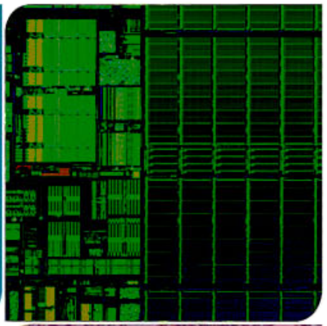
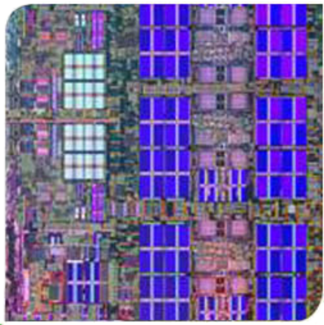
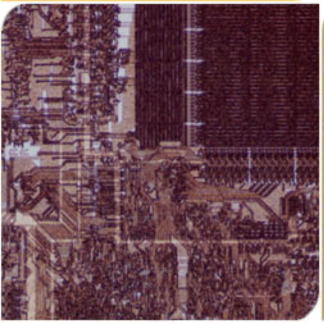



GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks

Lifeng Nai, Ramyad Hadidi, Jaewoong Sim*,
Hyojong Kim, Pranith Kumar, Hyesoon Kim
Georgia Tech, *Intel Labs



INTRODUCTION

Graph computing: processing big network data

- Social network, knowledge network, bioinformatics, etc.

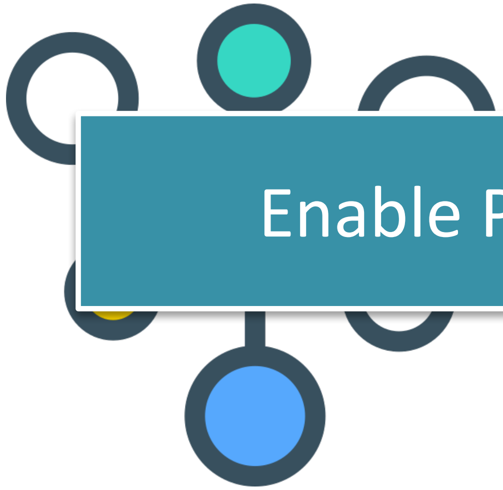
Graph computing is inefficient on conventional architectures

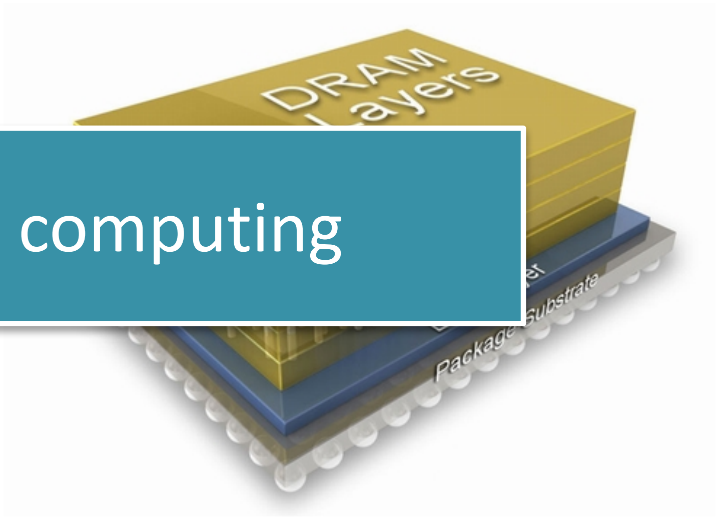
- Inefficiency in memory subsystems

INTRODUCTION

Processing-in-memory (PIM)

- PIM has the potential of helping graph computing performance
- PIM is being realized in real products: Hybrid memory cube (HMC) 2.0





Enable PIM for graph computing

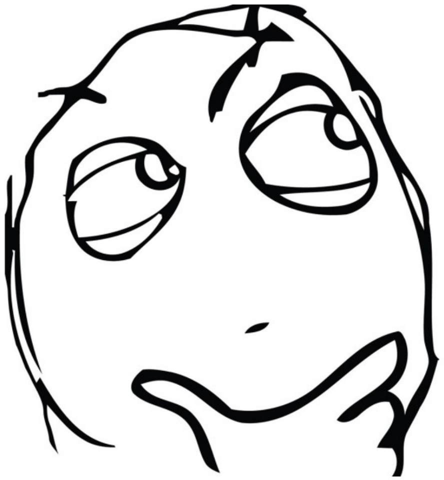
CHALLENGES

What are the benefits of PIM for graph computing?

- Known benefits of PIM
  - Bandwidth savings, latency reduction, more computation power
- But, they are not good enough
- **We explore something more!**

How to enable PIM for graph in a practical way?

- Minor hardware/software change
- No programmer burden



OVERVIEW

GraphPIM: a **PIM-enabled** graph **framework**

We identify a new benefit of PIM offloading

We determine PIM offloading targets

We enable PIM without user-application change

KNOWN PIM BENEFITS

More computation power

- Extra computation units in memory

Bandwidth savings

- Pulling data vs. pushing computation command

	Request	Response	Total
64-byte READ	1 FLIT (addr)	5 FLITs (data)	6 FLITs
64-byte WRITE	5 FLITs (addr, data)	1 FLIT (ack)	6 FLITs
CPU rd-modify-wr (rd-Miss; wr-evict)	6 FLIT	6 FLIT	12 FLITs ↓
PIM rd-modify-wr	2 FLITs (addr, imm)	1 FLIT (ack)	3 FLITs

(FLIT: 16 byte, basic flow unit)

PERFORMANCE BENEFITS

More computation power?

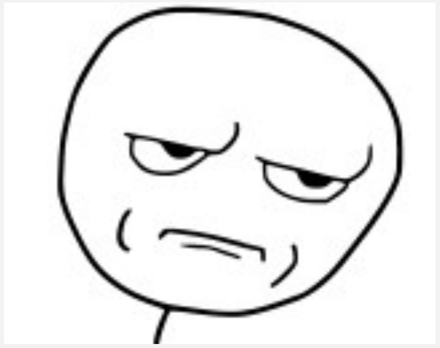
- Limited # of FUs in memory

Bandwidth savings?

- Not BW saturated

Latency reduction?

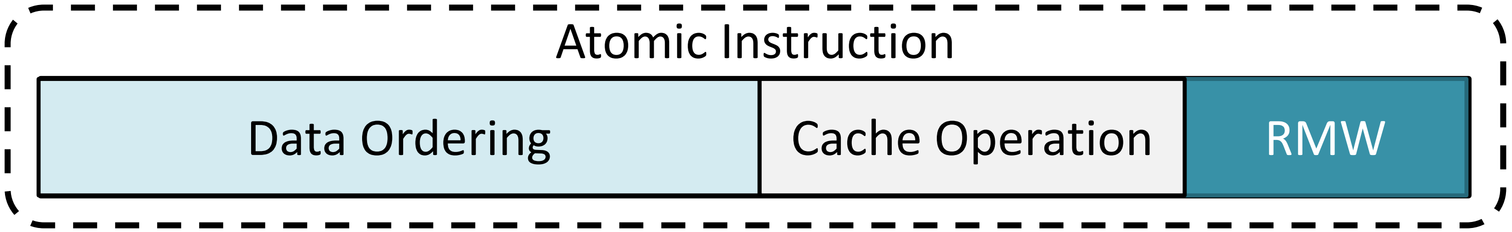
- Yes, but small



GRAPHPIM EXPLORES…

Atomic overhead reduction

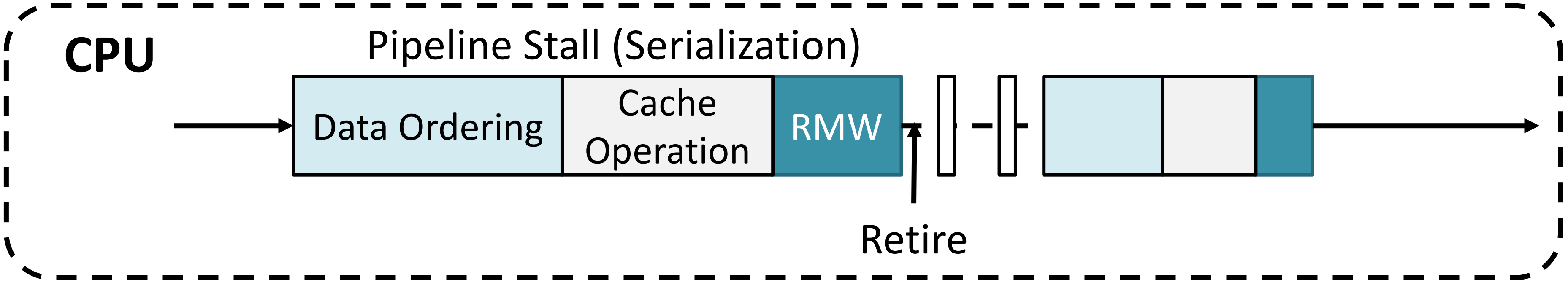
- Atomic instructions on CPUs have substantial overhead [Schweizer'15]

