Future Server Platforms: Persistent Memory for Data-intensive Applications

Sudarsun Kannan,

Ada Gavrilovska, **Karsten Schwan**

CERCS Research Center

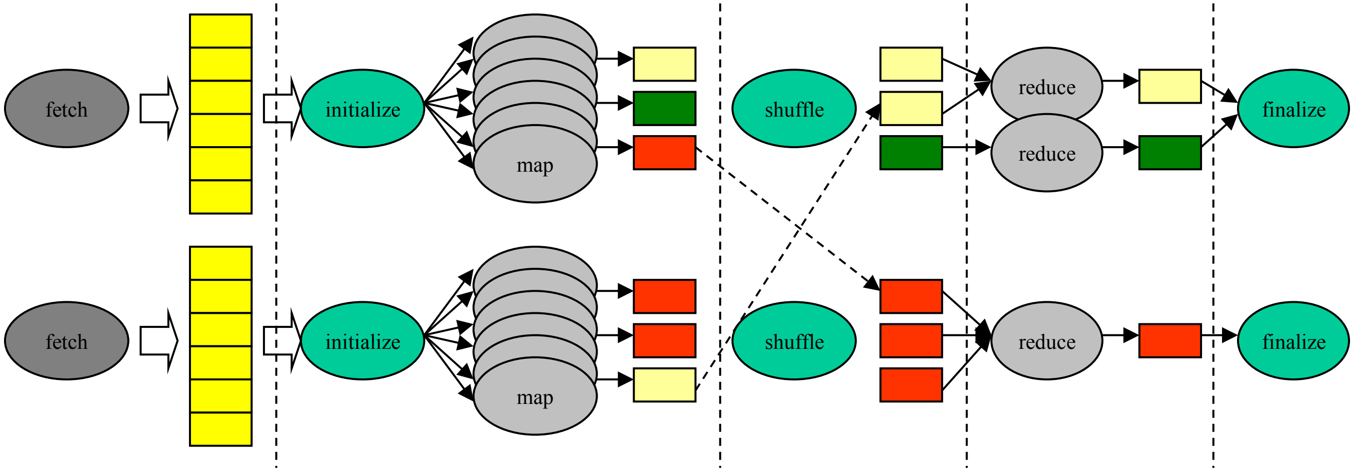
Georgia Institute of Technology

Background – ‘Big Data’ Research

– **Scientific/Technical Computing – *Scalable, Reliable Data Access:***

- *High Performance I/O:* in-situ and in-transit processing for HPC I/O:
  - **DOE SDAV, ExACT, SDM** awards **(ORNL, LBL); DOE Sandia** joint work on resilience (IPDPS12, SC13, …).
- *Heterogeneous multicore platforms* (+accelerator-based systems):
  - **DOE** ExaOS award (**Sandia, ORNL, LBL**); **Intel** NVM award (on clients); additional collab. with **Microsoft** (on servers); **HP Labs** collaboration (HPCA13, TRIOS13, …).

