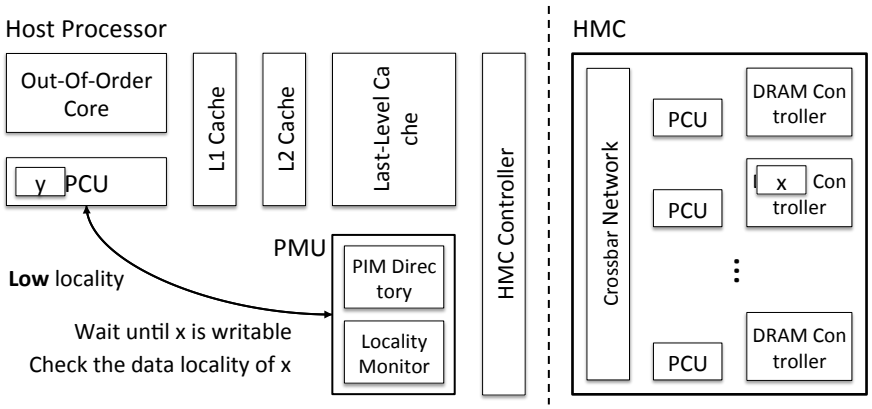
Locality Check for Data Accessed by PEIs

- Same structure as the LLC tag array, except
  - 10-bit partial tags (no need to be exact)
- Updated on
  - Each LLC access
  - Each issue of a PIM operation to memory



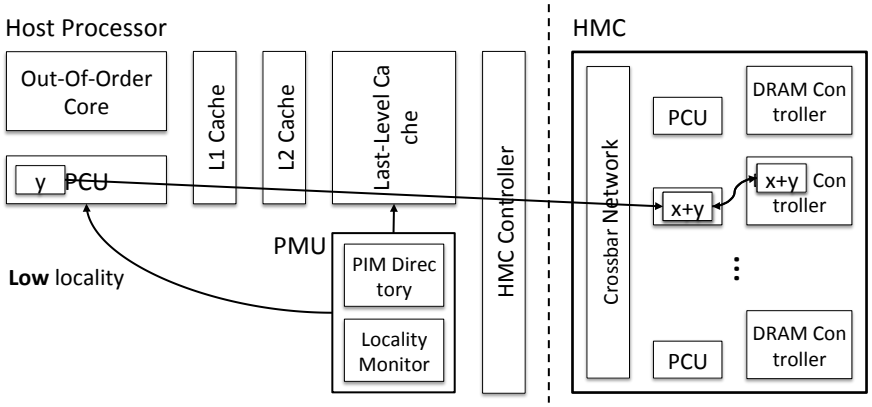
PIM_add(&x, y)

Memory-side PEI Execution



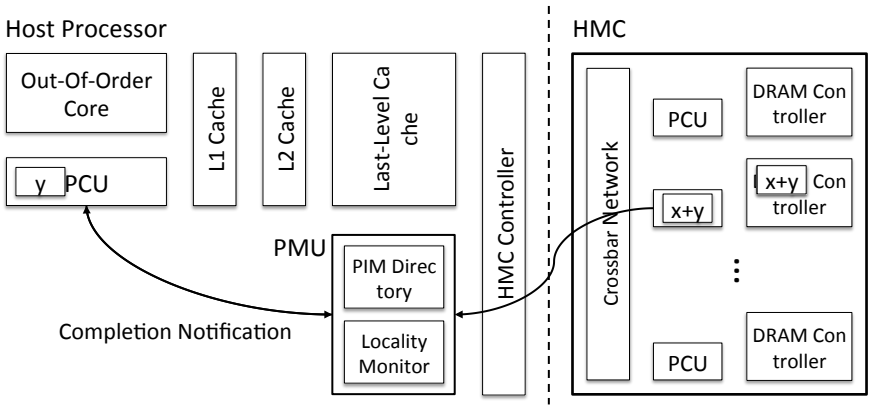
PIM_add(&x, y)