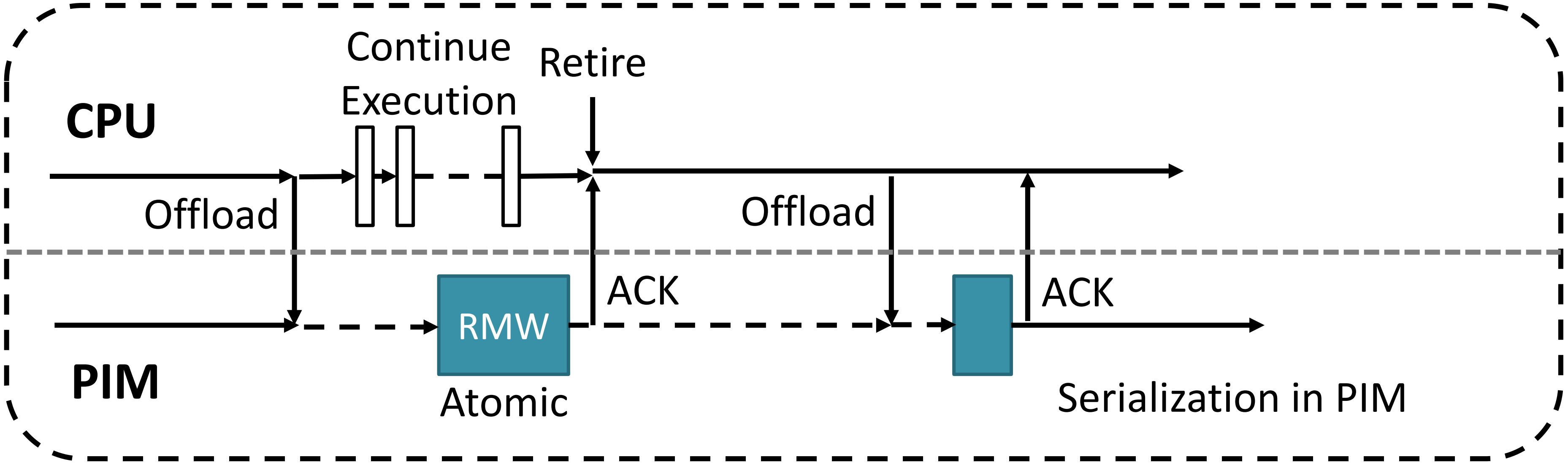


  - RMW: read-modify-write
  - Cache operation: cache-line invalidation, coherence traffic etc.
  - Data ordering: write buffer draining, pipeline freeze etc.
- Because of the characteristics of graph programming model, PIM offloading can avoid the atomic overhead

H. Schweizer et al., "Evaluating the Cost of Atomic Operations on Modern Architectures," PACT'15

ATOMIC OVERHEAD REDUCTION



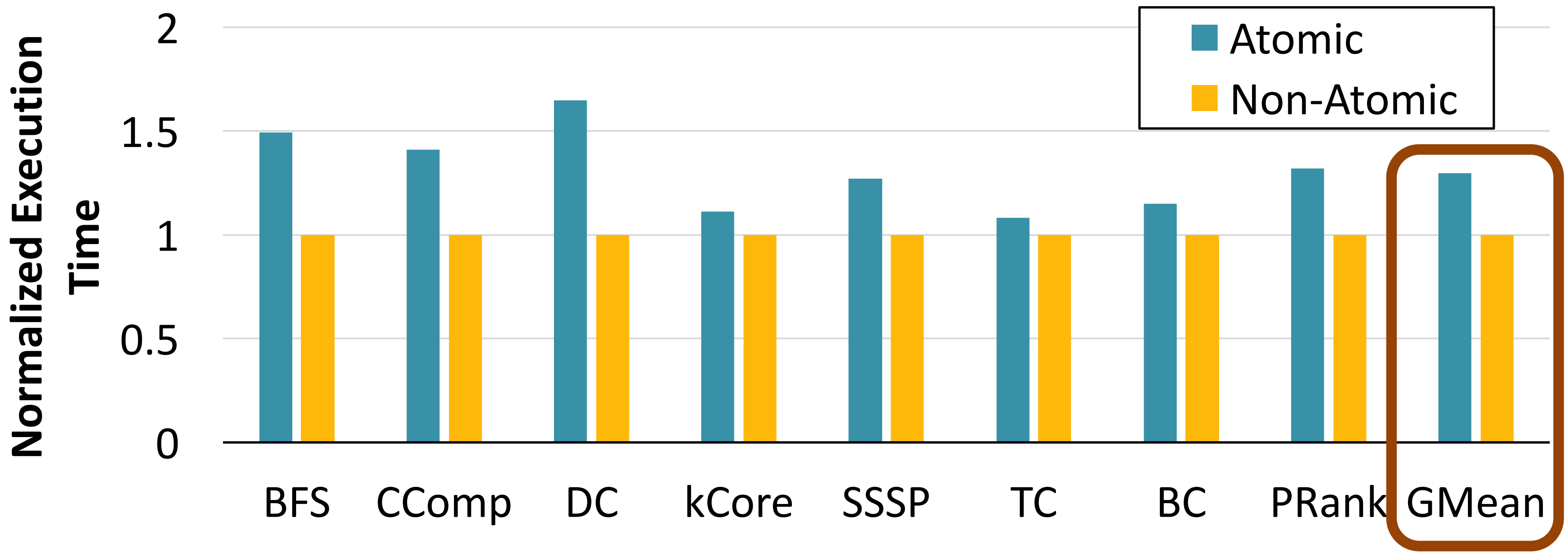


ATOMIC OVERHEAD ESTIMATION

Atomic overhead experiments on a Xeon E5 machine

- Atomic RMW → regular load + compute + store

Atomic instructions incur 30% performance degradation



(Non-Atomic: artificial experiment, not precise estimation)

PERFORMANCE BENEFITS

More computation power?

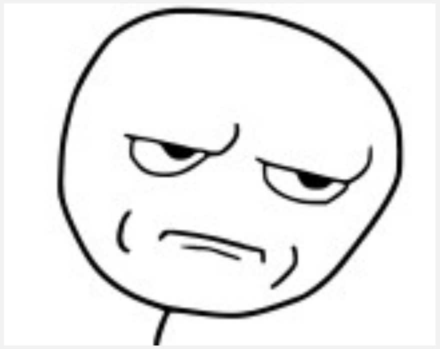
- Limited # of FUs in memory

Bandwidth savings?

- Not BW saturated

Latency reduction?

- Yes, but small



Atomic overhead reduction?

- Yes and significant!
- Main source of PIM benefit for graph

OFFLOADING TARGETS

Code snippet: Breadth-first search (BFS)

```
1   F ← {source}
2   while F is not empty
3       F′ ← {∅}
4       for each u ∈ F in parallel
5           d ← u.depth + 1
6           for each v ∈ neighbor(u)
7               ret←CAS(v.depth, inf, d)
11          endfor
15  endwhile
```

F: frontier vertex set of current step
F′: frontier vertex set of next step
u.depth: depth value of vertex u
neighbor(u): neighbor vertices of u
CAS(v.depth, inf, d): atomic compare and swap operation

line 4-5, 8-10: accessing **meta data**

line 6: accessing **graph structure**

Cache Unfriendly + Atomic

Cache Friendly

Offload **atomic** operations on graph **property**

INDICATE OFFLOADING TARGETS

How to indicate offloading targets?

