Runtime Data Management on Non-Volatile Memory-based High Performance Systems

Kai Wu

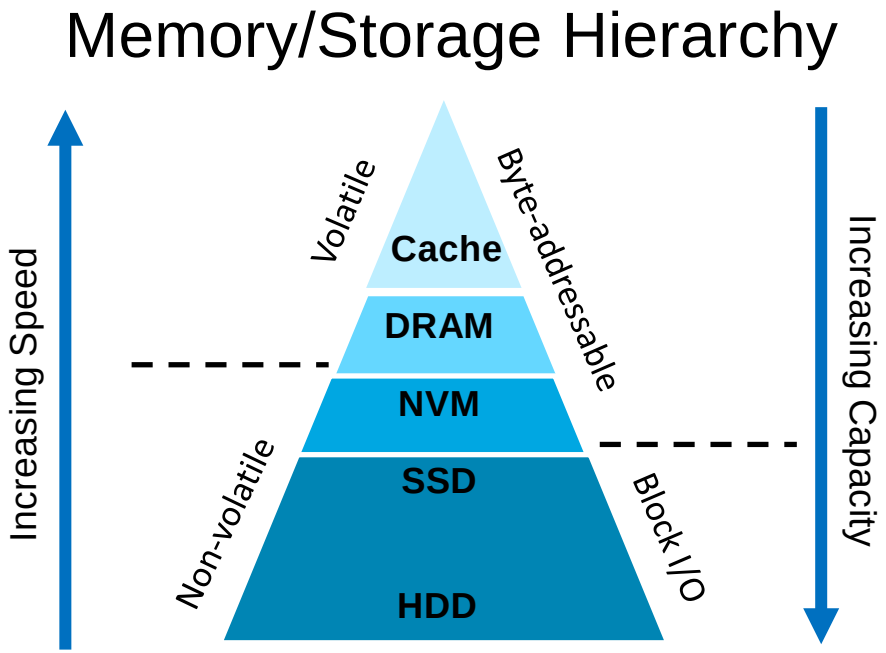
Advisor: Dong Li

University of California, Merced

Non-Volatile Memory (NVM)

- NVM techniques:
  - PCM, ReRAM, and 3DXpoint …
- NVM has arrived!

- Memory like performance
  - ~100x faster than SSDs
  - Offers byte-addressability
- Storage like characteristics
  - Data persistence
  - Large capacity
    - Each CPU socket can have as much as 4.5 TB



NVM and persistent memory (PM) can be used interchangeably in the slides

Using NVM is Challenging

- NVM is fast but not enough

	PCM	ReRAM	Optane	DRAM
Read (ns)	20 – 70	20 - 50	170 – 310	10
Write (ns)	150 – 220	70 – 140	110 – 180	10
Non-volatility	√	√	√	×
Standby Power	~0	~0	~0	High

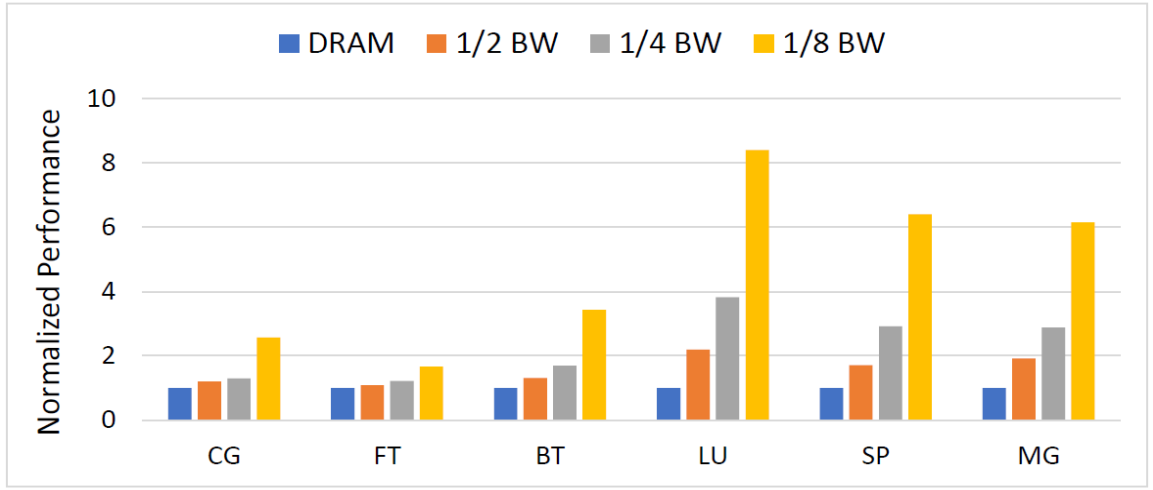
C. Xu et al. “Overcoming the Challenges of Crossbar Resistive Memory Architectures”, HPCA, 2015.
K. Suzuki and S. Swanson. “A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014”, IMW 2015.
A.Renen et al. “Persistent Memory I/O Primitives”, DaMon 2019.

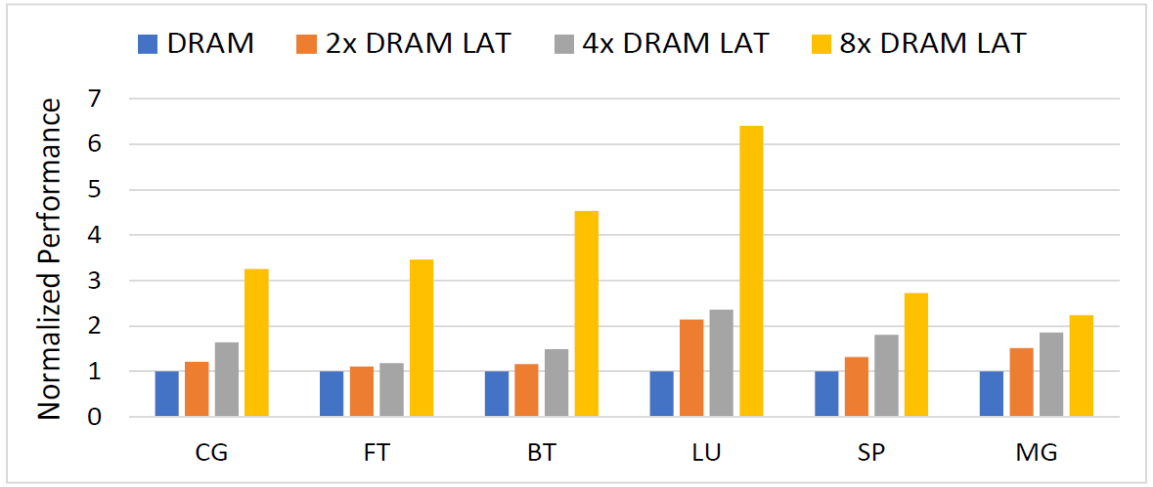
- Performance studies[1,2] show that using NVM substitute DRAM as main memory for running applications causes a significant performance slowdown on applications (Up to 15x)

[1] Kai Wu, Yingchao Huang, and Dong Li. "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogenous Main Memory“, SC’17.
[2] Ivy Peng, Kai Wu, Jie Ren, Dong Li and Maya Gokhale. "Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems", IPDPS’20.

Performance Study Using NVM as Main Memory

The benchmark performance (execution time) on NVM-only memory with various memory bandwidth and latency



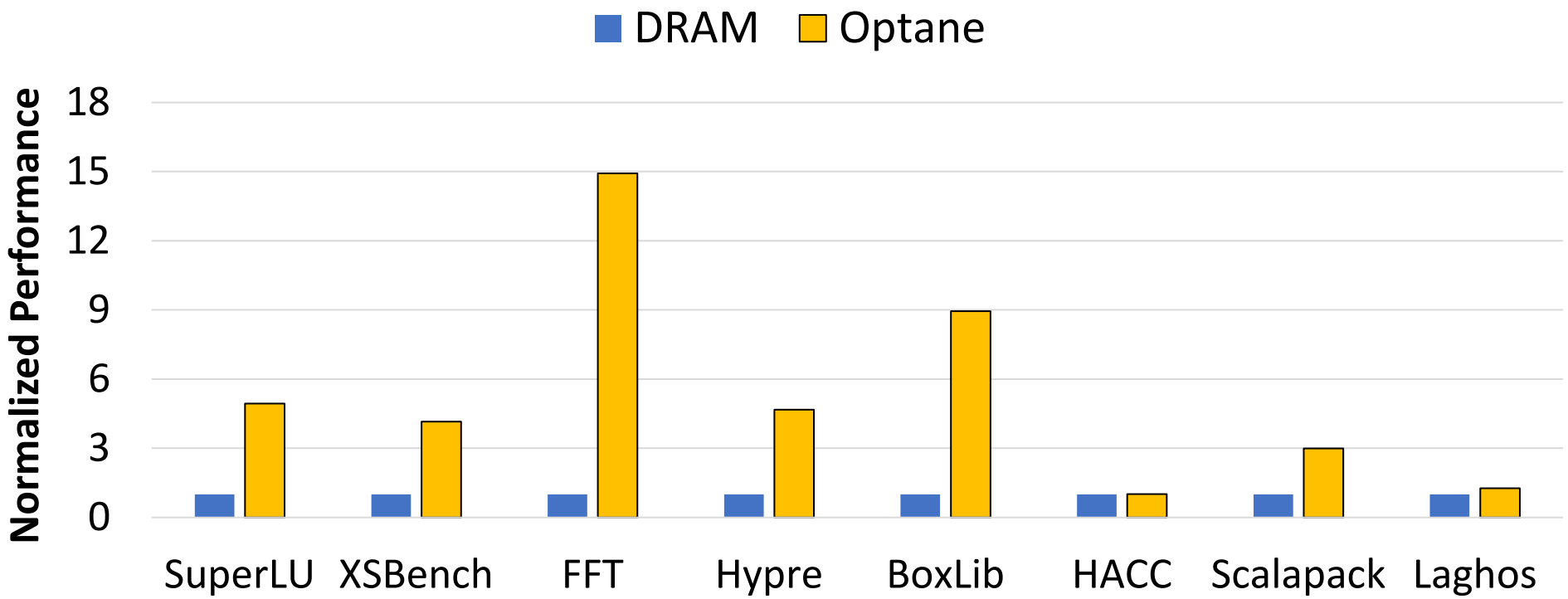


- We use six NPB benchmarks (Class D) as input and run 16 MPI processes on 4 nodes with Quartz emulator
- Performance decreases by 1.57x, 2.3x and 4.77x on average, when NVM is configured with 1/2, 1/4 and 1/8 DRAM bandwidth respectively
- Performance decreases by 1.41x, 1.72x and 3.77x, when NVM is configured with 2x, 4x and 8x DRAM latency respectively