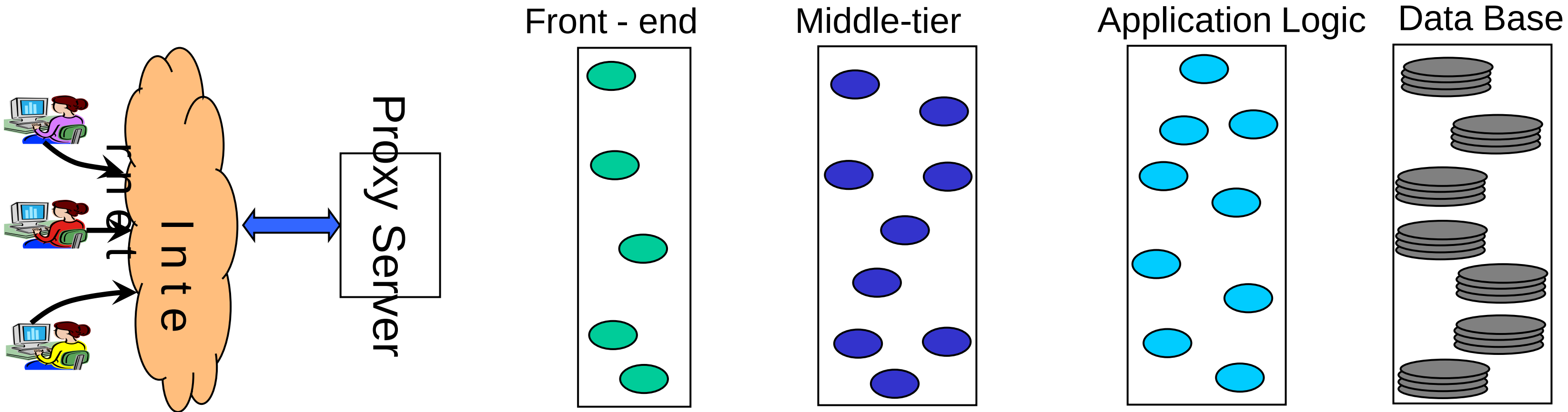
In-situ
&
In-transit
data
processing



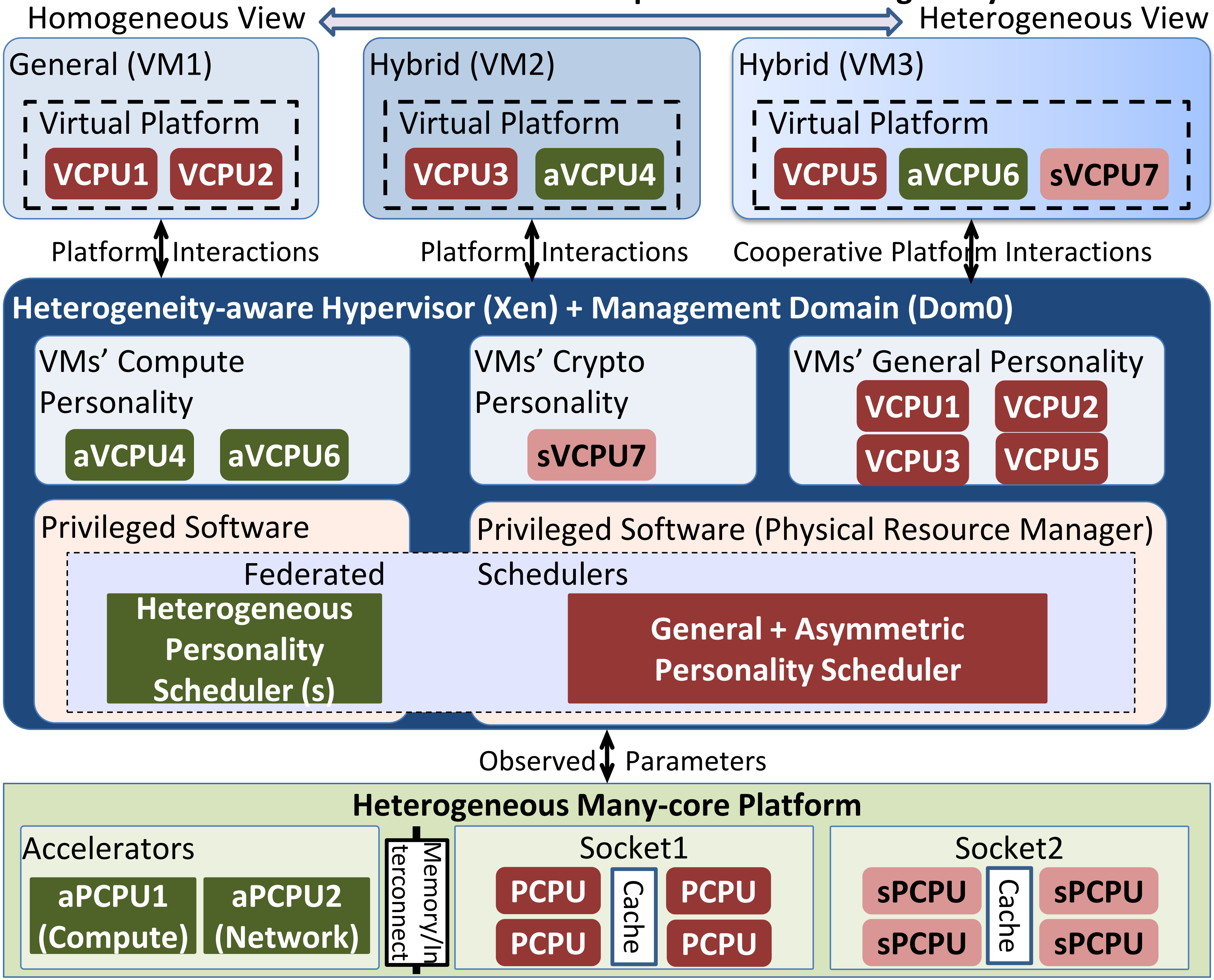
Background – cont.

– Enterprise and Cloud Computing – *Fast Data:*

- *QoS Clouds* (IPDPS13); I/O virtualization and IB bypass (Cluster 13).
- *'Monalytics'*: real-time data monitoring and analytics; online troubleshooting; scalable Flume-based benchmark suite; annotated biblio on troubleshooting (Middleware13, ACM SigOps14, ICAC 14, …). Scalable Flume-based data streaming benchmark.
- *ELF*: ELastic Fast data processing (ATC 14); *Nectere* benchmark.
- *Data-intensive applications on GPUs:*
  - SQL operators on GPUs (Yalamanchili); PGAS for extended memory (Cluster13, GPGPU13, GPGPU14, CGO14, …).
- **Note:** Big Data management track in ICAC conference (with **HP**).



Future Server Platrforms - Spectrum of Heterogeneity



Hetero Processors => Hetero Memory

Growth in data intensive applications, coupled with increased node core counts and thread parallelism

Demands *increased on-node memory capacities*:

Toward exascale systems:

- Science simulations using experimental data or co-running with online data analytics

Next generation server and cloud platforms:

- Heterogeneous server nodes for perceptual and cognitive applications (zillians, …)

End client devices:

- Apps. with rich features and data, enabled by Machine Learning, Graph Processing, MR, …

Motivation

Power (or Battery) constraints, along with cost, prevents use of DRAM to address
all of these needs.

SSD/Nand-Flash remains comparatively slow, so …

Non-Volatile Memory (NVM) to the Rescue

NVM (e.g., PCM) is byte addressable
Provides persistence – 100x faster than SSD
Higher density compared to DRAM (~128GB)
NVM as low power memory in future Exascale Platforms

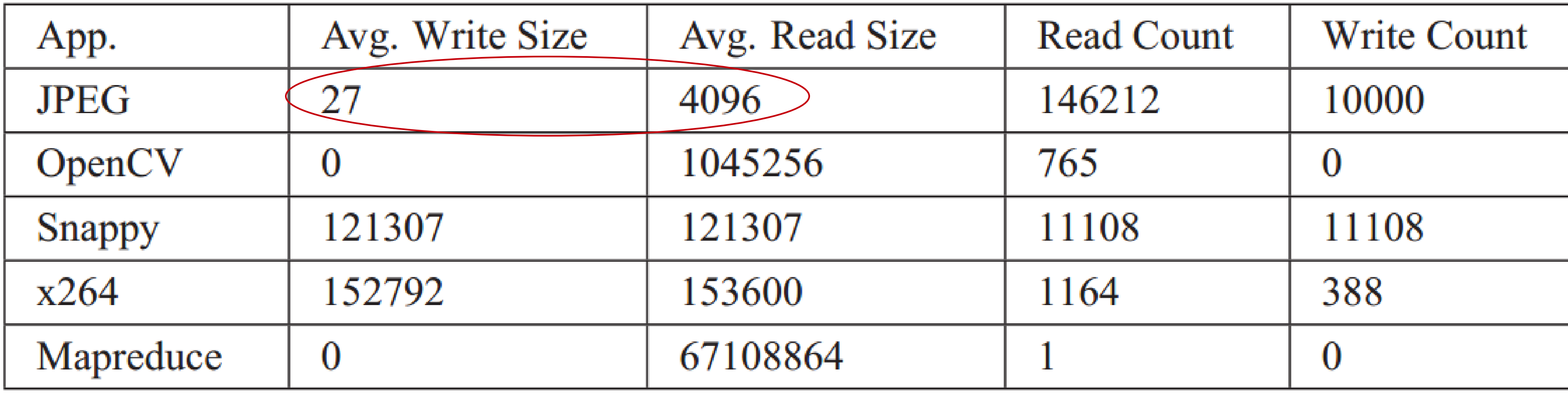
- Technical approach:
  - Use NVM `as memory' vs. `as storage' (under the I/O stack): to enable rapid 'compute' on data
  - Use processor caches to reduce write latency impact and improve endurance (4x-10x slower writes, and limited endurance (~10^8 writes))
  - `Enable' applications to use NVM + `NVM-aware' systems

NVM: Why use it as ‘Memory’?