Option #1: Mark instructions

- New instructions in ISA
- Requires changes in user-level applications

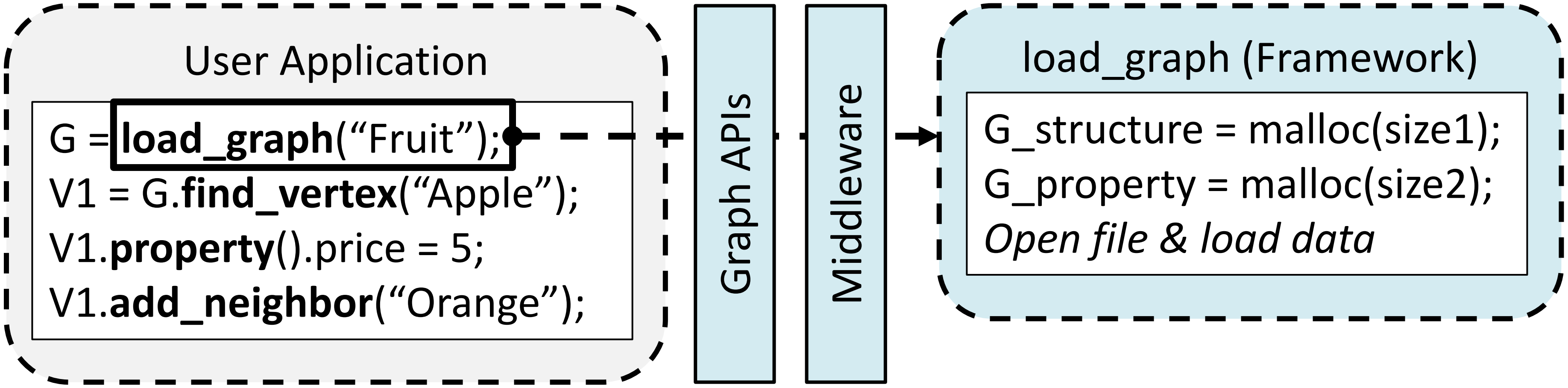
Option #2: Mark memory regions

- Special memory region for offloading data
- Can be transparent to application programmers

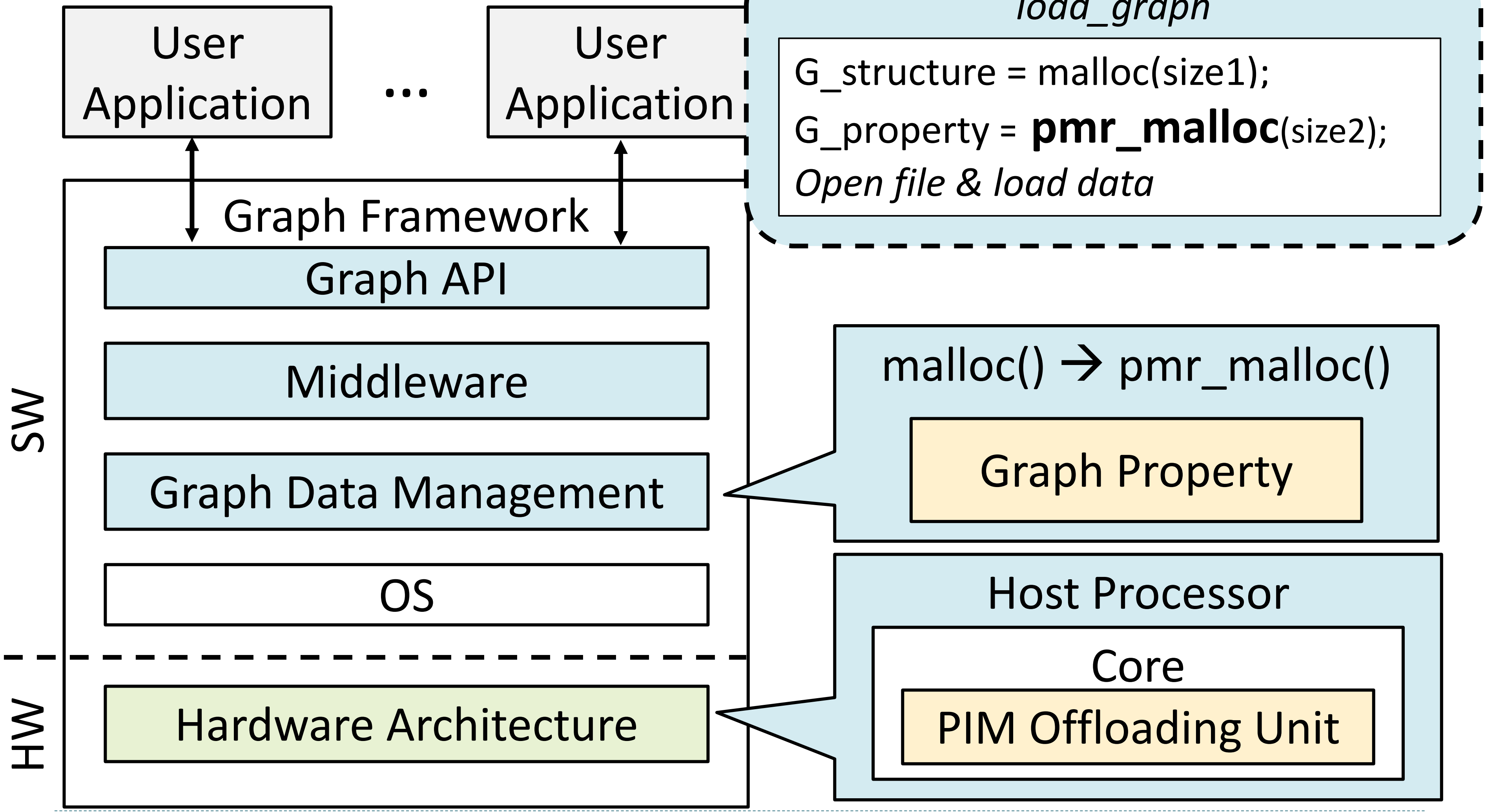
GRAPH FRAMEWORK

Graph computing is **framework**-based

- User application is designed on top of framework interfaces
- Data is managed within the framework


User Application
G = **load_graph**("Fruit");
V1 = G.**find_vertex**("Apple");
V1.**property**().price = 5;
V1.**add_neighbor**("Orange");
Graph APIs
Middleware
load_graph (Framework)
G_structure = malloc(size1);
G_property = malloc(size2);
Open file & load data

ENABLE PIM IN GRAPH FRAMEWORK



```
G_structure = malloc(size1);
G_property = pmr_malloc(size2);
Open file & load data
```

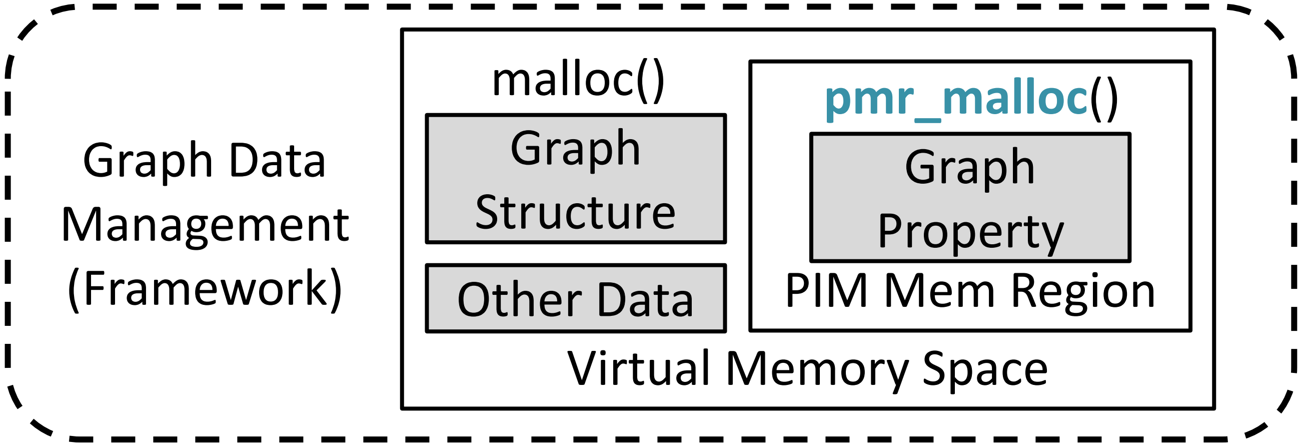
FRAMEWORK CHANGE

PIM memory region (PMR)

- Uncacheable memory region in virtual memory space
- Utilizing existing uncacheable (UC) support in X86

Framework change

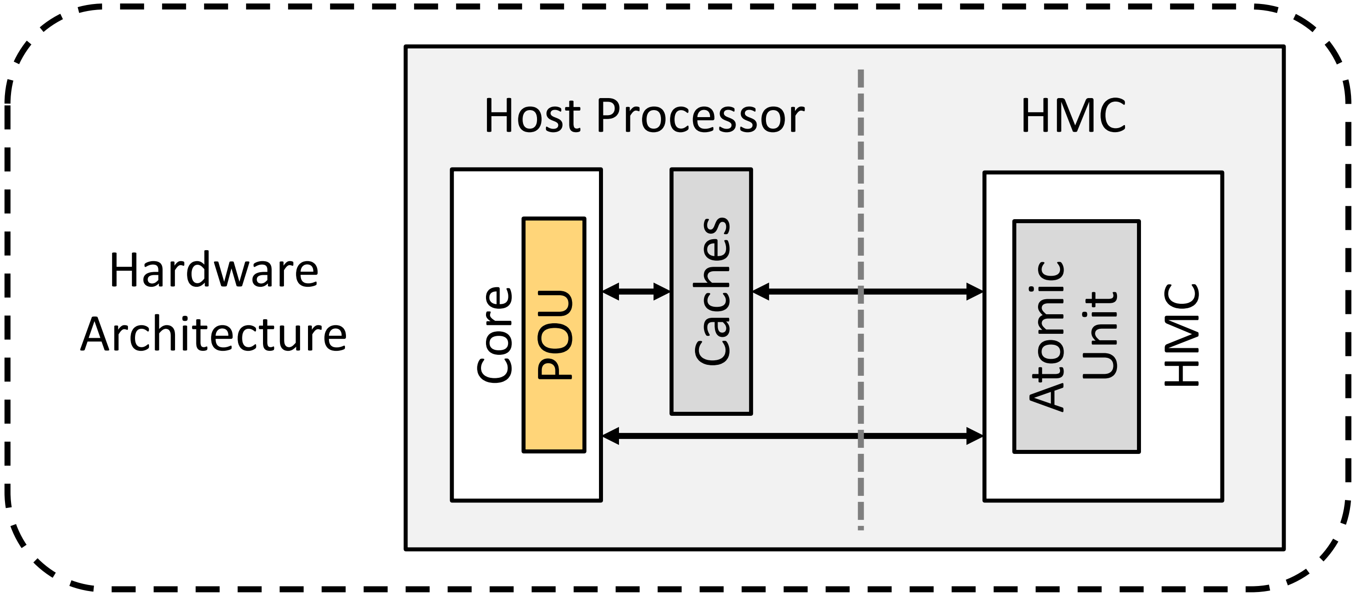
- malloc() → pmr_malloc()
- pmr_malloc(): customized malloc function that allocates mem objects in PMR