Kai Wu, Yingchao Huang, and Dong Li. "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogenous Main Memory“, SC’17

Performance Study Using NVM as Main Memory

The benchmark performance (execution time) on NVM-only memory



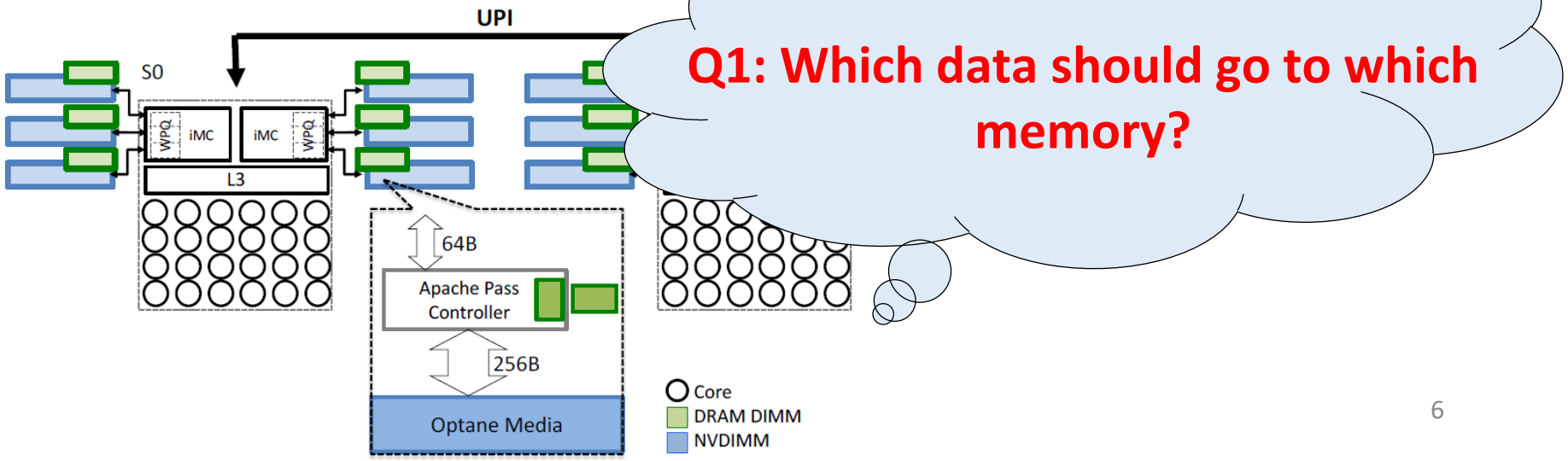
- Run seven representative HPC applications on Intel Optane DC PM-based platform
- Performance decreases by 5.36x on average

Ivy Peng, Kai Wu, Jie Ren, Dong Li and Maya Gokhale. "Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems", IPDPS'20.

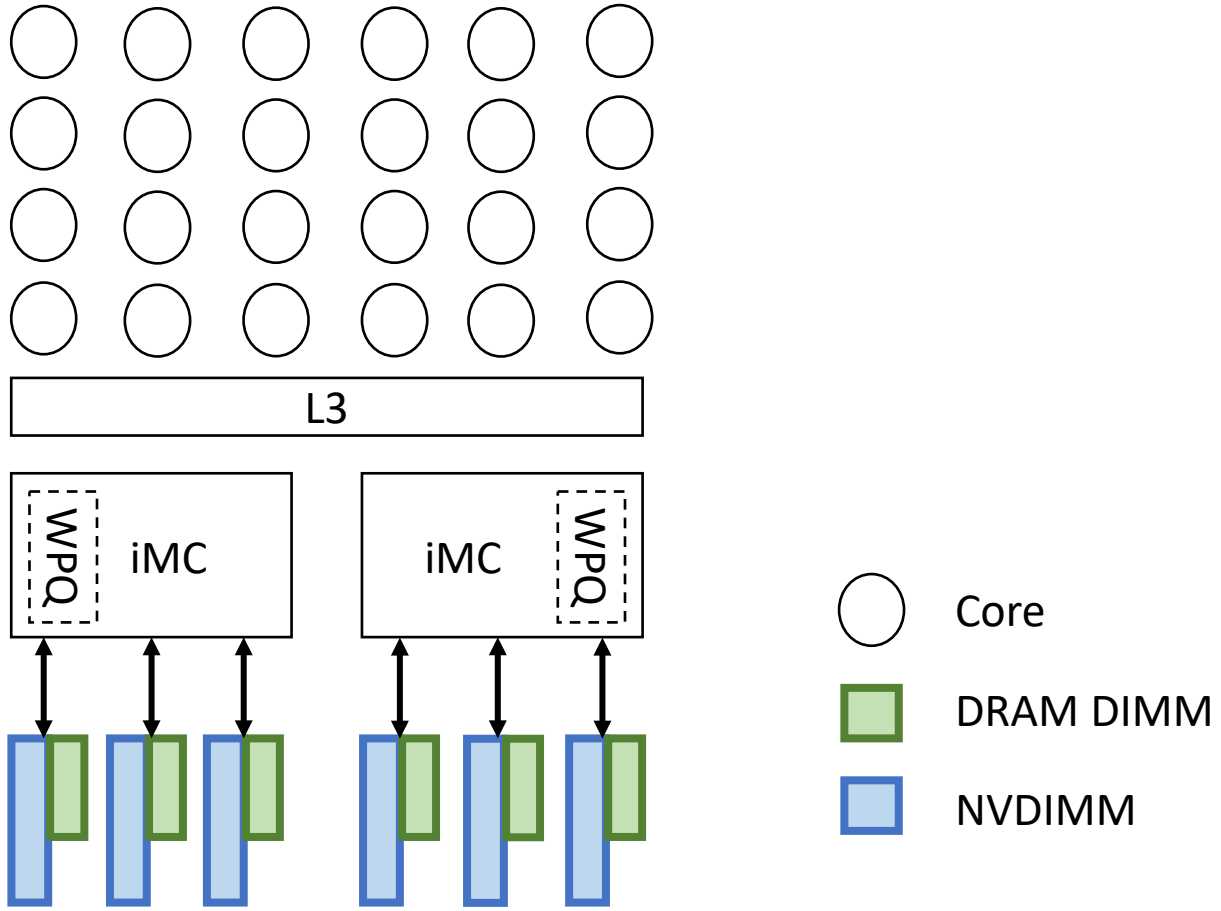
NVM-based Heterogeneous Memory System

- Pair NVM and DRAM to build a heterogeneous memory system (HMS)
- State-of-the-art NVM-based HMS (Intel Optane platform)
  - Two sockets, 2nd Gen Intel Xeon Scalable processor
  - Six 128-GiB NVDIMMs, six 16-GiB DRAM DIMMs, two internal memory controllers (iMCs), per socket
    - DRAM: 87 ns latency & 104 GB/s peak bandwidth
    - Optane: 304 ns read latency & 39 GB/s peak read bandwidth & 13 GB/s peak write bandwidth
  - 256-byte Optane Intel transaction, 64-byte Host-Memory transaction