Example: end client devices and applications

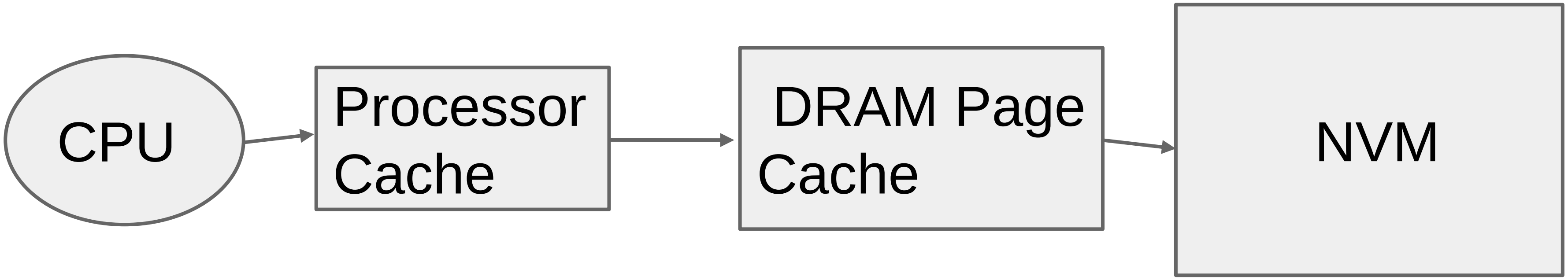
NVM (and/or the slower SSD devices) used via I/O APIs:

- High software overheads for block-based I/O interfaces
- Low per call data sizes, hence more calls
- Just using 'mmap‘ not sufficient:
  - every mmap/munmap call implies a user/kernel transition
  - requires multiple supporting POSIX calls like open, close



App.	Avg. Write Size	Avg. Read Size	Read Count	Write Count
JPEG	27	4096	146212	10000
OpenCV	0	1045256	765	0
Snappy	121307	121307	11108	11108
x264	152792	153600	1164	388
Mapreduce	0	67108864	1	0

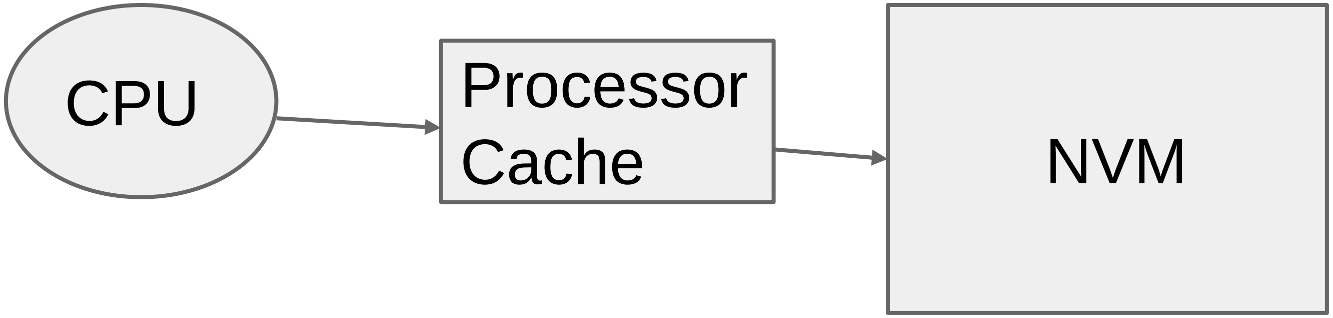
NVM as Memory:
Prior Work: DRAM as Cache



DRAM acts like a page cache

Good for addressing `capacity only’ needs
May work well for server machines with TBs of DRAM
Power/performance issues for exascale or end client codes

Alternative: Fast Non-Volatile Heap



To Make Persistence Guarantees:

Frequent cache flushing, memory fencing, writes to PCM
High persistence management overheads
    Includes user and kernel level overheads

Our Approach: pMem: Dual-Use NVM

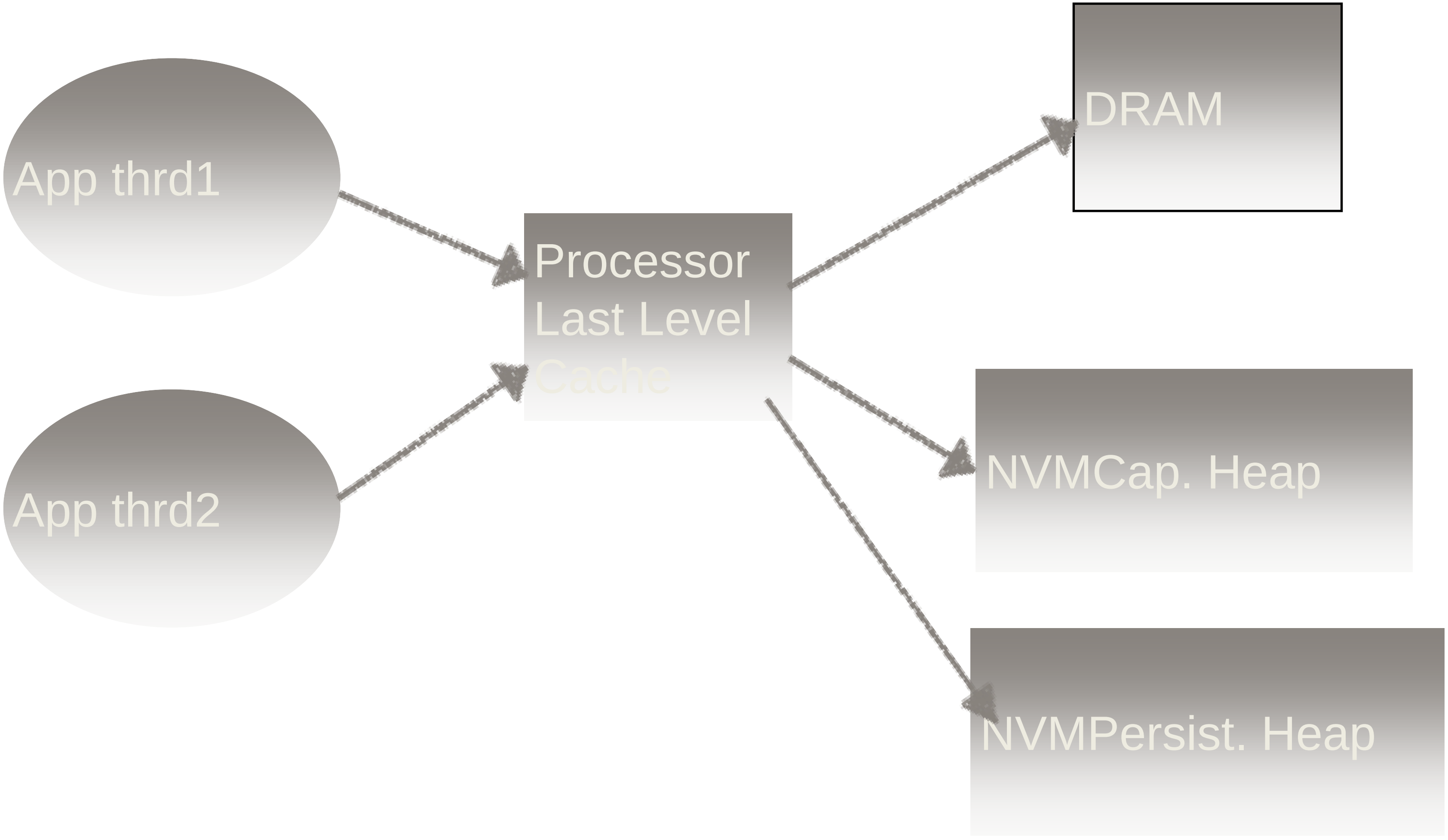
Prior Research: use NVM **either** for persistence **or**
as an additional `capacity heap’