ARCHITECTURE CHANGE

PIM offloading unit (POU)

- Identifies **atomic** instructions that are accessing **PIM Memory Region**
- Offloads them as PIM instructions



CHANGES

Software changes:

- No **user application** change
- Minor change in framework: malloc() → pmr_malloc()

Hardware changes:

- PIM memory region: utilizes existing uncacheable (UC) support
- PIM offloading unit (POU): identifies offloading targets

No burden on programmers + Minor HW/SW change

EVALUATION

Simulation Environment

- SST (framework) + MacSim (CPU) + VaultSim (HMC)

Benchmark

- GraphBIG benchmark suite [Nai'15] (https://github.com/graphbig)
- LDBC dataset from *Linked Data Benchmarking Council* (LDBC)

Configuration

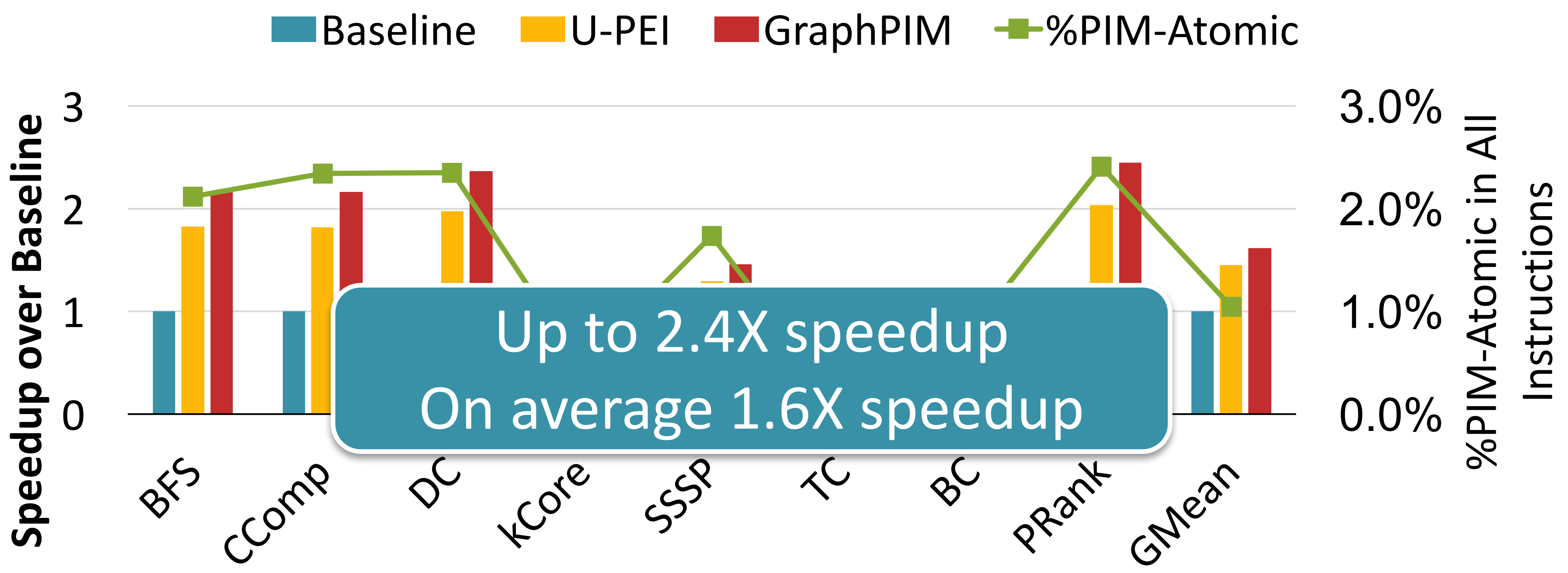
- 16 OoO cores, 2GHz, 4-issue
- 32KB L1/256KB L2/16MB shared L3
- HMC 2.0 spec, 8GB, 32 vaults, 512 banks, 4 links, 120GB/s per link

L. Nai et al. "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," SC'15

EVALUATION: PERFORMANCE

Baseline: No PIM offloading

U-PEI: Performance upper-bound of PIM-enabled instructions [Ahn'15]



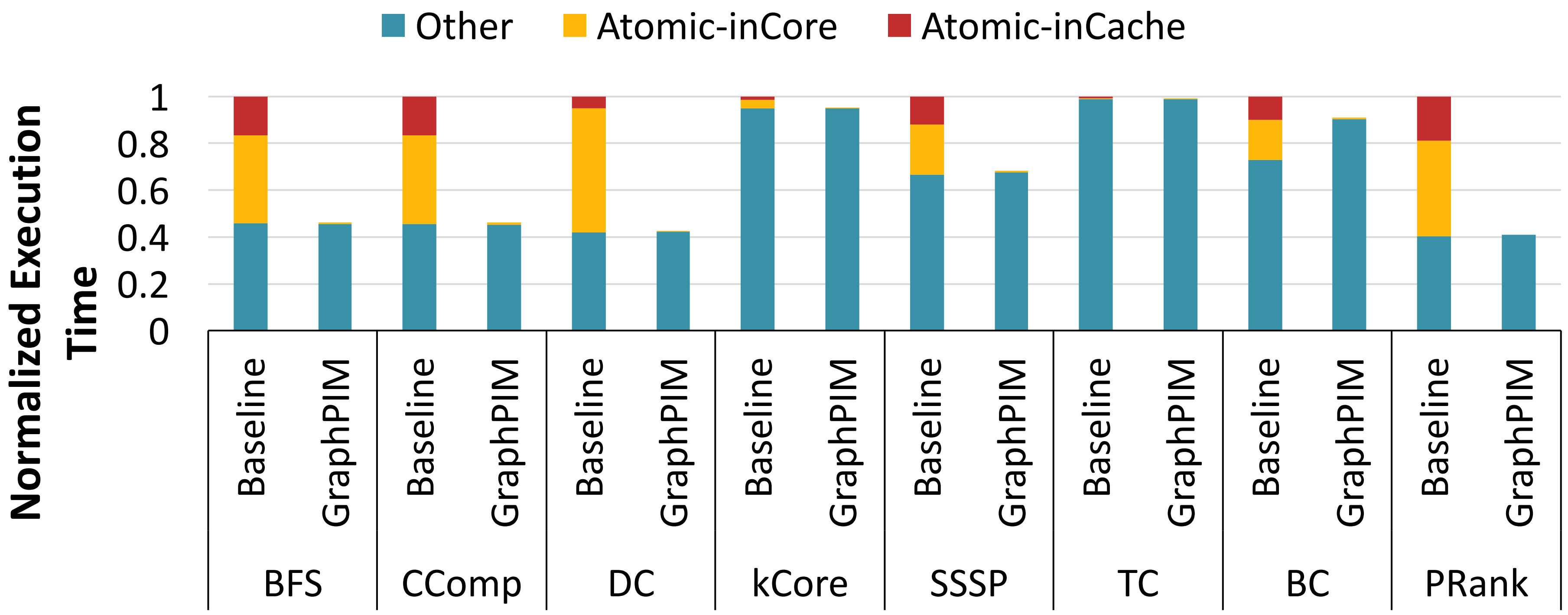
Up to 2.4X speedup
On average 1.6X speedup

J. Ahn et al. "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware PIM Architecture," ISCA'15

EVALUATION: EXECUTION TIME BREAKDOWN

Breakdown of normalized execution time

- Atomic-inCore: **atomic overhead** of offloading targets (atomic inst.)
- Atomic-inCache: **cache-checking overhead** of offloading targets



CONCLUSION

Graph computing is inefficiency on conventional architectures

GraphPIM enables PIM in graph computing frameworks

- Explores a new benefit of PIM offloading: **atomic overhead** reduction
- Identifies **atomic operations** on **graph property** as the offloading target
- Requires **no user-application change** and only **minor change** in framework and architecture

THANK YOU!