Byteaddresable NVM

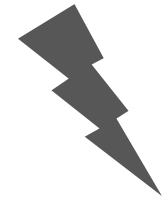


Using NVM is Challenging

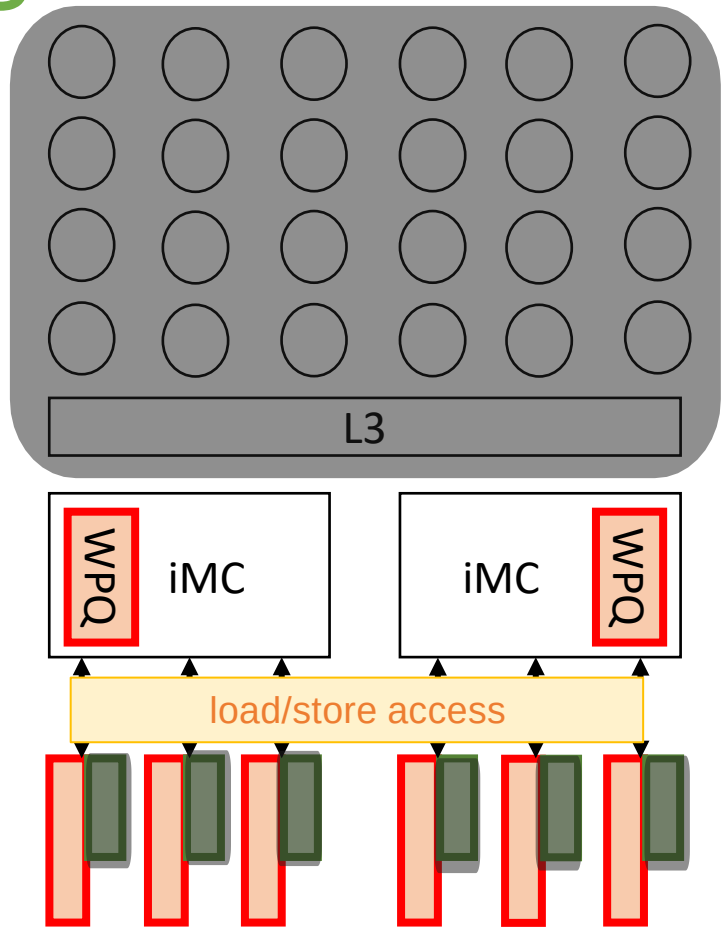


Using NVM is Challenging

Failure

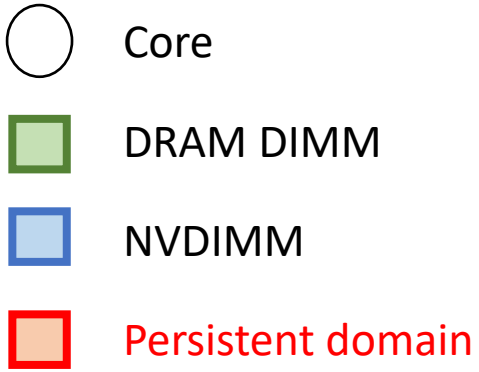


Data Lost



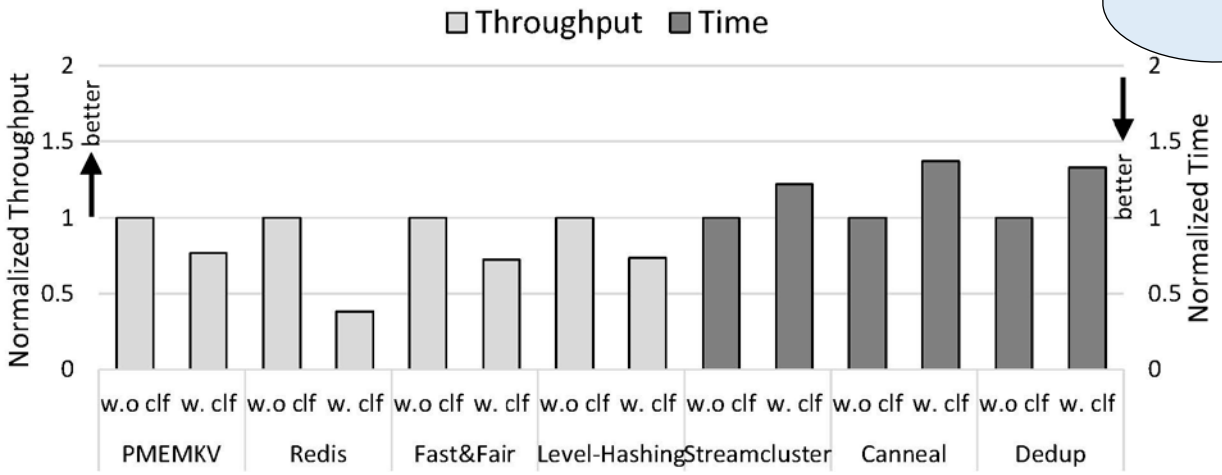
- PM is byte-addressable (load & store access) as DRAM
  - The file system is unaware of the data changes on PM
  - CPU or memory controller reorders memory writes

After a failure, the data on the PM may not be in the correct state



Cache Line Flushing (CLF) is a Fundamental Building Block for Programming NVM

- Ensure write ordering to NVM
  - Programmers need to use assembly instructions (e.g., clflush, clflushopt, clwb, and sfence) to explicitl flush data from cache to NVM



Q2: How to efficiently ensure data persistence on NVM?

rep ent aware wo loads with 24 application threads. CLF and fence significantly affect performance by 24% - 62%

Kai Wu, Ivy Peng, Jie Ren, and Dong Li. "Ribbon: High Performance Cache Line Flushing for Persistent Memory", PACT'20

Research Progress on NVM

- Focus on using software solutions to solve data placement and data persistence on NVM-based systems

- **Q1: Data placement on NVM-based HMS**
  - Runtime data management in NVM-based HMS for MPI programs (SC'17) 
  - Runtime data management on NVM-based HMS for task-parallel programs (SC'18)

- **Q2: Data persistence on NVM-based main memory**
  - High performance cache flushing for persistent memory (PACT'20) 
  - Exploring non-volatility of NVM for HPC (Cluster'20)
  - Algorithm-directed crash consistence in NVM for HPC (Cluster'17)
  - Architecture-aware, high performance transaction for persistent memory (Under submission)

- Others
  - Demystifying the performance of HPC scientific applications on NVM-based memory systems (IPDPS'20)

Runtime Data Placement on NVM-based Heterogeneous Memory System for MPI Programs

Kai Wu, Yingchao Huang, and Dong Li. "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogenous Main Memory." In 29th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2017.

Definitions and Basic Assumptions

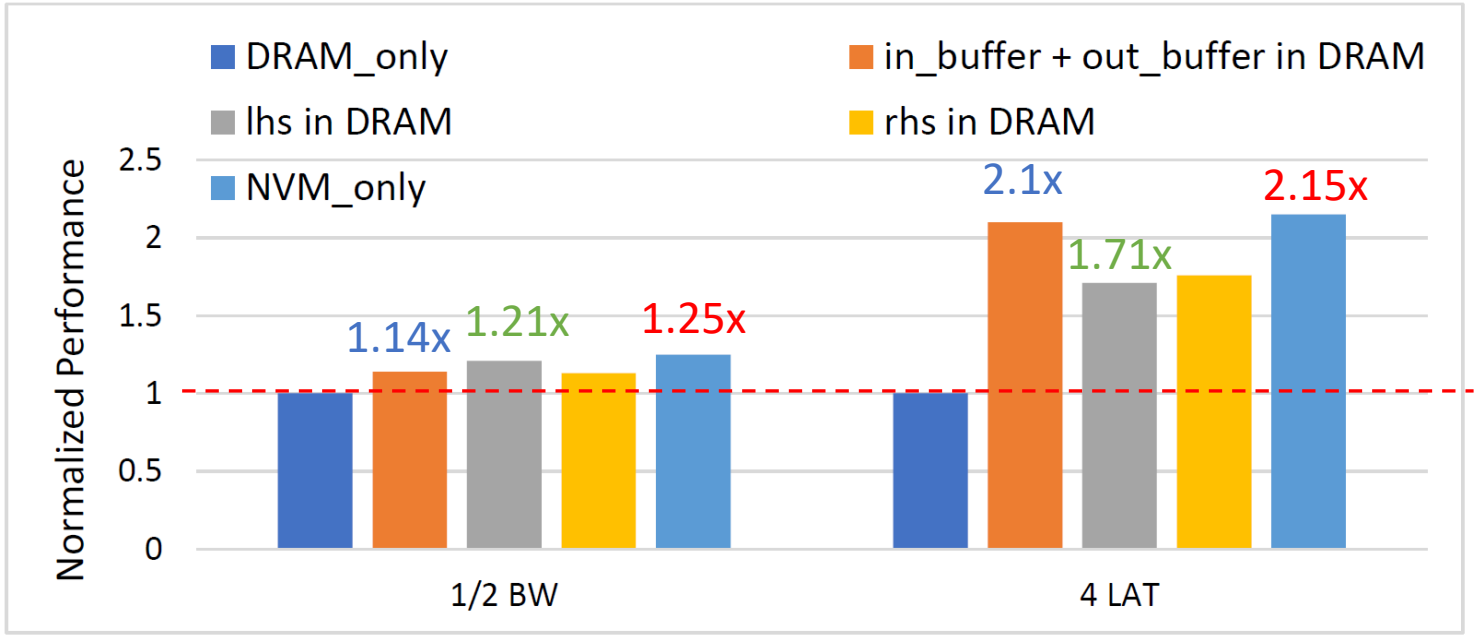
- We target on the MPI programming model, and decompose the MPI-based application into phases

- We target on parallel applications from the HPC domain with an iterative structure

CG (Conjugate Gradient) benchmark

```
...
loop
{
  q = A.p                    <-- A phase
  mpi_irecv                  <-- A phase
  mpi_send                   <-- A phase
  mpi_wait                   <-- A phase
  p.q                        <-- A phase
  mpi_irecv                  <-- A phase
  mpi_send                   <-- A phase
  mpi_wait                   <-- A phase
  alpha = rho/(p.q)          <-- A phase
  z = z + alpha*p            <-- A phase
  r = r- alpha*q             <-- A phase
  rho = r.r                  <-- A phase
  mpi_irecv                  <-- A phase
  mpi_send                   <-- A phase
  mpi_wait                   <-- A phase
  p = r + beta*p             <-- A phase
}
...
```

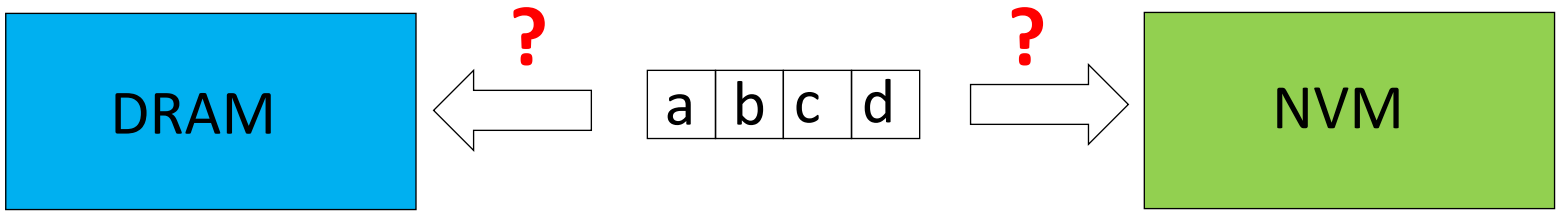
A Preliminary Performance Study for Data Placement



- Run SP (Scalar Penta-diagonal solver) benchmark on 4 nodes with 4 MPI processes
- A good data placement can effectively bridge the performance gap between DRAM and NVM (31% performance improvement on average)
- Different data objects manifest different sensitivity to memory bandwidth and latency
  - Data objects in_buffer and out_buffer are sensitive to bandwidth, not latency
  - Data object lhs is sensitive to latency, not bandwidth

Research Challenges

- First, how to capture and characterize memory access patterns associated with data objects?
  - Bandwidth sensitive vs. Latency sensitive
- Second, how to strike a balance between different requirements on the frequency of data movement?
  - Frequent movement for better performance vs. Frequent movement resulting in data movement overhead
- Third, how to minimize the impact of data movement on application performance?