pMem: use NVM for persistence **and** for add’tl capacity

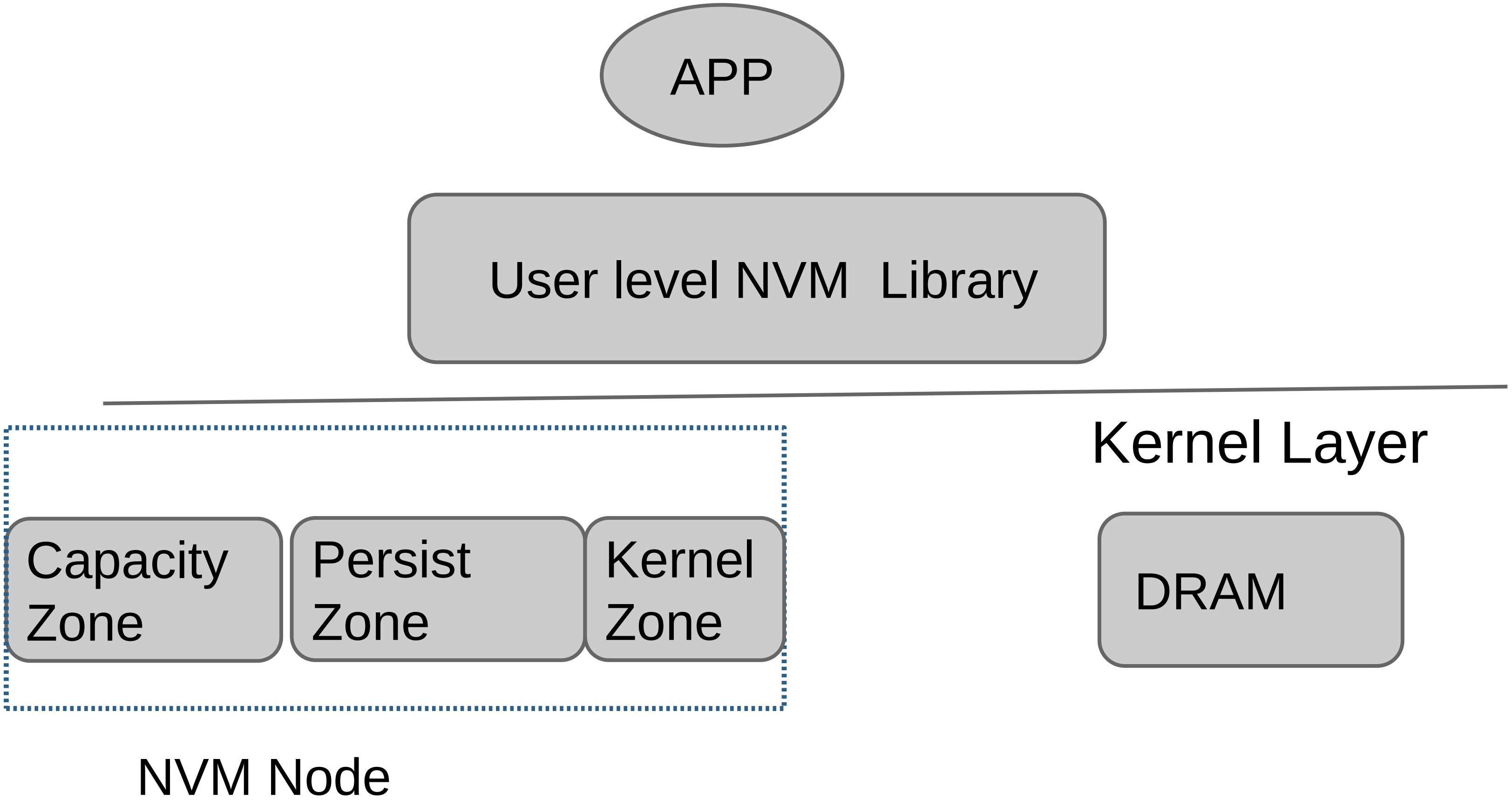
NVM for persistence - *“NVMPersist”*
NVM for capacity as a heap - *“NVMCap”*