Memory-side PEI Execution



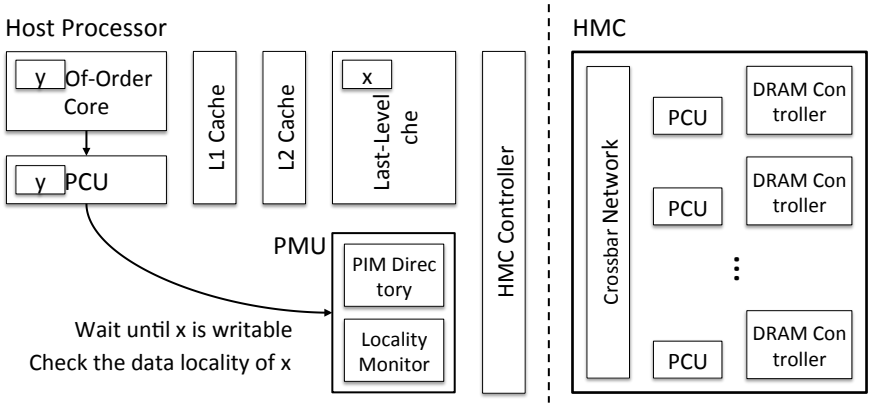
PIM_add(&x, y)

Memory-side PEI Execution



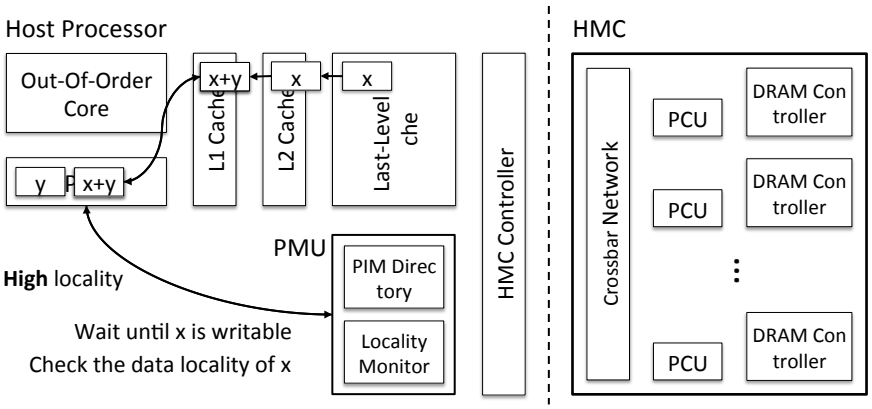
PIM_add(&x, y)

Host-side PEI Execution



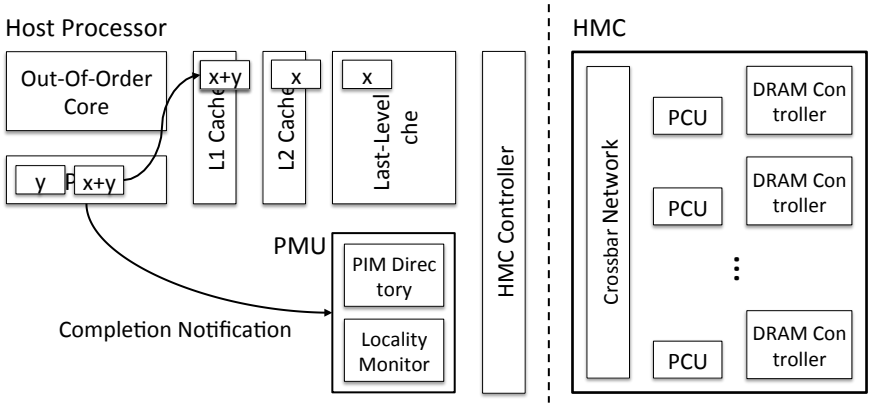
PIM_add(&x, y)

Host-side PEI Execution



PIM_add(&x, y)