Story in a Nutshell

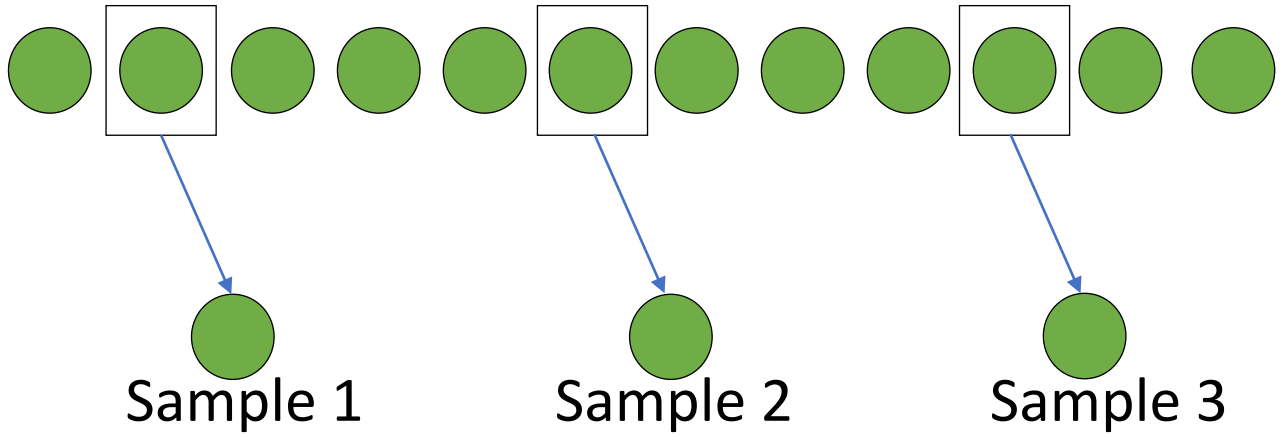
- Unimem: A lightweight runtime system to manage data placement
  - No hardware modification
  - No disruptive changes to application and system software
- Performance model driven
- Performance optimizations to hide the cost of data movement

Unimem Design: Three Steps

- Step 1: phase profiling
  - Collect memory access information for each phase
- Step 2:performance modeling
  - Bandwidth sensitivity vs. latency sensitivity
  - Modeling benefit and cost of data movement
- Step 3: data placement decision and enforcement
  - Decide the best data placement locally and globally

Step 1: Phase Profiling

- Our goal: collect main memory access events and map them into data objects.
- Use sampling-based hardware performance counters
  - Known as Precise Event-Based Sampling (Intel) or Instruction-based Sampling (AMD)
  - Collect the number of last level cache miss event
  - Map the event information to data objects via memory addresses
- Introduce constant factors to account for the sampling inaccuracy



Step 2: Performance Modeling

- The performance models estimate performance benefit and data movement cost between NVM and DRAM
  - We trigger data movement only when the benefit outweighs the cost

- To calculate the performance benefit, we must decide if the data object is bandwidth sensitive or latency sensitive

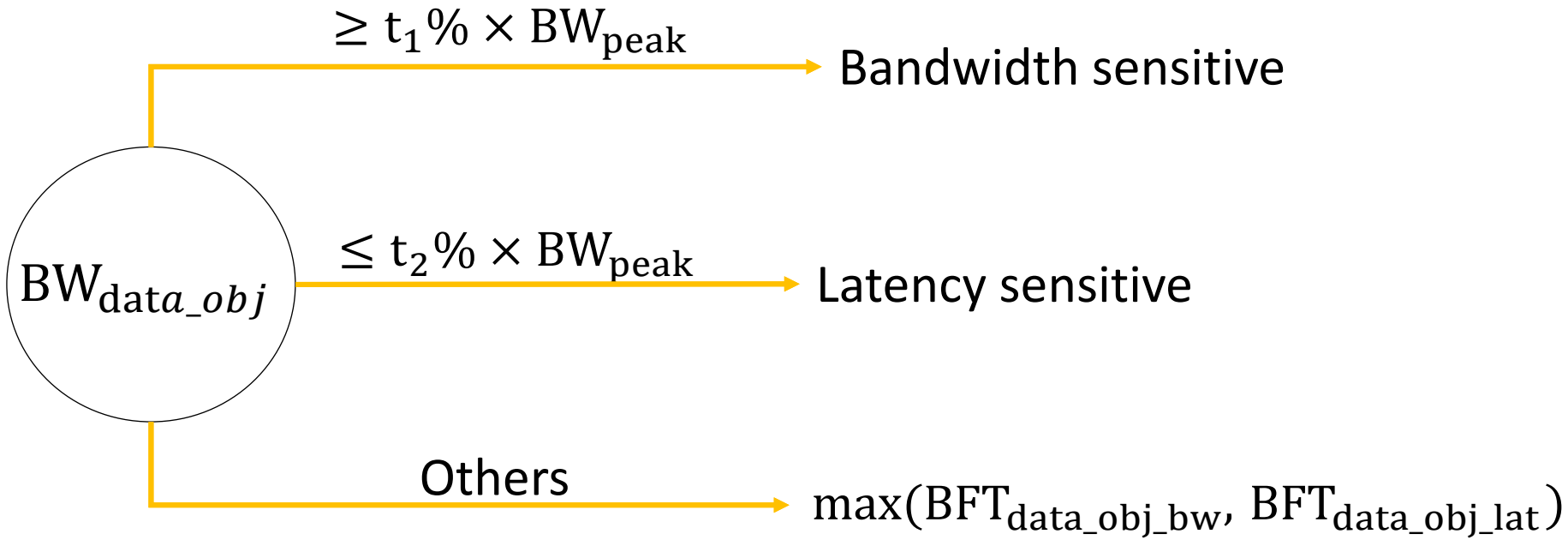
Estimating Memory Bandwidth Consumption

- We estimate main memory bandwidth consumption due to memory accesses to the each data object (BW_{data_obj})

$$BW_{data\_obj} = \frac{\#data\_access \times cacheline\_size}{\frac{\#samples\_with\_data\_accesses}{\#sample} \times phase\_execution\_time}$$

$\#data\_access$: the number of memory accesses to the data object in main memory (collected in phase profiling)
$\#samples\_with\_data\_accesses$: the number of samples that accesses the target data object
$\#samples$: the total number of samples

Bandwidth Sensitivity vs. Latency Sensitivity



- $t_1 = 80$ in our evaluation
- $t_2 = 10$ in our evaluation
- BW_{peak}: Practical peak bandwidth measured by STREAM benchmark (highly memory bandwidth intensive)
- Others: $t_2\% \times BW_{peak} < BW_{data\_obj} < t_1\% \times BW_{peak}$
- $BFT_{data\_obj\_bw}$ and $BFT_{data\_obj\_lat}$ will be discussed in the next slide

Calculation of Data Movement Benefit

- If the data object is bandwidth-sensitive:

$$BFT_{data\_obj\_bw} = \left(\frac{\#data\_access \times cacheline\_size}{NVM_{bw}} - \frac{\#data\_access \times cacheline\_size}{DRAM_{bw}}\right) \times CF\_bw$$

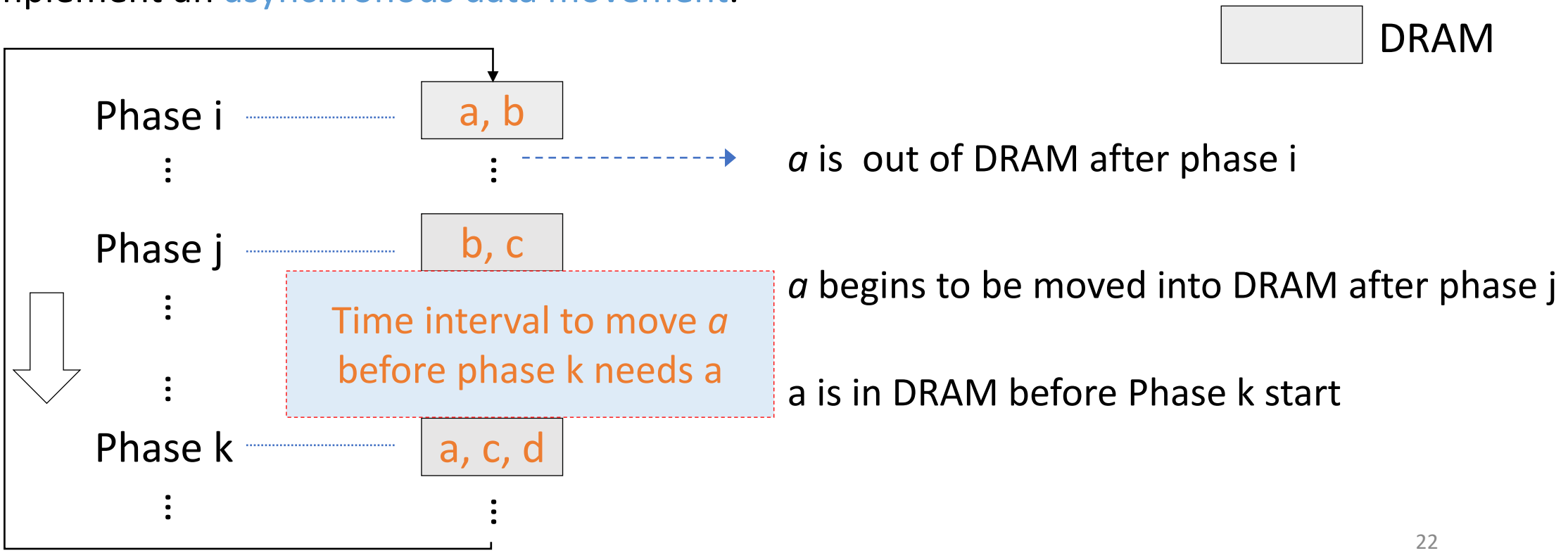
Compensating for HW counter inaccuracy

- If the data object is latency-sensitive:

$$BFT_{data\_obj\_lat} = (\#data_{access} \times NVM_{lat} - \#data\_access \times DRAM\_\text{lat}) \times CF\_lat$$