NVMCap and NVMPersist threads share use of the same last level cache

NVM Dual Use – High Level View

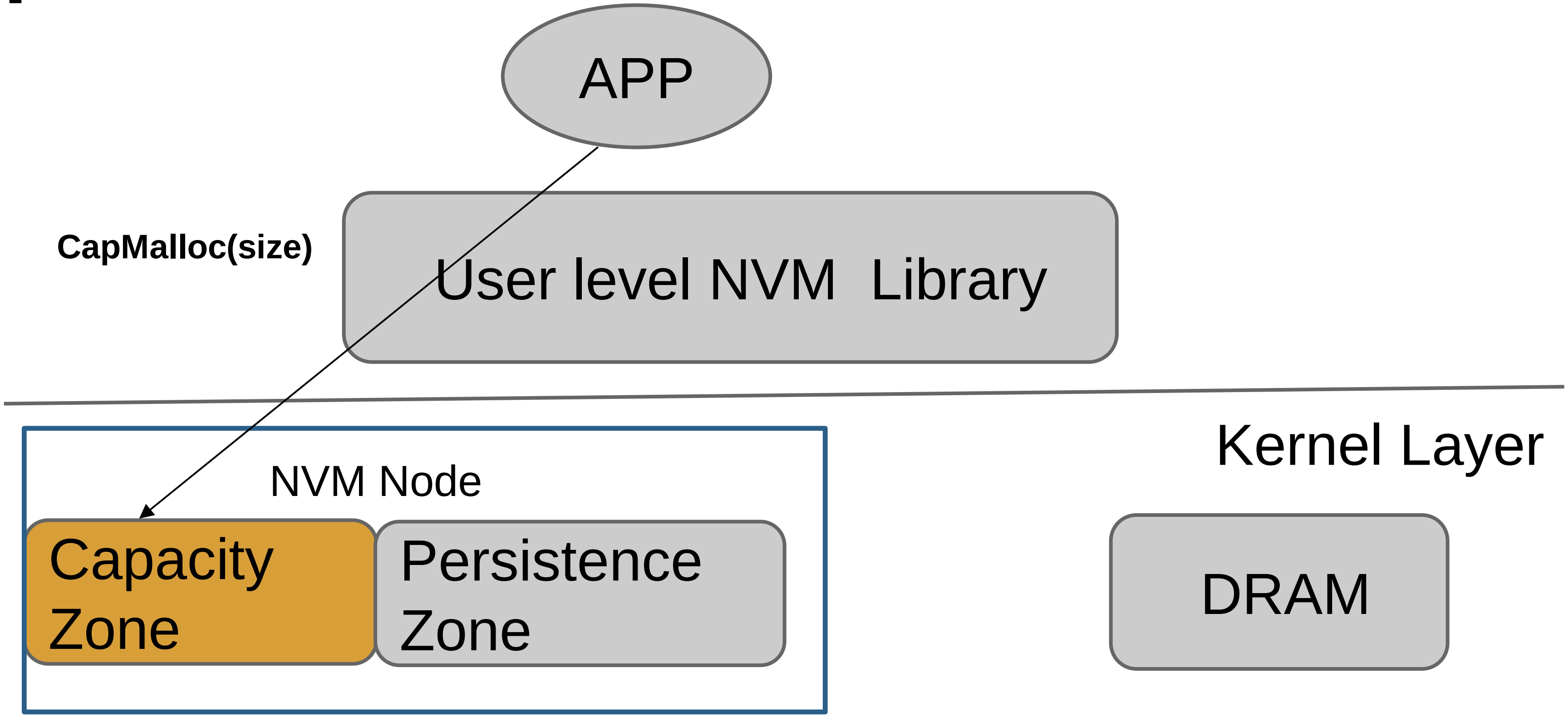


NVM-pMem: Dual Use Interface

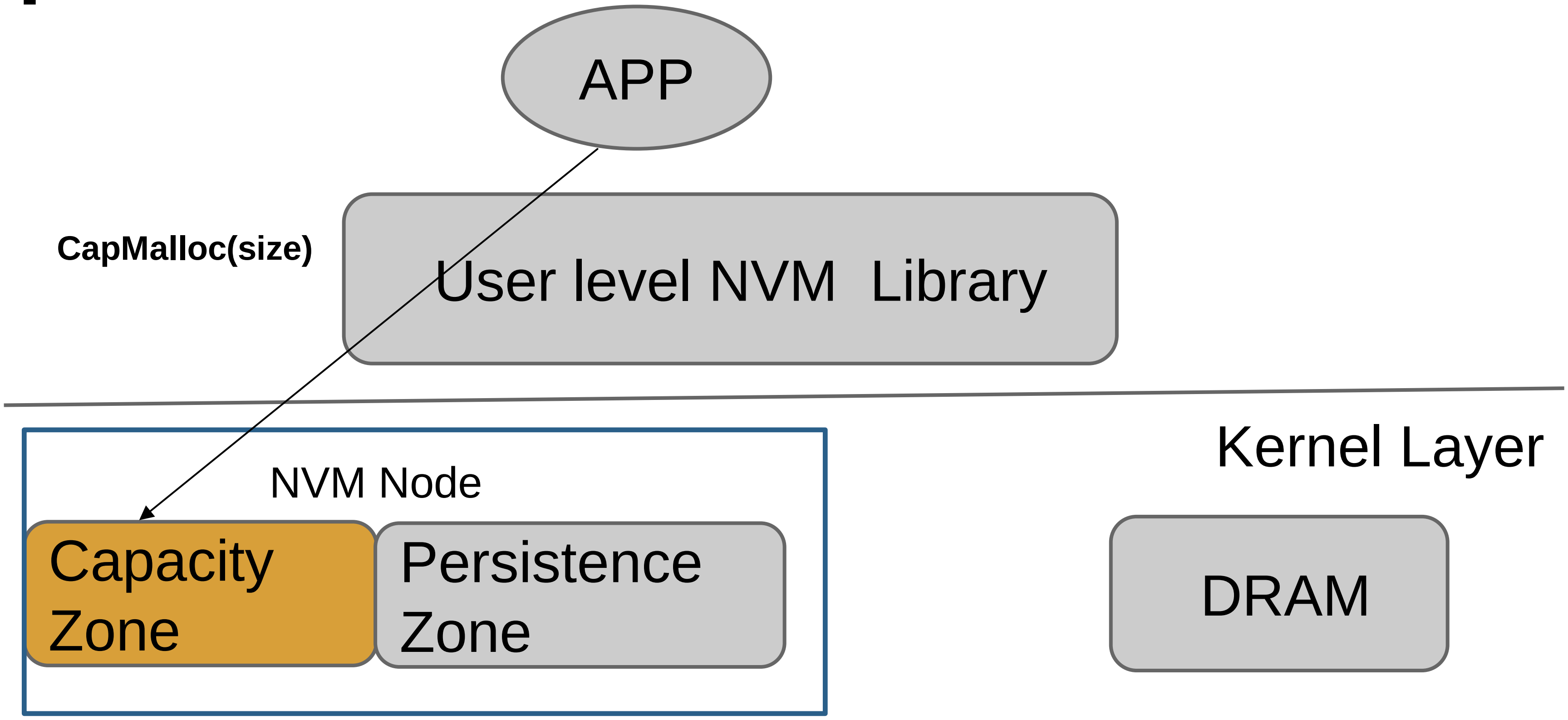


NVM is with partitioned capacity & persistence zones

pMem: Dual Use NVM



pMem: Dual Use NVM



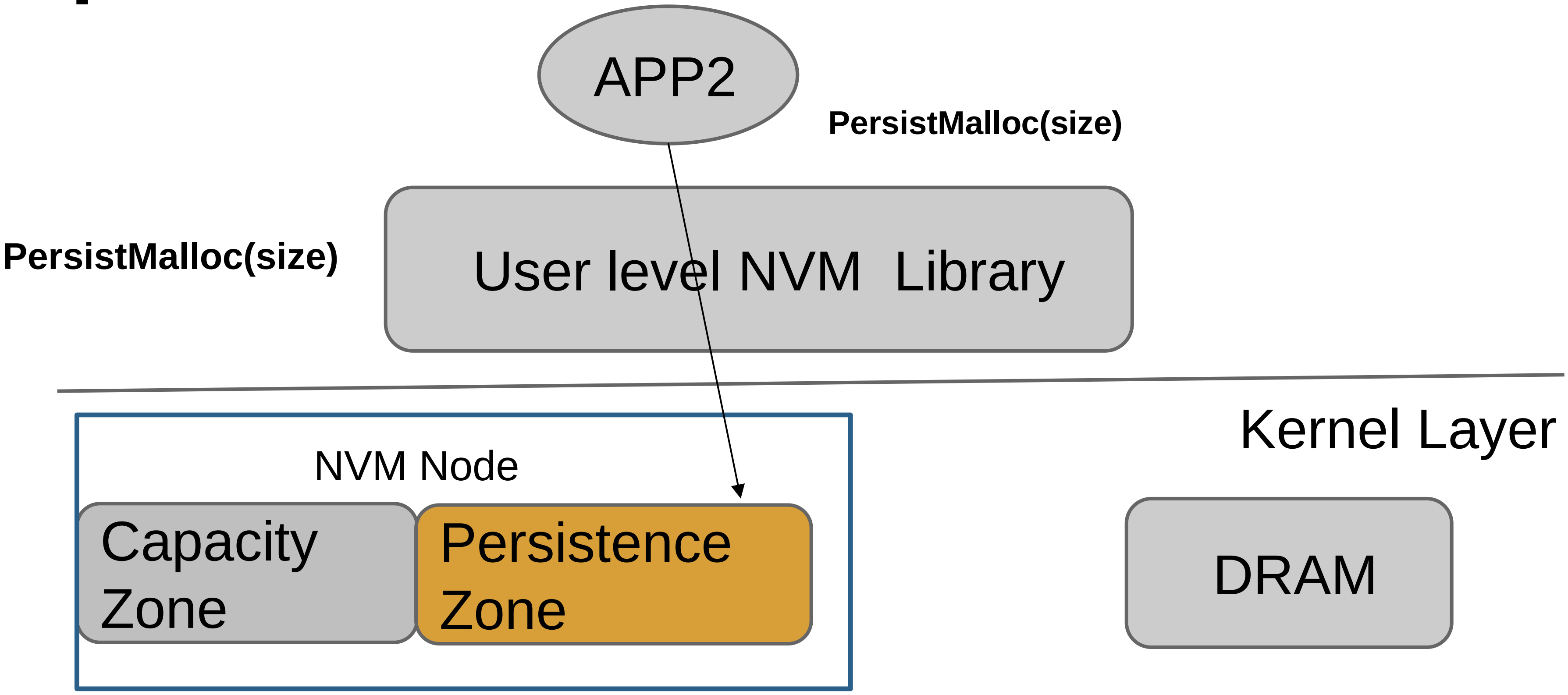
Application decides when to use NVM for capacity

NVM used as heap without persistence

User level and kernel managers route application calls

Think of persistent metadata as a light weight file system metadata.

pMem: Dual-Use NVM



pMem: Dual-Use NVM

Key Ideas:

- Application-level control:
  - Suitable library-based interfaces for p vs. np data
  - Expensive I/O calls replaced with ‘memory’ accesses