Host-side PEI Execution



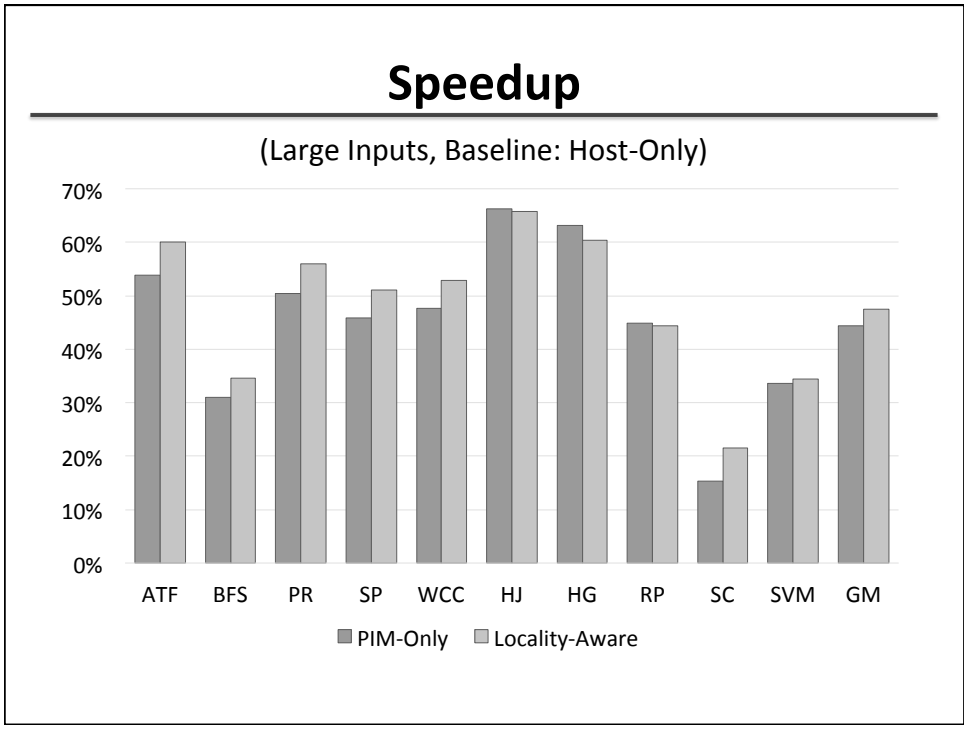
PIM_add(&x, y)