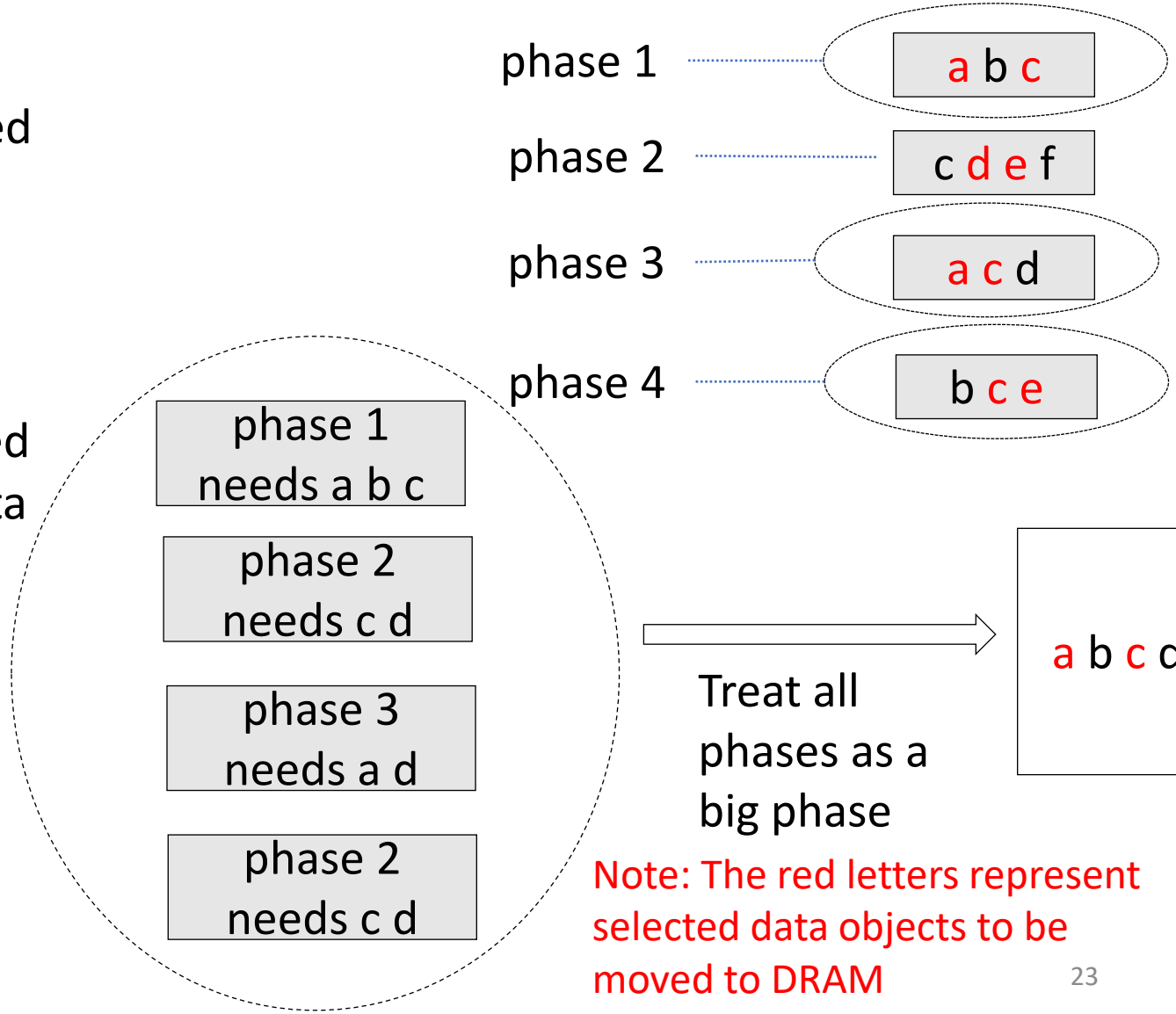
Calculation of Data Movement Cost

- $COST_{data\_obj} = \max\left(\frac{data\_size}{mem\_copy\_bw} - mem\_comp\_overlap, 0\right)$
- $mem\_comp\_overlap$: run a helper thread in parallel with the application to implement an asynchronous data movement.



Step 3: Data Placement Decision and Enforcement

- Phase local search
  - Find the local optimal solution based on dynamic programming
- Cross-phase global search
  - All phases are treated as a combined single phase to find the optimal data placement
- Compare the two searches and then choose the better one

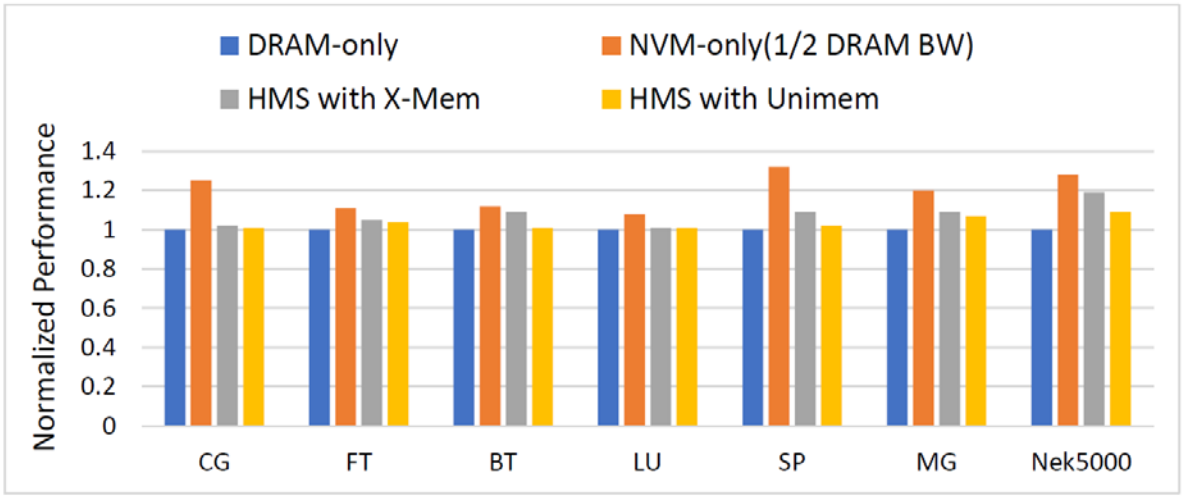


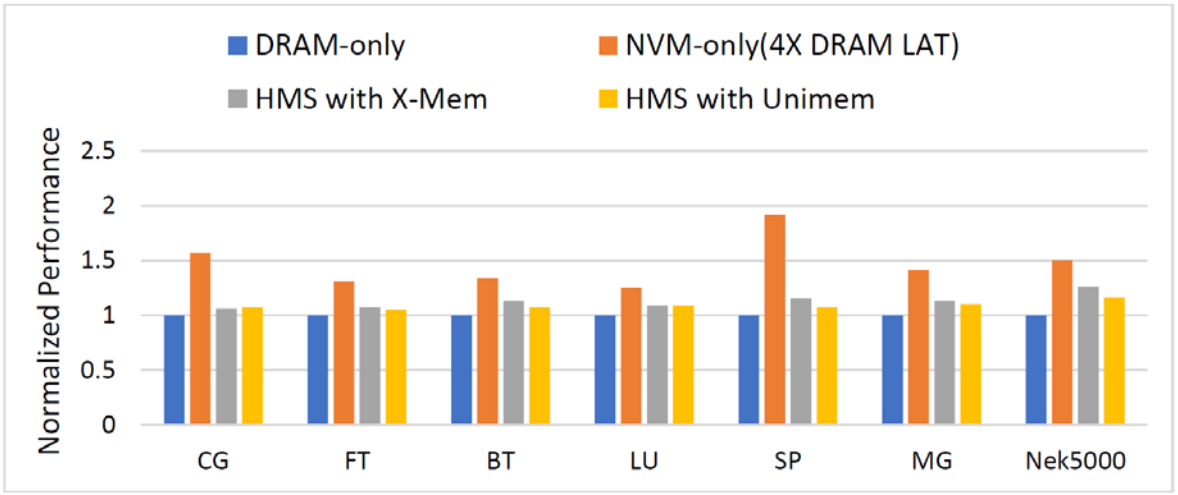
Performance Evaluation

- NVM emulators
  - HP Quartz: enables the emulation of NVM latency and bandwidth characteristics
- Workloads
  - CG, FT, BT, LU, SP, MG from NAS parallel benchmark suite 3.3.1
  - Nek5000: eddy input problem with a 256 × 256 mesh
- Compare with X-Mem (Intel)[1]
  - PIN-based offline profiling method
  - Do not consider data movement cost
  - Assume a homogeneous memory access pattern associated within a data object

[1] S. Dulloor et al. “Data tiering in heterogeneous memory systems”, EuroSys, 2016.

Basic Performance Test





- Unimem greatly narrows the performance gap between NVM and DRAM
  - The average performance difference between DRAM-only and HMS is only 3% for NVM with 1/2 DRAM bandwidth
  - The average performance difference between DRAM-only and HMS is 7% for NVM with 4X DRAM latency
- Unimem performs 10% better than X-mem for Nek5000

Summary

- We quantify the performance gap between NVM- and DRAM-based systems, and demonstrate that using a carefully designed runtime, it is possible to significantly reduce the performance gap

- We introduce a lightweight runtime for MPI-based HPC applications

- We evaluate Unimem with six representative HPC workloads and one production code (Nek5000). The performance differnce between DRAM-only and HMS with Unimem is only 6.2% on average and 16% at most. We successfully narrow the performance gap and demonstrate better performance than a state-of-the-art software-based solution

High Performance Cache Line Flushing for Persistent Memory

Kai Wu, Ivy Peng, Jie Ren, and Dong Li. "Ribbon: High Performance Cache Line Flushing for Persistent Memory." In 29th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2020.