  - Goal: Reduced software use (includes OS)
- System-level:
  - Deploy NVM as OS `memory node’
  - NVM ‘node' partitioned into volatile + persistent heap
  - NUMA-like kernel allocation policies

Advantages

- Dual-benefit NVM: capacity + fast persistence

Enabling Persistence Support

```
hash *table = PersistAlloc(entries, "tableroot");
 for each new entry:
  entry_s *entry = PersistAlloc (size, NULL);
   table =  entry;
   count++;
    temp_buff = CapAlloc(size);
```

Requires persistence metadata in library & OS

No persistent metadata required

Plus the following additional requirements:

flush app. data cache to avoid loss on power failure

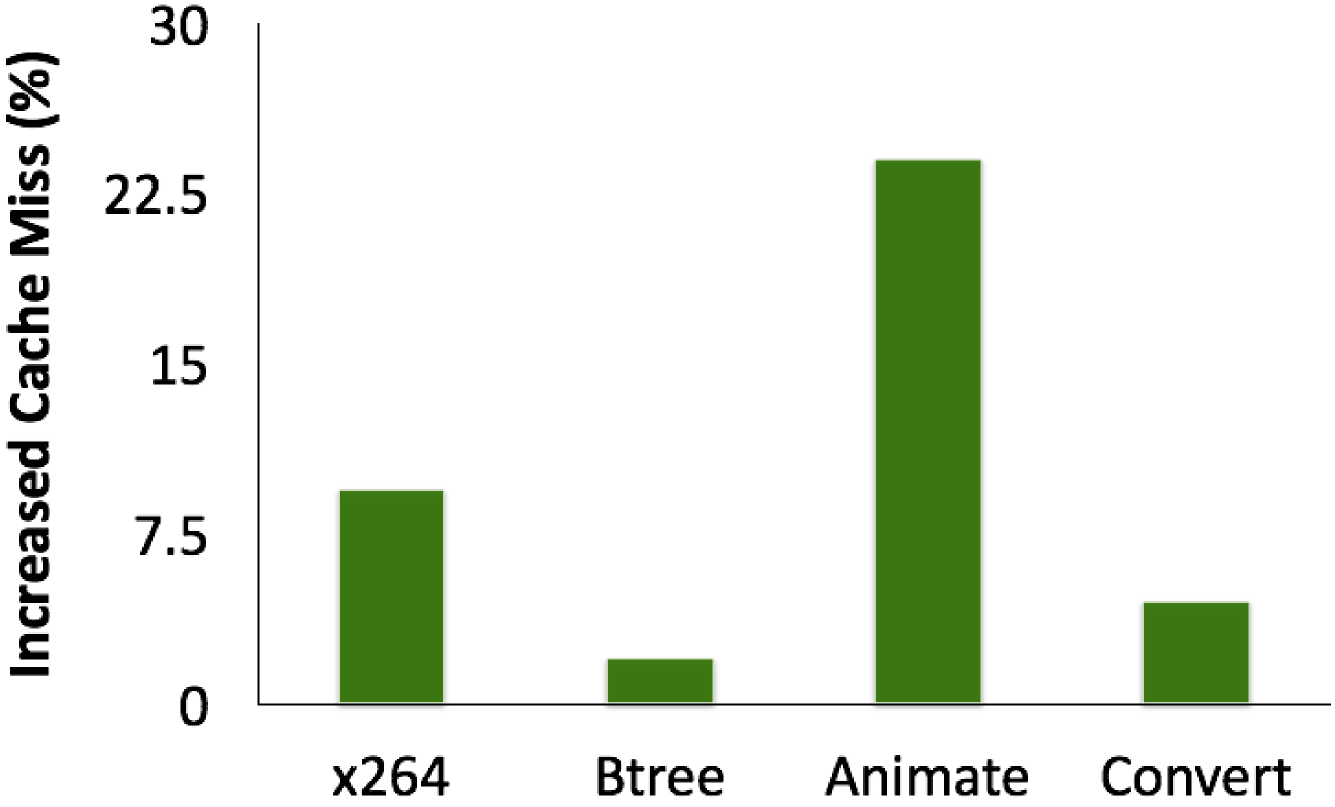
flush OS data-structures and library metadata