BACKUP SLIDES

DYNAMIC CACHE BEHAVIOR?

GraphPIM marks memory region statically

- Cons: cannot be **adaptive** to the working set sizes
- But, property accesses to graphs have very high cache misses regardless of graph inputs except for really small graph sizes [JPDC'16, SC'15]
- Pros: **coherence** support between memory and processor-cache is not required

CONSISTENCY?

PIM offloading for atomic instructions works fine because…

- The programming model of graph applications naturally avoids consistency issues: *all PIM writes are done before reads*
- Graph applications require only atomicity from atomic instructions
- But, atomic instructions in CPUs don't allow to specify atomicity without fence

- We also have a follow-up work discussing the consistency issue for PIM instructions in the context of **general applications** [MEMSYS'17]

CONSISTENCY?

Graph applications with BSP model naturally avoids consistency issues

- Barriers ensures all PIM writes are done before reads

Program Phases	Operation
loop:	
foreach vertex in task queue:	
read property	*// Reads*
fetch neighbor list	
foreach neighbor:	
update neighbor property	*// HMC Inst.*
update next-iter task queue	
barrier	

WHY GRAPHPIM IS A FRAMEWORK?

GraphPIM

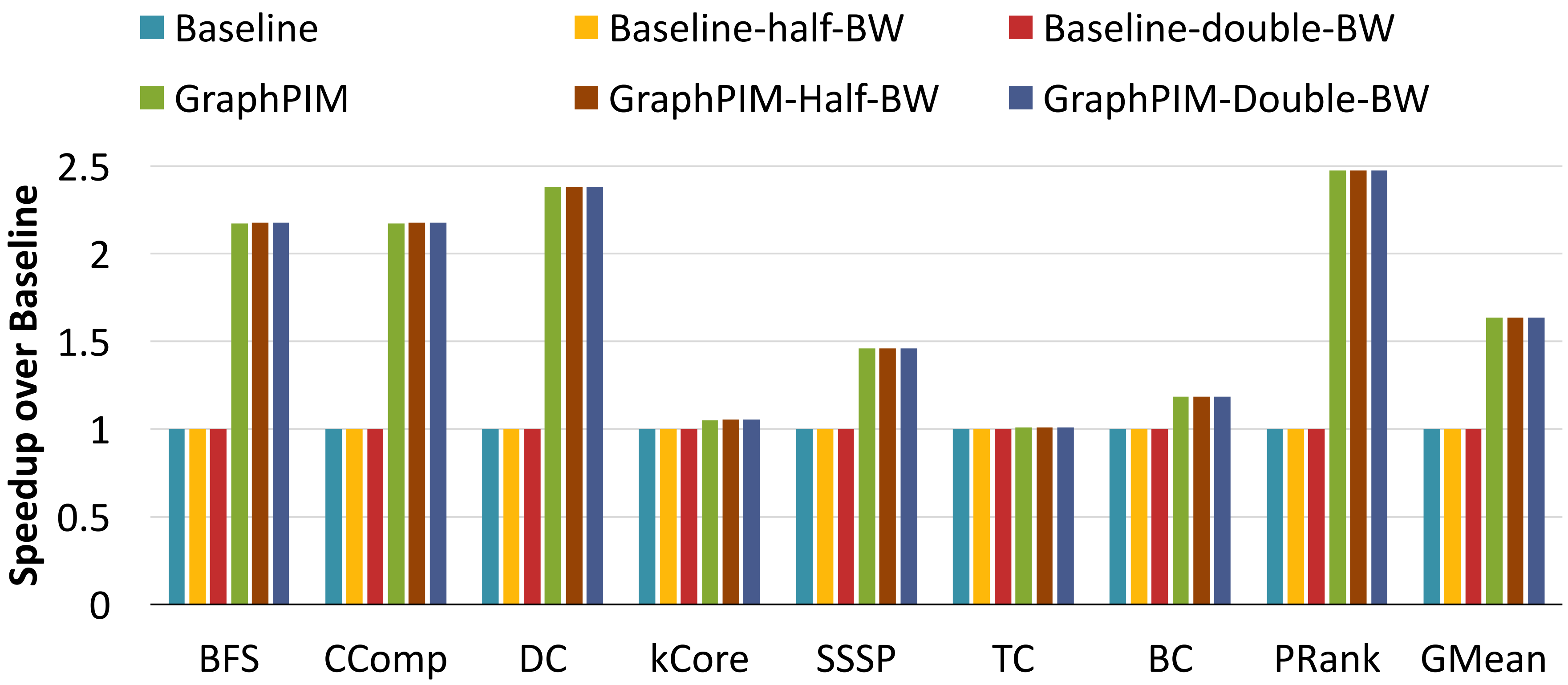
- Considers the **separation** of framework and user application
- Proposes a **full-stack** solution: SW framework + HW architecture
- Requires **no** application programmers' efforts

Users can easily enable/disable GraphPIM by switching between different framework libraries.

BANDWIDTH SENSITIVITY?

Graph on CPUs are not very sensitive to BW changes

- Speedup over baseline system with different HMC link bandwidth

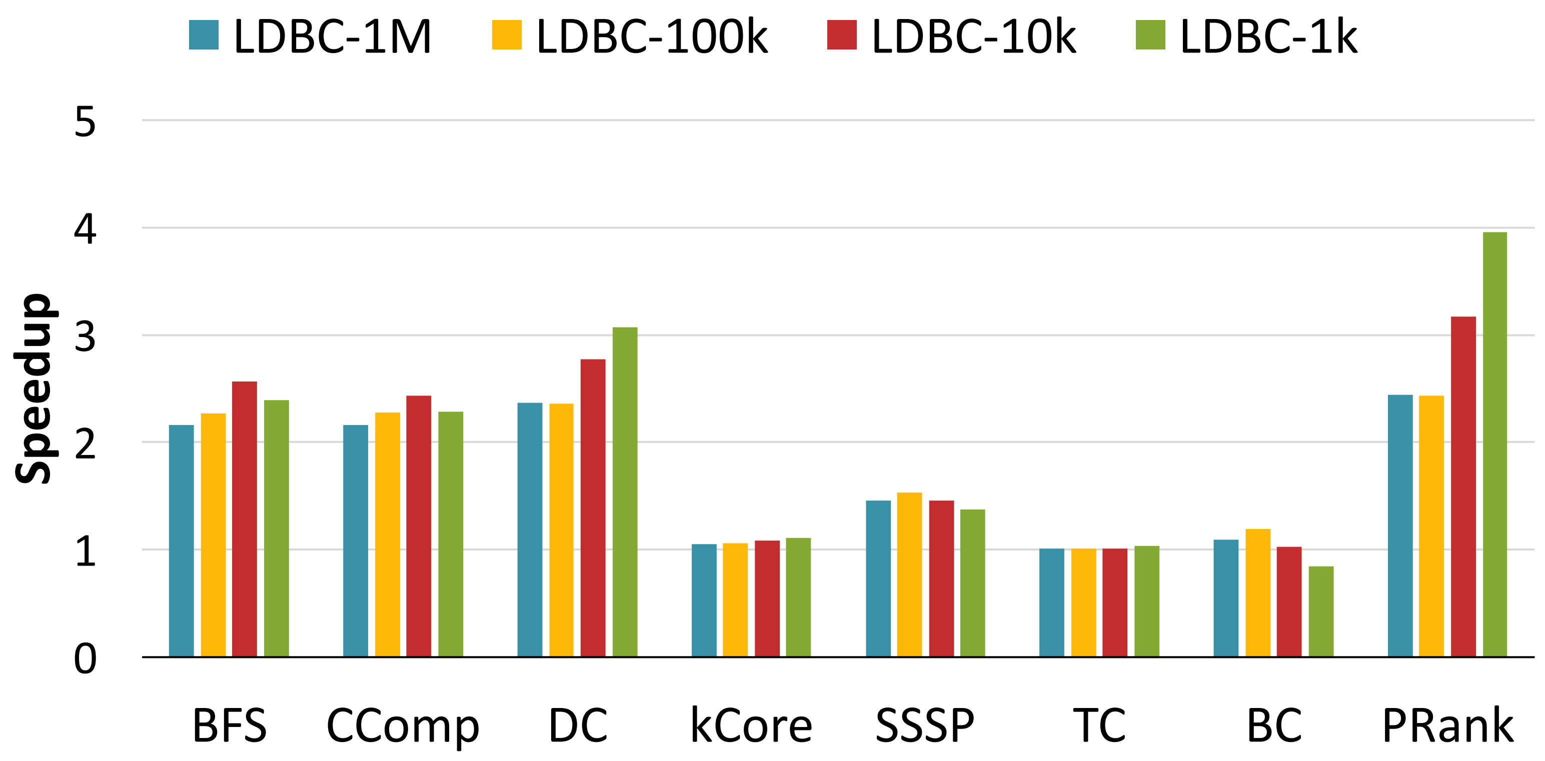


APPLICABILITY?