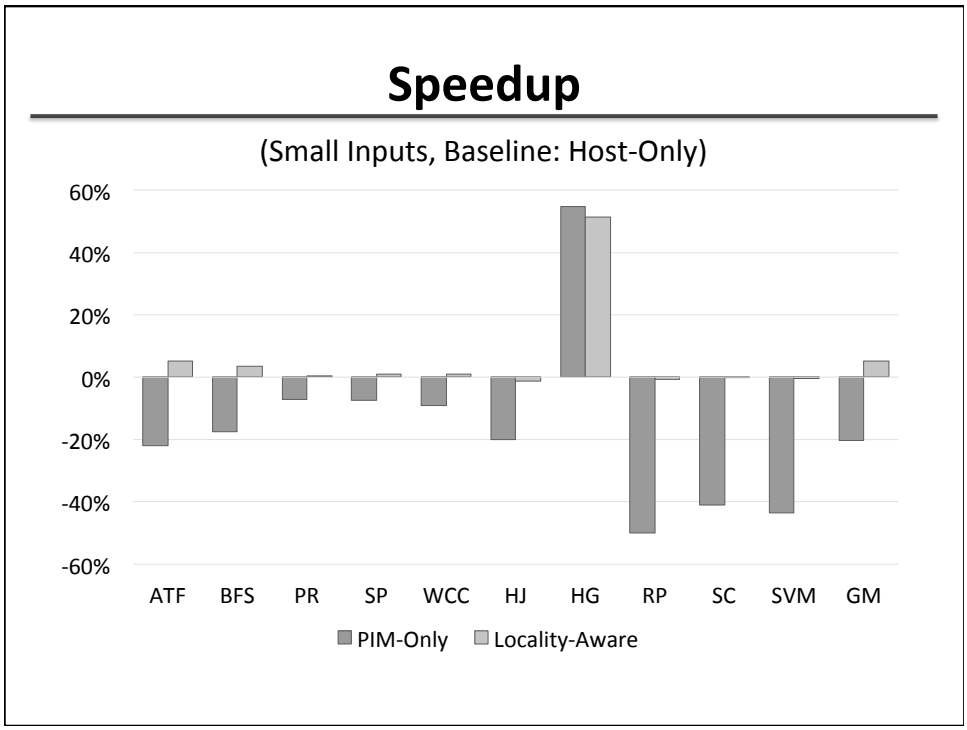
Simulation Configuration

- 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

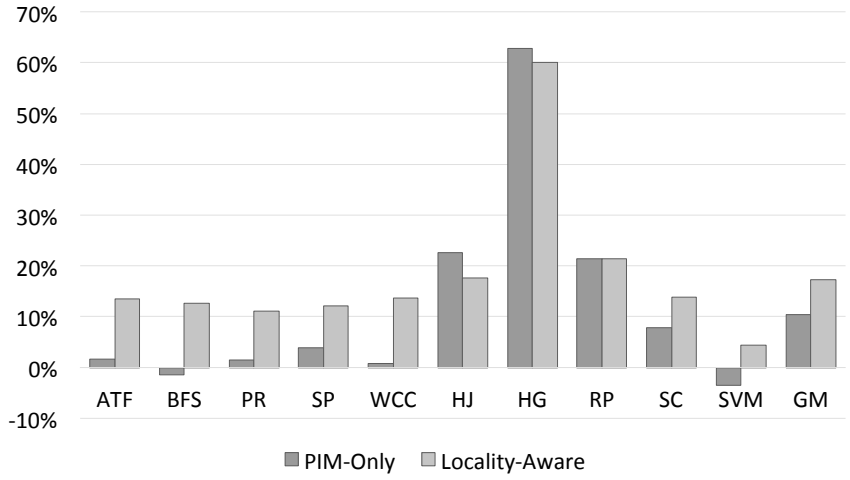
- Ten emerging data-intensive workloads
  - Large-scale graph processing
    - Average teenage followers, BFS, PageRank, single-source shortest path, weakly connected components
  - In-memory data analytics
    - Hash join, histogram, radix partitioning
  - Machine learning and data mining
    - Streamcluster, SVM-RFE
- Three input sets (small, medium, large) for each wor kload 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

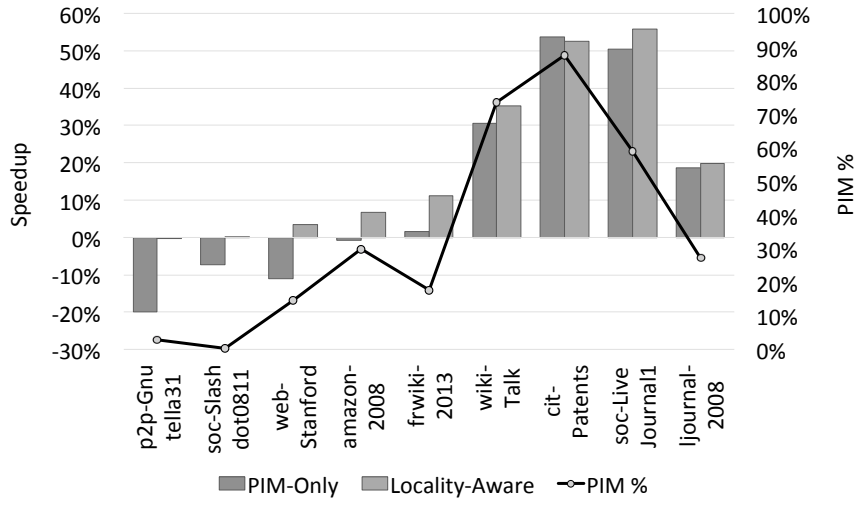

Speedup
(Small Inputs, Baseline: Host-Only)
60%
40%
20%
0%
-20%
-40%
-60%
ATF
BFS
PR
SP
WCC
HJ
HG
RP
SC
SVM
GM
PIM-Only
Locality-Aware