Dual-Use Solution Challenges: `Persistence Impact’ on NVMCap

- `Persistence-unaware’ OS page allocations cause *cache conflicts* between NVMCap and NVMPersist
- Persistent library allocator metadata maintenance *increases flushes*
  - Increases NVMCap cache misses for shared data
- Transactional (durability) logging of persistent application state *increases flushes and NVM Writes*

Persistence Increases #Cache Misses

- Atom platform with 1MB LLC
- MSR counters to record LLC misses

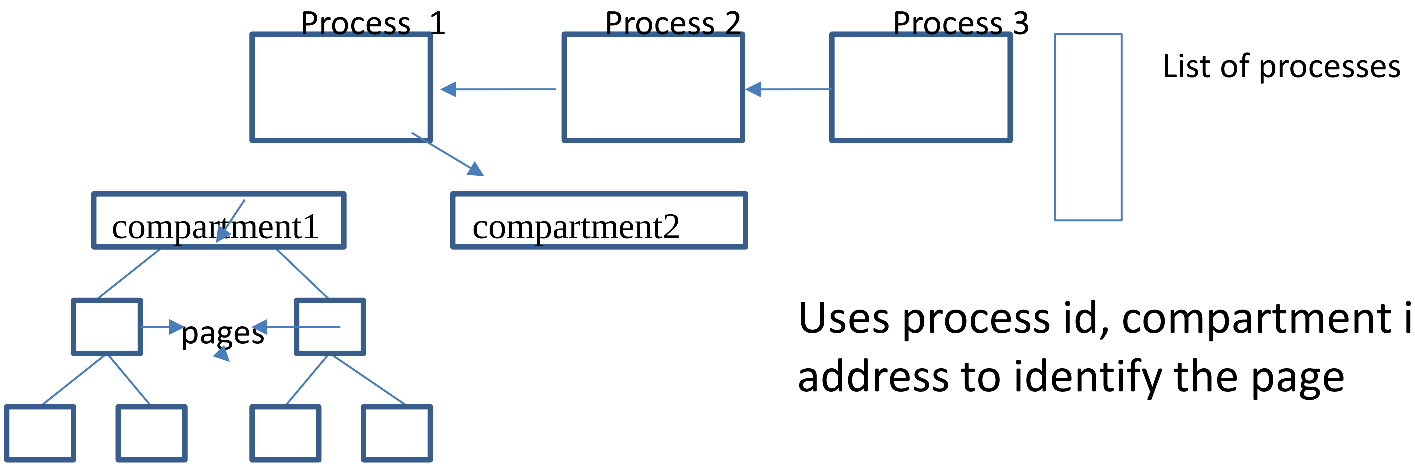


pMem Implementation: Emulated NVM Node for Persistence

On boot, configure an OS memory node to emulate NVM

- All NVM pages locked, swapping disabled
- Provides persistence across application sessions
- For persistence across boots, write it to SSD
- Paging uses allocate on write policy
- Cache line flushes for user level data and for kernel data-structures

pMem Implementation - Kernel



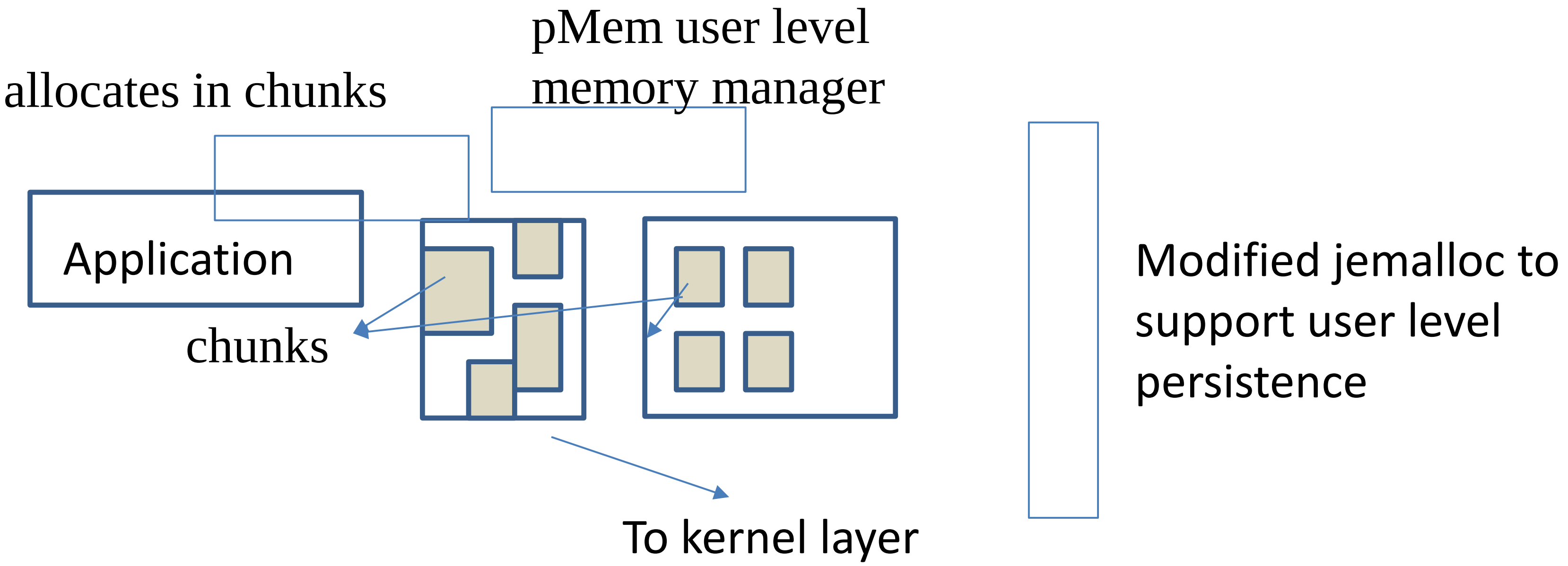
Uses process id, compartment id and fault address to identify the page