Category	Workload	Applicable?	Offloading Target	PIM Inst.
Graph Traversal	Breadth-first search	✔	lock cmpxchg	CAS if equal
	Degree centrality	✔	lock addw	Singed add
	Betweenness centrality	✘ ✔	(Floating point add)	(FP add)
	Shortest path	✔	lock cmpxchg	CAS if equal
	K-core decomposition	✔	lock subw	Singed add
	Connected component	✔	lock cmpxchg	CAS if equal
	Page rank	✘ ✔	(Floating point add)	(FP add)
Dynamic Graph	Graph construction	✘	(Complex operation)	
	Graph update	✘	(Complex operation)	
	Topology morphing	✘	(Complex operation)	
Rich Property	Triangle count	✔	lock add	Singed add
	Gibbs inference	✘	(Compute intensive)	

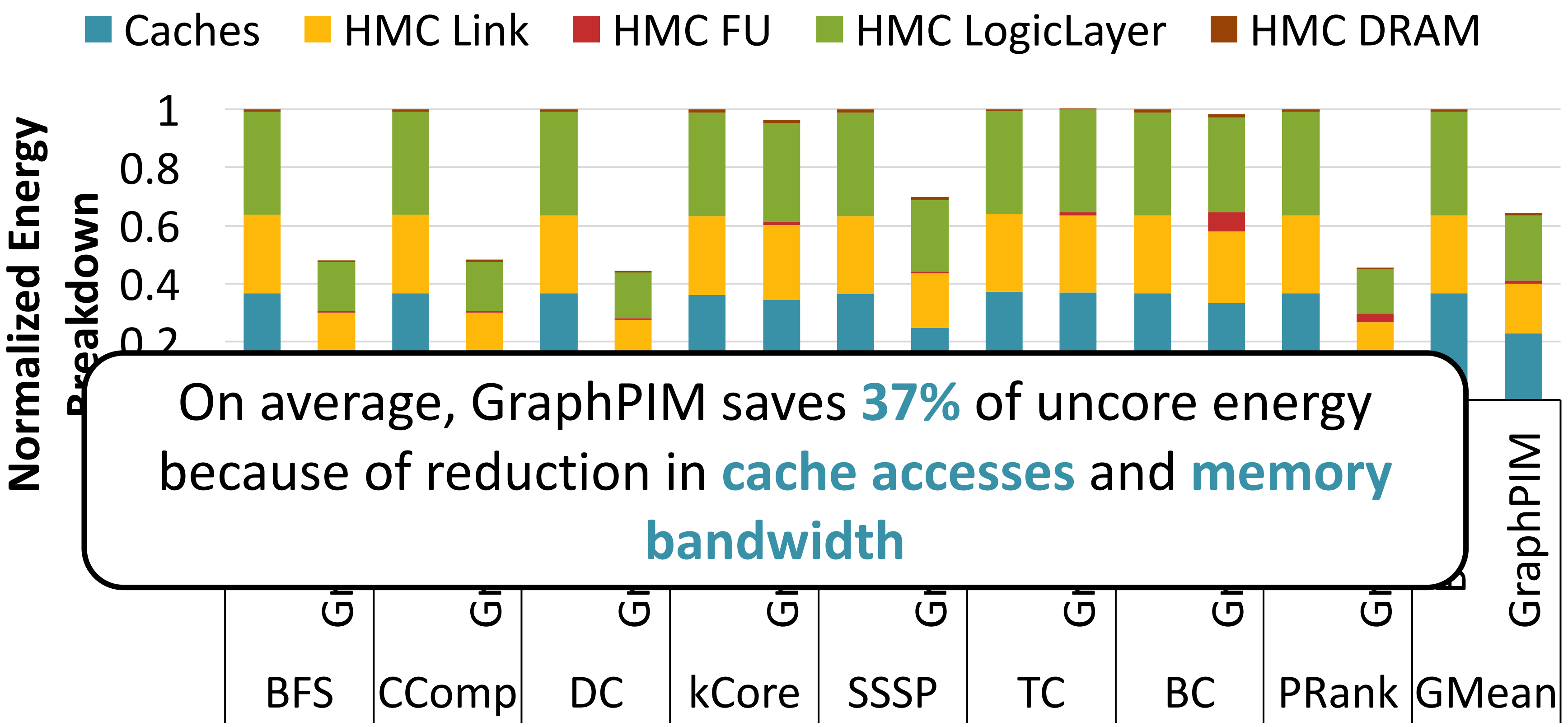
GRAPHPIM: EVALUATION

GraphPIM speedup over baseline with different dataset sizes



GRAPHPIM: EVALUATION

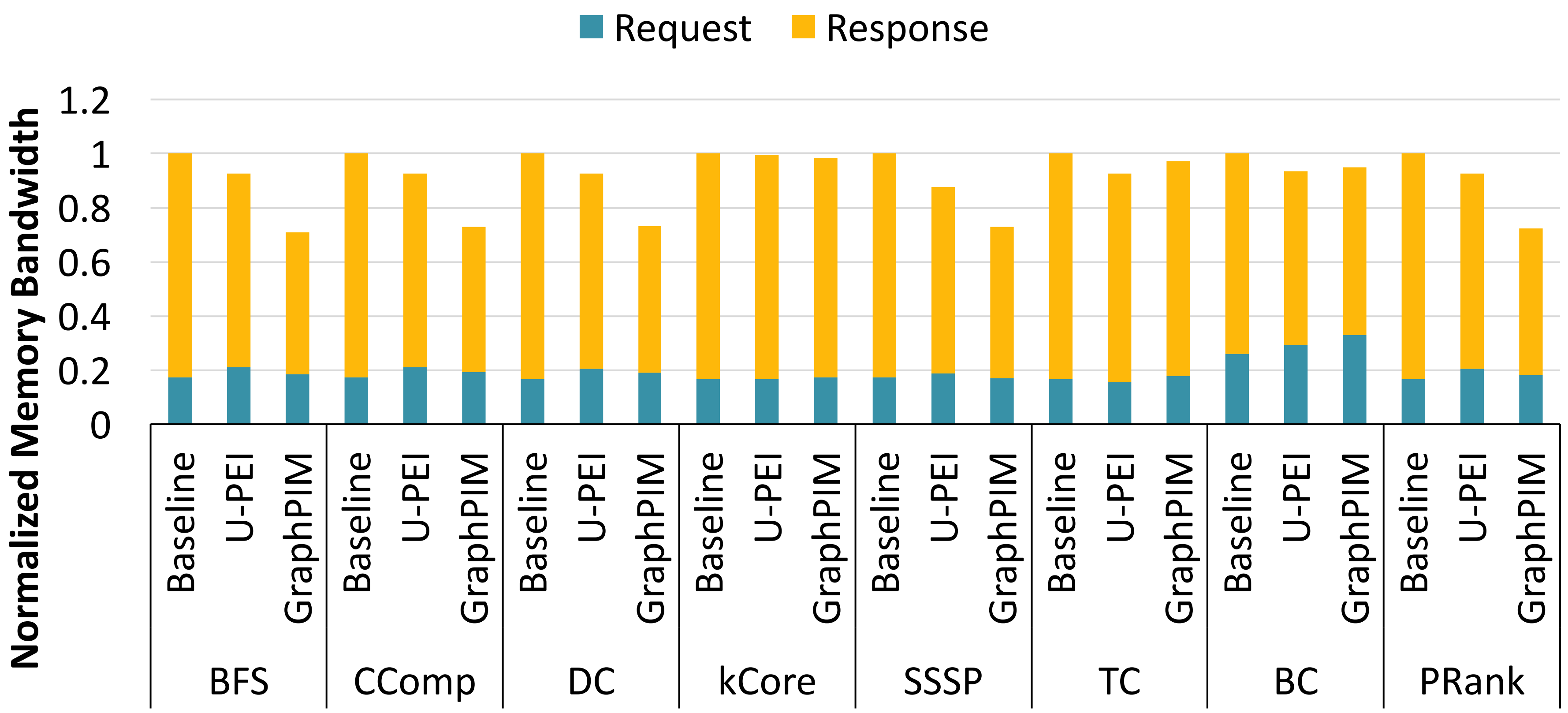
Normalized uncore energy consumption



On average, GraphPIM saves **37%** of uncore energy because of reduction in **cache accesses** and **memory bandwidth**