Speedup

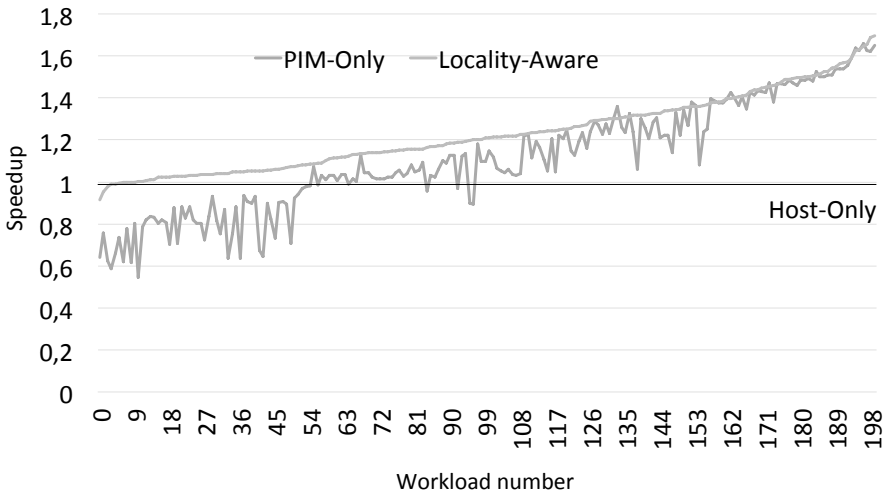
(Medium Inputs, Baseline: Host-Only)



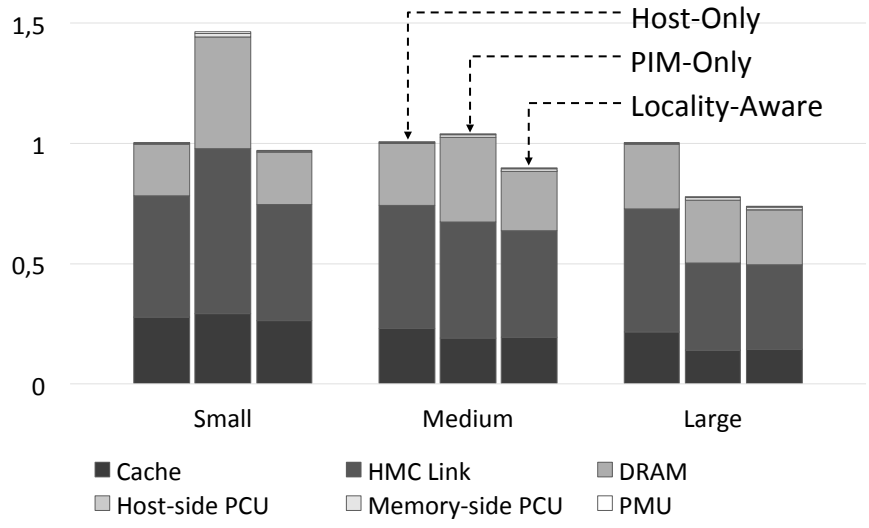
Sensitivity to Input Size (PR)



Multiprogrammed Workloads



Energy Consumption



Conclusion

- Challenges of PIM architecture design
  - Costly integration of logic and memory
  - Unconventional programming models
  - Lack of interoperability with caches and virtual memory
- PIM-enabled instruction
  - low-cost PIM implementation
  - 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