Lawrence Livermore National Laboratory

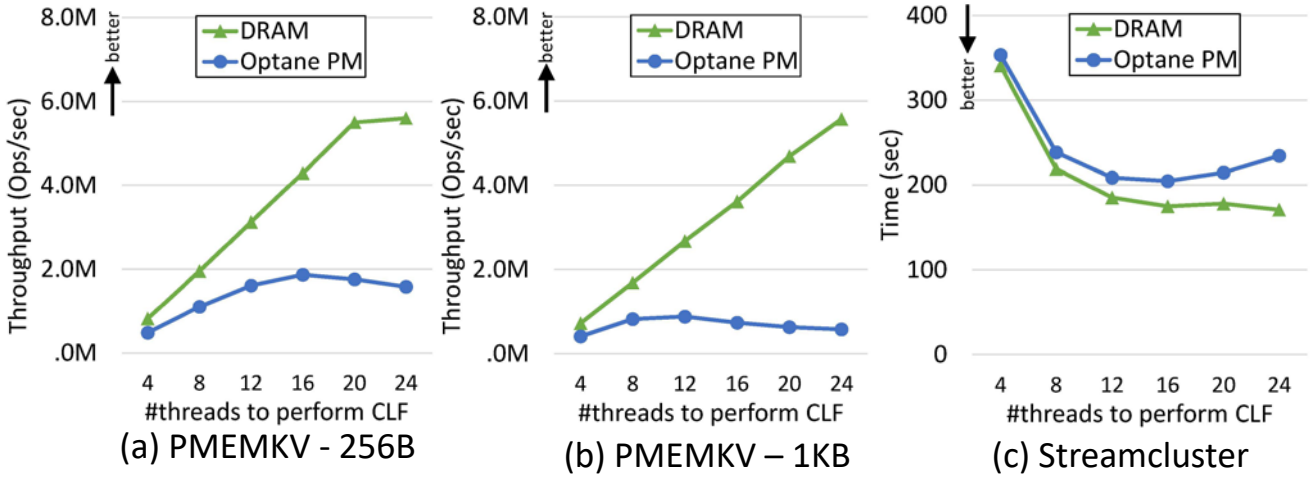
Review of Existing Optimizations for Reducing Cache Line Flushing Overhead on Persistent Memory

- Eager CLF
  - Trigger CLF explicitly at the application-level after the data value is updated (e.g., Intel PMDK, ISCA'14)
- Deferred CLF
  - Group data modifications into failure-atomic intervals and delay CLF to the end of each interval (e.g., OOPSLA'14, DISC'17, Eurosys'17, ASPLOS'19)
- Lazy CLF
  - Rely on natural cache eviction from the cache hierarchy to persist data (e.g., ISCA'18)

Existing methods focus on optimizing CLF policy (i.e., when to use CLF or how to avoid CLF). Need to modify the application code and/or persistency semantics

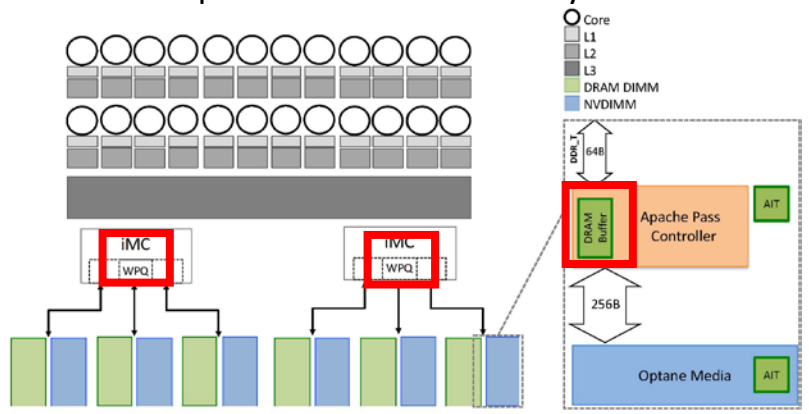
In this work, we focus on the CLF mechanism, instead of CLF policy

The Performance Impact of CLF Concurrency



(a) PMEMKV - 256B (b) PMEMKV – 1KB (c) Streamcluster

The Intel Optane Persistent Memory Architecture

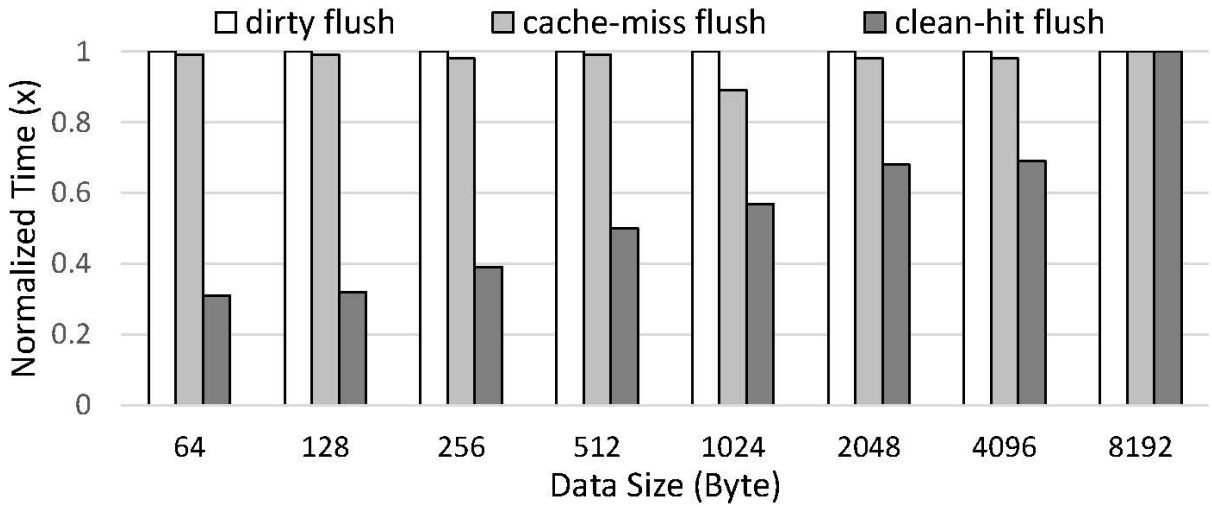


- On DRAM, the performance sustains scaling as the concurrency increases
- On Optane PM, all workloads reach their peak performance at small number of threads, and then the performance starts degrading

- Optane shows lower scalability than DRAM because the contention at the internal buffer of Optane and the WPQ in iMC

How to control the intensity of CLF without affecting application computation? How to determine the appropriate CLF concurrency?

The Performance Impact of Cache Lines Status



- Use micro-bench to measure the cost of flushing dirty (resident) cache lines, non-resident cache lines, and clean resident cache lines
- Flushing a clean cache line could be 3.3x faster than flushing a dirty cache line
- The cost of looking up the whole cache coherence directory is high

How to increase the possibility of flushing clean cache lines? How to capture the PM modifications (stores) associated with the application?

Average Dirtiness of Flushed Cache Lines

- Cache line dirtiness: the ratio of dirty bytes to total number of bytes in a cache line

Workloads	YCSB							TPC-C
	Load	A	B	C	D	E	F	
Dirtiness	0.43	0.55	0.56	0	0.51	0.51	0.47	0.32

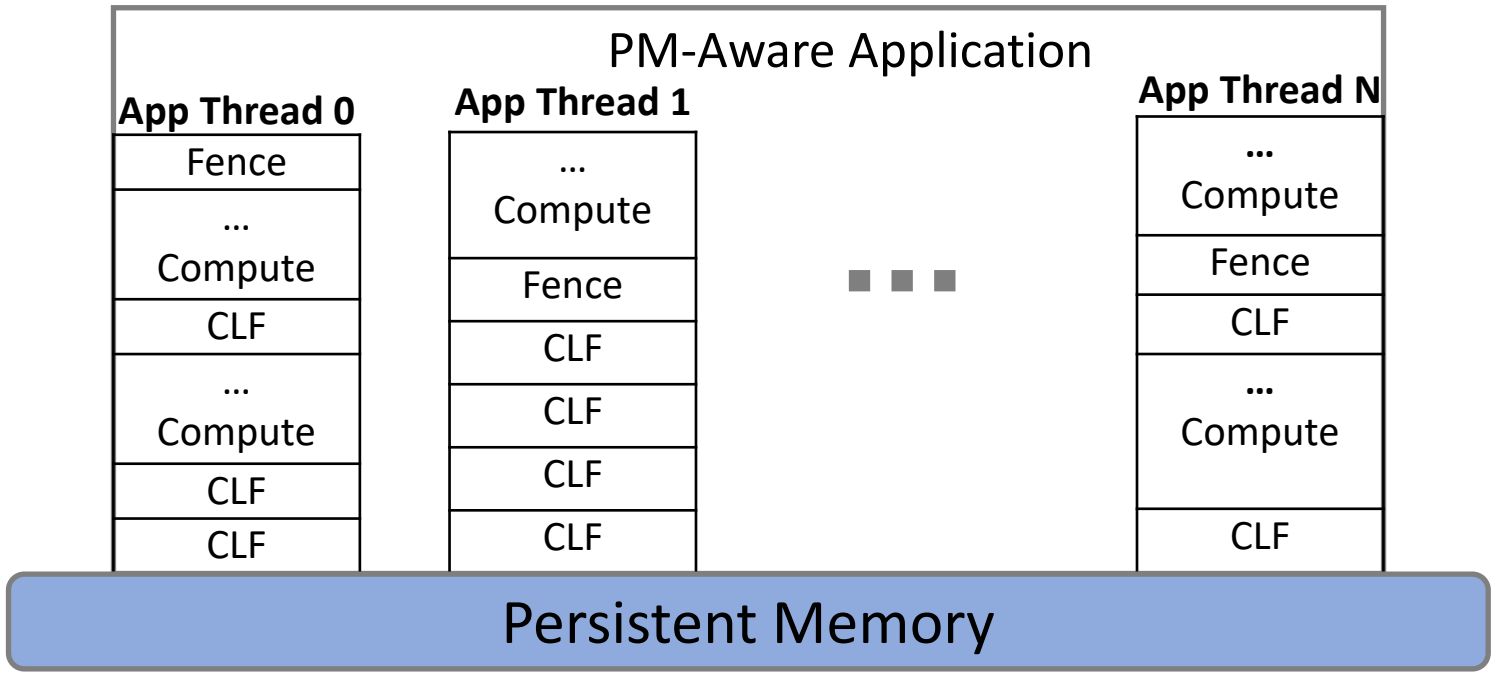
- Run TPC-C and YCSB on Redis
- All workloads have low cache-line dirtiness which is less than 0.6
  - More than 40% of flushed bytes are clean
- Write amplification inside the PM hardware buffer may further increase the number of clean bytes written back to PM
  - If only one byte in four consecutive cache lines is updated, 256 bytes will be eventually written to Optane PM, because the internal transactions have a granularity of 256 bytes