GRAPHPIM: EVALUATION

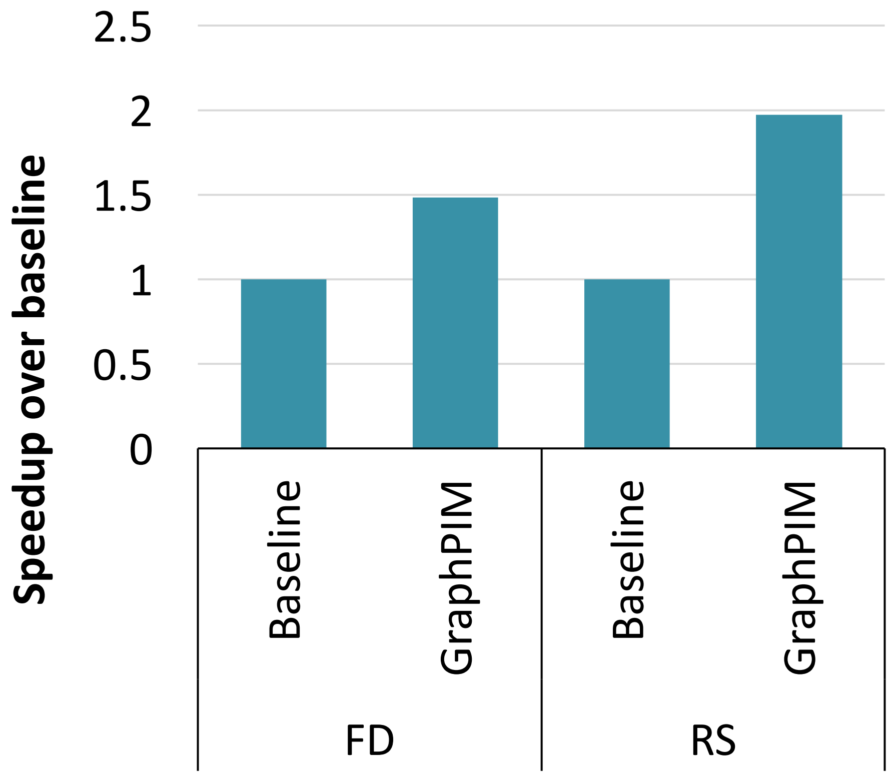
Normalized bandwidth consumption with request/response breakdown

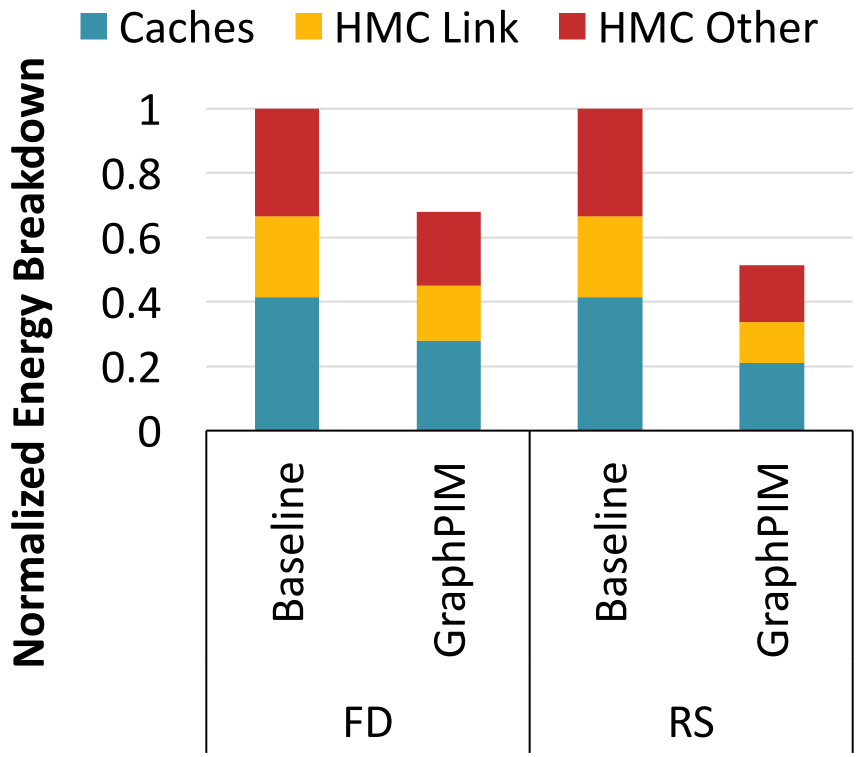


GRAPHPIM: EVALUATION

Performance and energy results of two real-world applications

- Based on an analytical model
- FD: Financial fraud detection; RS: Recommender system



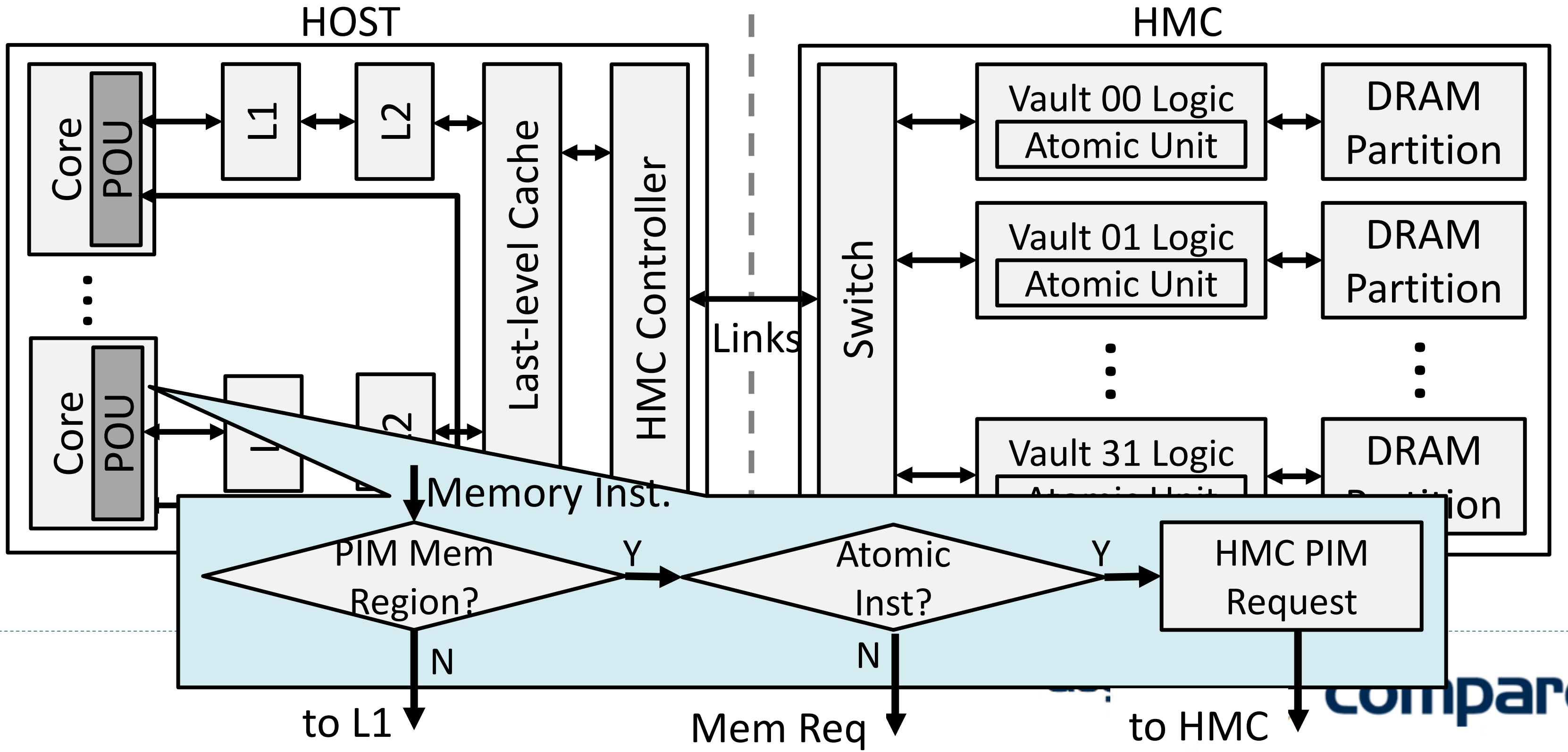


HARDWARE CHANGES

PIM Memory Region (PMR)

- A uncacheable memory region in virtual memory space
- Utilizing existing **uncacheable** (UC) support in X86