Improving cache line dirtiness could benefit CLF performance on the PM hardware

Our Design

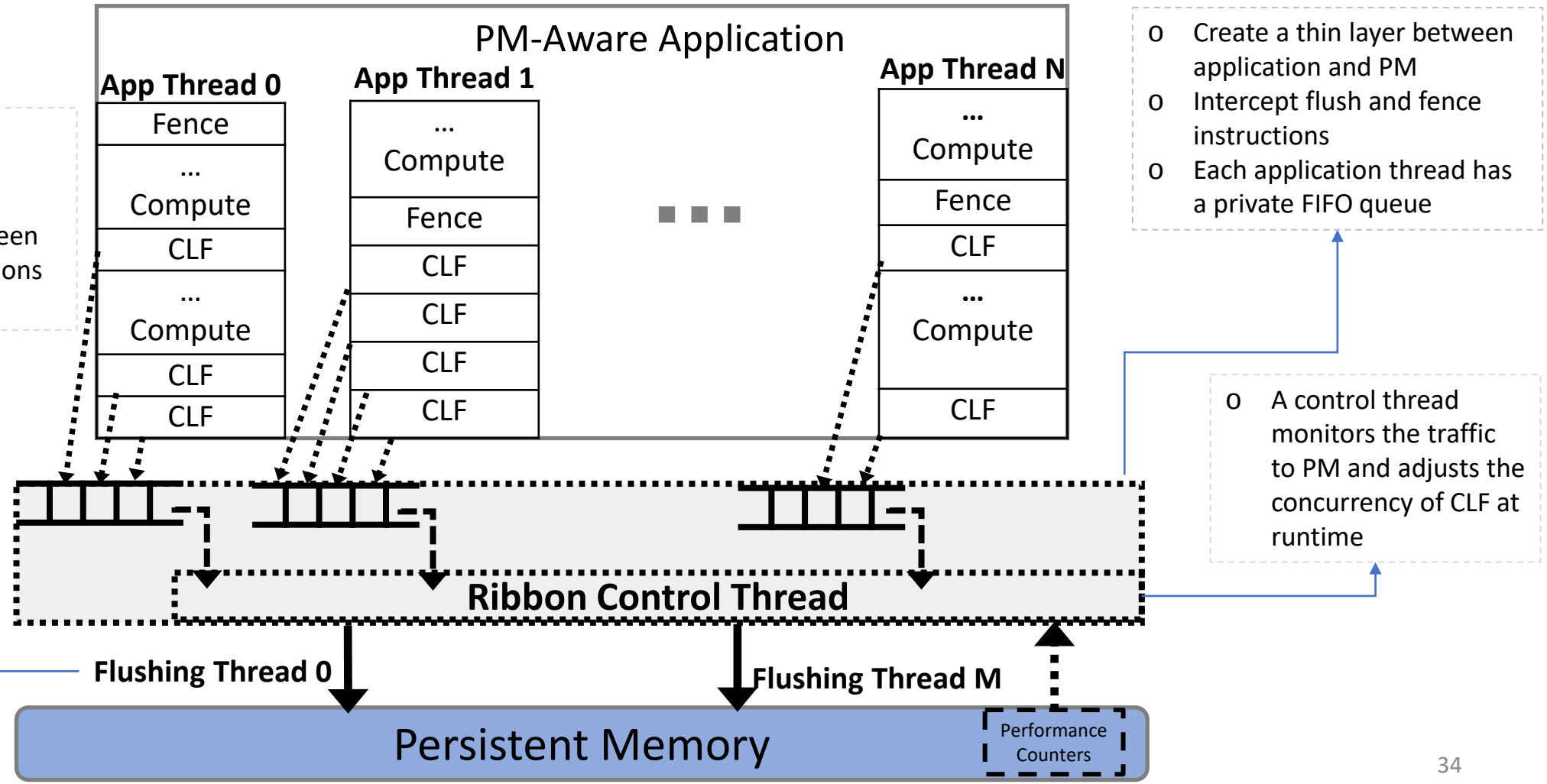
- Ribbon: A lightweight runtime system to manage CLF mechanism
  - Applicable for all workloads
  - No application semantics modification
- Decoupled CLF
  - Propose an adaptive algorithm to select the concurrency level of flushing threads
- Proactive CLF
  - Leverage sampling-based hardware performance counters to opportunistically detect modified cache lines
- Coalescing CLF
  - Identify the reasons of low dirtiness of CLF and introduce an application-specific optimization
  - Find more details in our paper

Decoupled Concurrency Control of CLF



Decoupled Concurrency Control of CLF

- Use a group of flushing threads to execute the intercepted instructions
- Respect the order between flush and fence instructions as in the program order

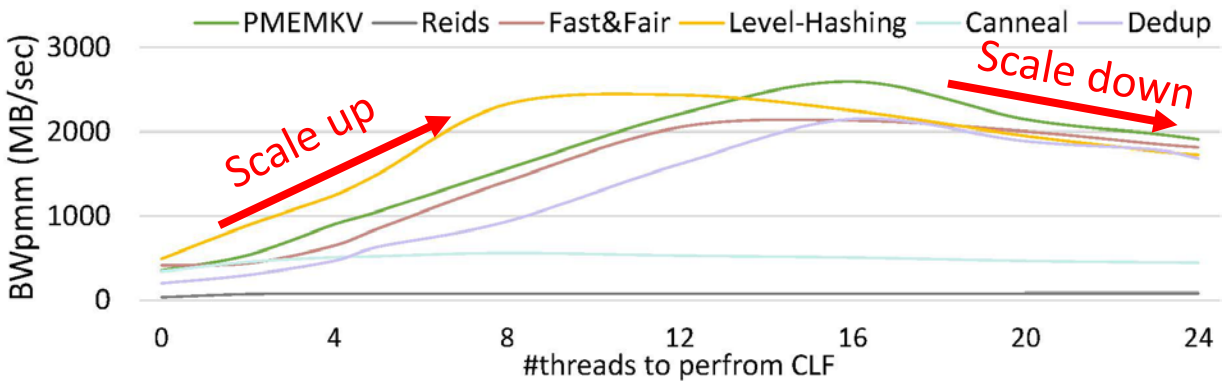


- Create a thin layer between application and PM
- Intercept flush and fence instructions
- Each application thread has a private FIFO queue

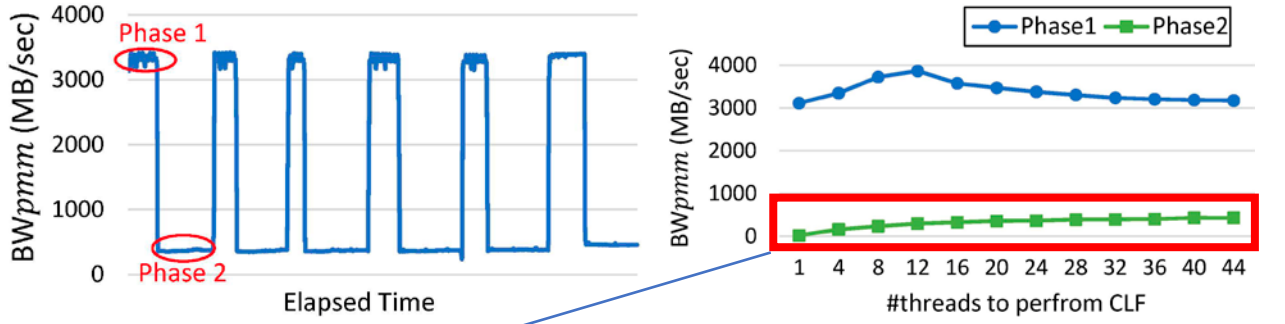
- A control thread monitors the traffic to PM and adjusts the concurrency of CLF at runtime

PM Bandwidth When Running Benchmarks with Various Numbers of Flushing Threads

- We sweep all levels of CLF concurrency in all evaluated workloads and track the write bandwidth to PM
- Run 24 application threads
- All of workloads (except one phase in Streamcluster) exhibit a similar trend



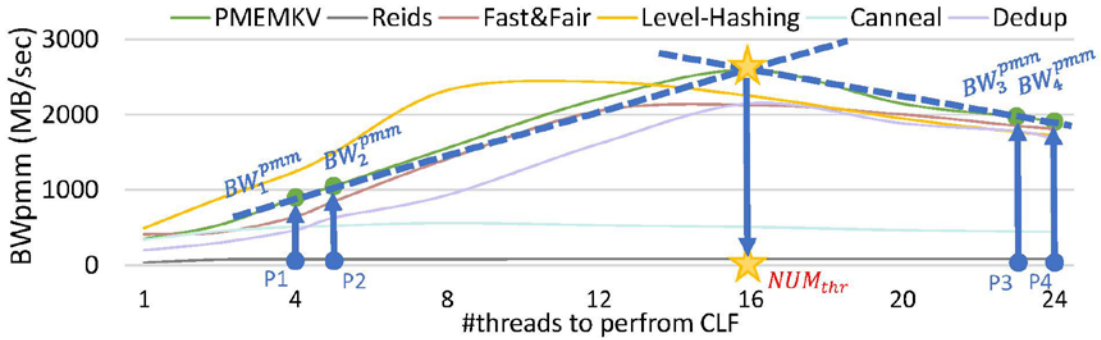
Streamcluster



The second phase of Streamcluster does not enter the contention as its bandwidth continues increasing

Determining the Concurrency Level of CLF (Num_{thr})

- The control thread samples PM bandwidth at four different CLF concurrency levels (P1, P2, P3, and P4)
  - P1 = the number of threads achieves the peak write bandwidth (four on the Optane PM)
  - P4 = the number of cores (24 on the Optane PM)
  - P2 = P1 + 1
  - P3 = P4 - 1



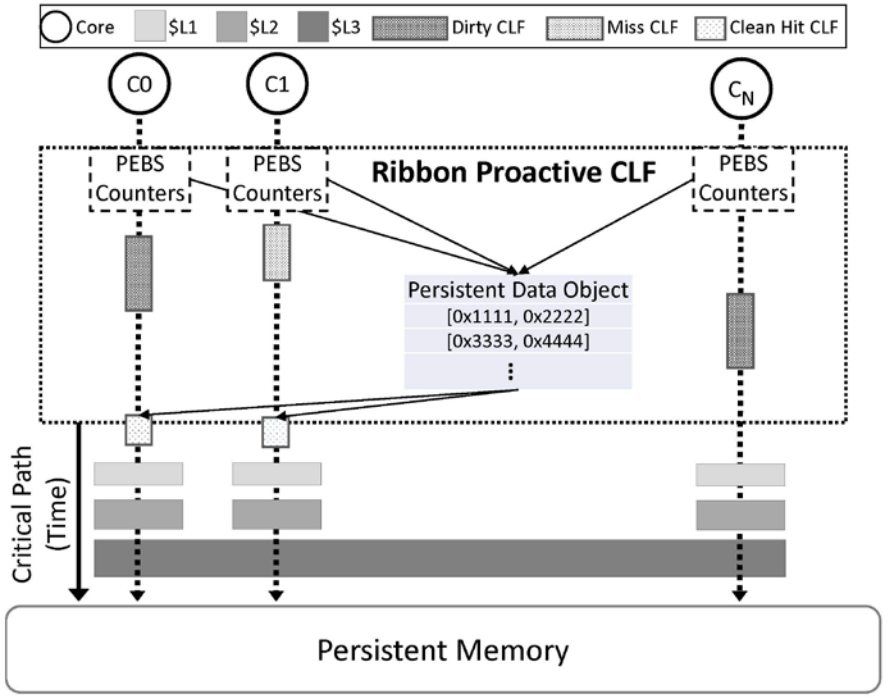
- If $BW_{p2} > BW_{p1}$ && $BW_{p4} < BW_{p2}$ ➔ $Num_{thr} = the\ intersection\ between\ l_{p1,p2}$ and $l_{p3,p4}$
- If $BW_{p2} > BW_{p1}$ && $BW_{p4} > BW_{p3}$ ➔ $Num_{thr} = P4$
- If $BW_{p2} < BW_{p1}$ && $BW_{p4} < BW_{p2}$ ➔ $Num_{thr} = P1$

- The control thread periodically tracks the write bandwidth to PM DIMMs and adjusts the concurrency of CLF at runtime

P4

Proactive Cache Line Flushing

- Leverage the precise address sampling capability in hardware performance counters to collect the virtual memory addresses of store instructions
- Use a background thread to proactively flush cache lines to transform cache lines to clean state
- Reduce the time spent on the critical path of the application executing the CLF



Performance Evaluation

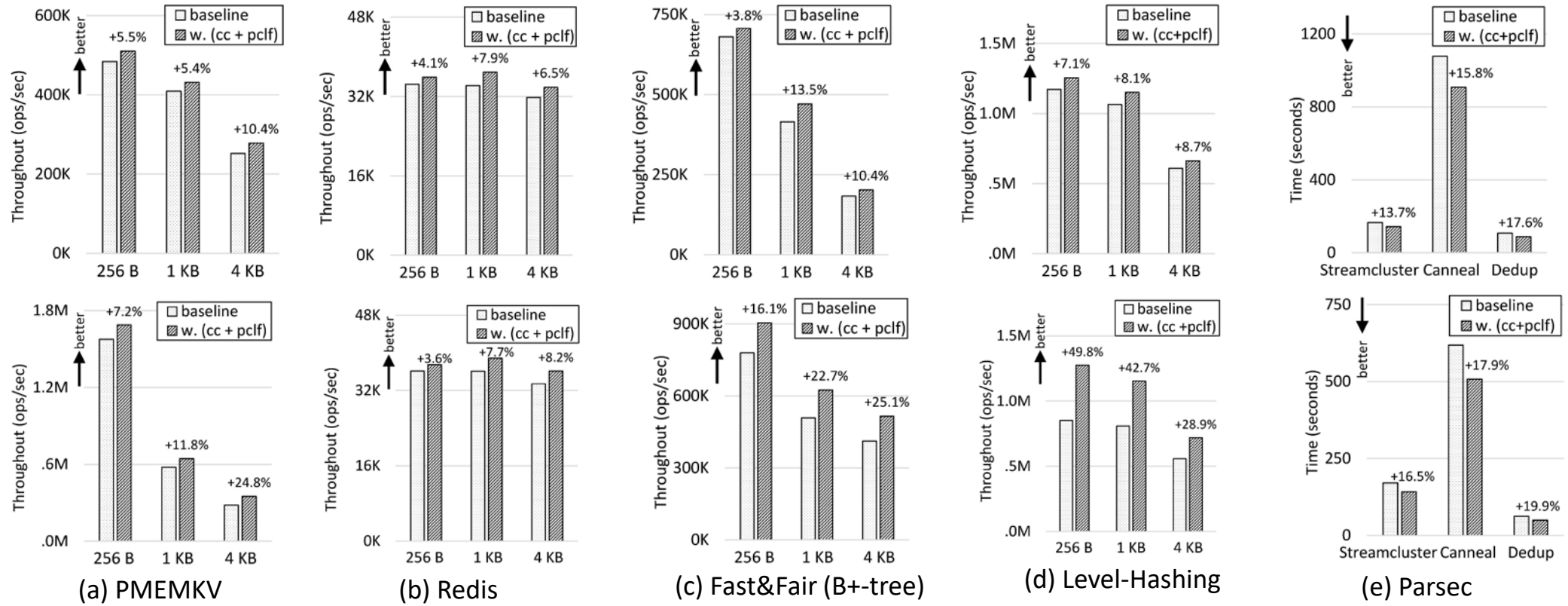
- Real PM platform (Intel Optane DC PMM)
  - 2nd Gen Intel Xeon Scalable processor (2 * 24 cores)
  - 192 GB DRAM and 1.5 TB PM
- Evaluate seven representative PM-aware workloads

Applications	Program Type	PM Access Layer	Persistency
PMEMKV [Intel]	Database	Library/Intel PMDK (undo/redo)	Eager
Redis	Database	Library/IntelPMDK (undo/redo)	Eager
Fast & Fair (B+-tree)[FAST'18]	PM-aware index	Native (add custom assembly instructions)	Customized
Level-Hashing [OSDI'18]	PM-aware index	Native (add custom assembly instructions)	Customized
Streamcluster [Parsec]	Lock-based parallel code	Library/HP Nvthread (redo) [Eurosys'17]	Deferred
Canneal [Parsec]	Lock-based parallel code	Library/HP Nvthread (redo) [Eurosys'17]	Deferred
Dedup [Parsec]	Lock-based parallel code	Library/HP Nvthread (redo) [Eurosys'17]	Deferred

Overall Performance

App threads = 4

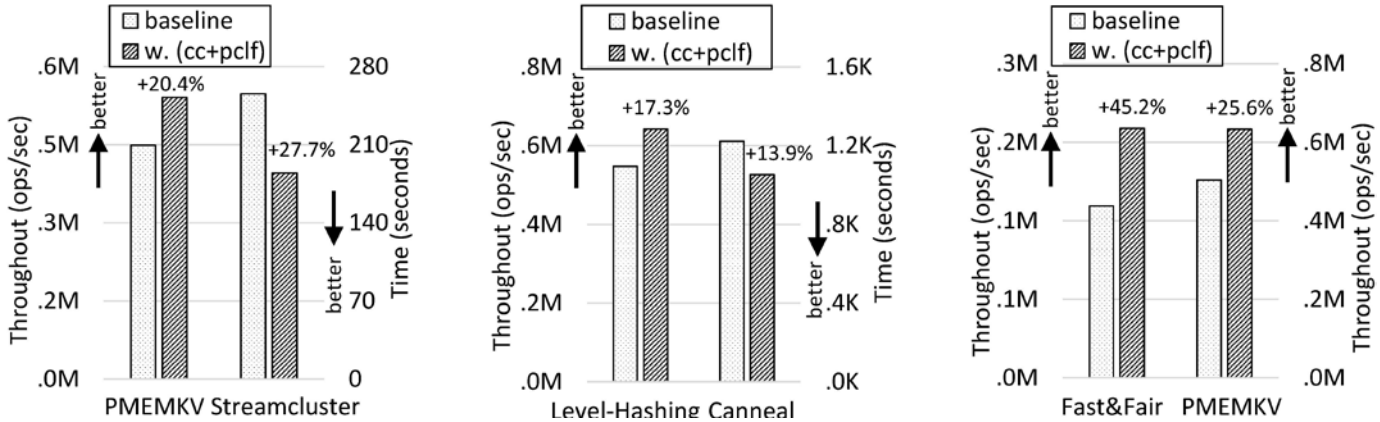
App threads = 24



(a) PMEMKV (b) Redis (c) Fast&Fair (B+-tree) (d) Level-Hashing (e) Parsec

- At four application threads, Ribbon achieves up to 17.6% improvement (9.3% on average)
- At 24 application threads, Ribbon improves the performance by up to 49.8% (20.2% on average)

Heavily Loaded System Evaluation



- Co-run three different application combinations (two applications each combination)
- Each application use 24 application threads
- Ribbon significantly improve the performance from 13.9% to 45.2%

Summary

- We characterize the performance of the CLF mechanism in PM-aware workloads on Optane
- We propose decoupled concurrency control, proactive CLF, and cache line coalescing to improve performance of the CLF mechanism
- We design Ribbon, a runtime to optimize PM-aware applications automatically
- We evaluate Ribbon on a variety of PM-aware workloads and achieve up to 49.8% improvement (14.8% on average)

Thank you! Question?