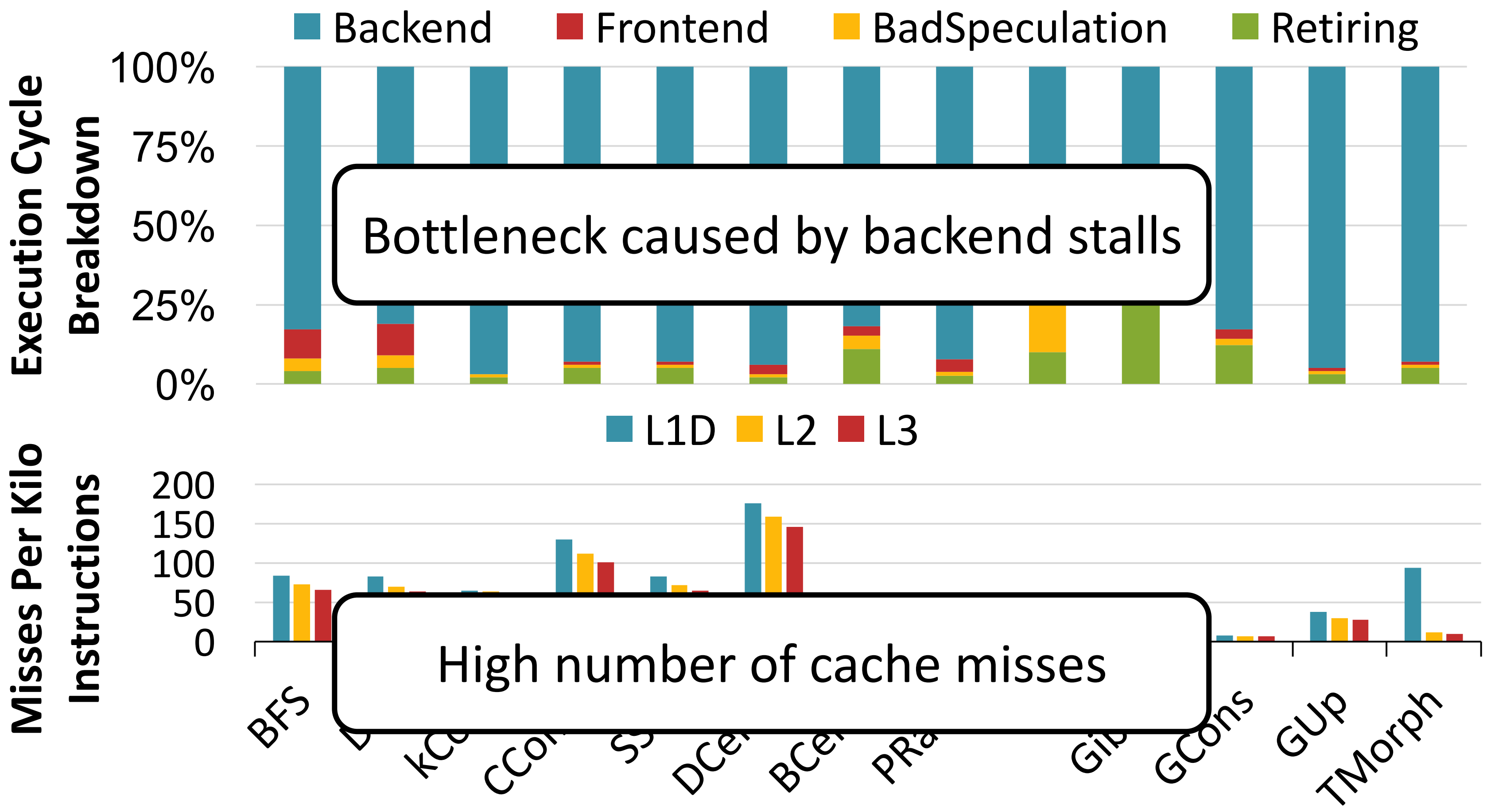
PIM Offloading Unit (POU)



MOTIVATION

Profiling using HW performance counters

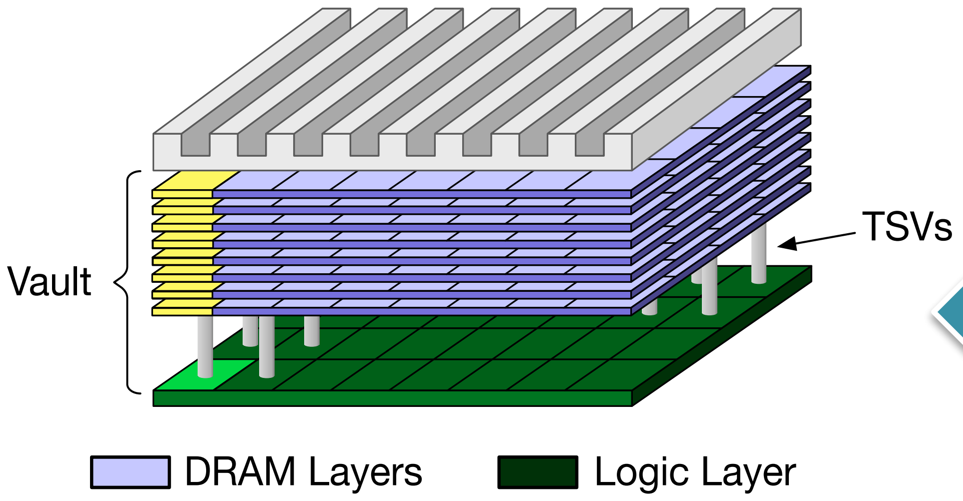
- Execution cycle breakdown: top-down methodology from Intel



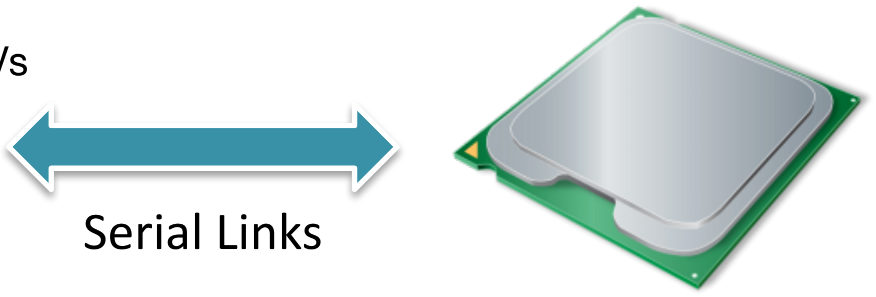
BACKGROUND: PIM OFFLOADING IN HMC 2.0

Hybrid Memory Cube (HMC) 2.0

- One of the first industrial PIM proposals
- **Instruction-level** PIM offloading



- 1 logic die + 4/8 DRAM dies
- 32 Vaults
- 4 serial links



BACKGROUND: PIM OFFLOADING IN HMC 2.0

Packet-based protocol

8 byte	0~256 byte	8 byte
Tail	Payload	Header

Regular READ/WRITE

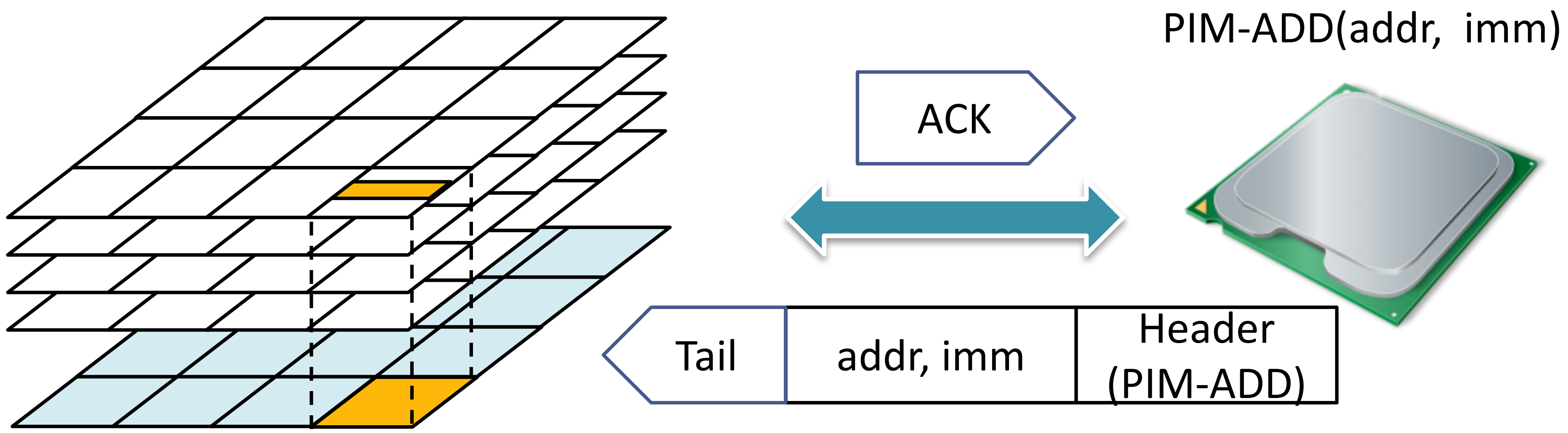
- FLIT: 16-byte; basic flow unit

	Request	Response
64-byte READ	1 FLIT	5 FLITs
64-byte WRITE	5 FLITs	1 FLIT

BACKGROUND: PIM OFFLOADING IN HMC 2.0

PIM Instruction: **read-modify-write** (RMW) operation

- Similar as regular READ/WRITE, just different **CMD** in the **Header**
- DRAM bank is locked during the whole RMW for atomicity


PIM-ADD(addr, imm)
ACK
Tail
addr, imm
Header (PIM-ADD)