1 bit for each NVM page flag and 1 bit flush flag

Compartments:

- large region of NVM allocated by user-level NVM manager using *nvmmap*
- they are virtual memory area structures (VMA)
- apps. can explicitly request separate compartments (‘nvmmap’)
- isolates persistent from non-persistent NVM regions

pMem Software Architecture - Allocator


pMem user level memory manager
allocates in chunks
Application
chunks
To kernel layer
Modified jemalloc to support user level persistence

- Provides application interfaces like “capmalloc”, “persistmalloc”, logging, and application transparent non-persistent allocations
- Manages application data in chunks
- Implemented by extending the jemalloc library

Methods for Cache Conflict Reduction

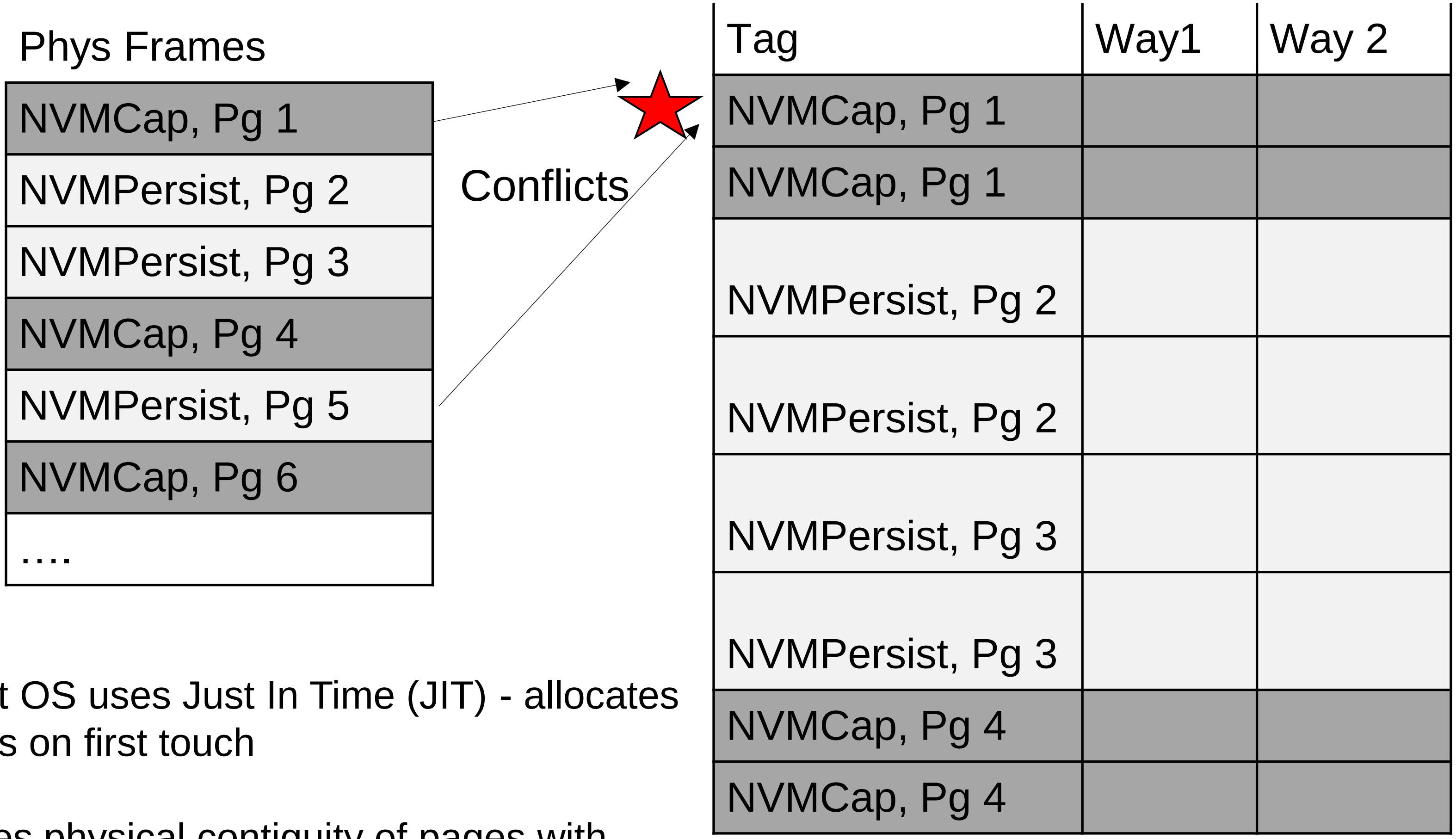
Co-running NVMPersist and NVMCap increases cache conflicts

- Solution: Cache partitioning

- Hardware techniques: little flexibility
- Software techniques: page coloring complex (FreeBSD)
  - Focused on allocating physically contiguous pages to application

- Software-based partitioning – ‘page bucket’ solution to allocating persist vs. cap pages

Conflict Unaware JIT Allocation



Phys Frames
NVMCap, Pg 1
NVMPersist, Pg 2
NVMPersist, Pg 3
NVMCap, Pg 4
NVMPersist, Pg 5
NVMCap, Pg 6
….

Conflicts

Tag	Way1	Way 2
NVMCap, Pg 1		
NVMCap, Pg 1		
NVMPersist, Pg 2		
NVMPersist, Pg 2		
NVMPersist, Pg 3		
NVMPersist, Pg 3		
NVMCap, Pg 4		
NVMCap, Pg 4		

Current OS uses Just In Time (JIT) - allocates pages on first touch

Reduces physical contiguity of pages with increasing no. of threads

Ideal Conflict-Free Allocator

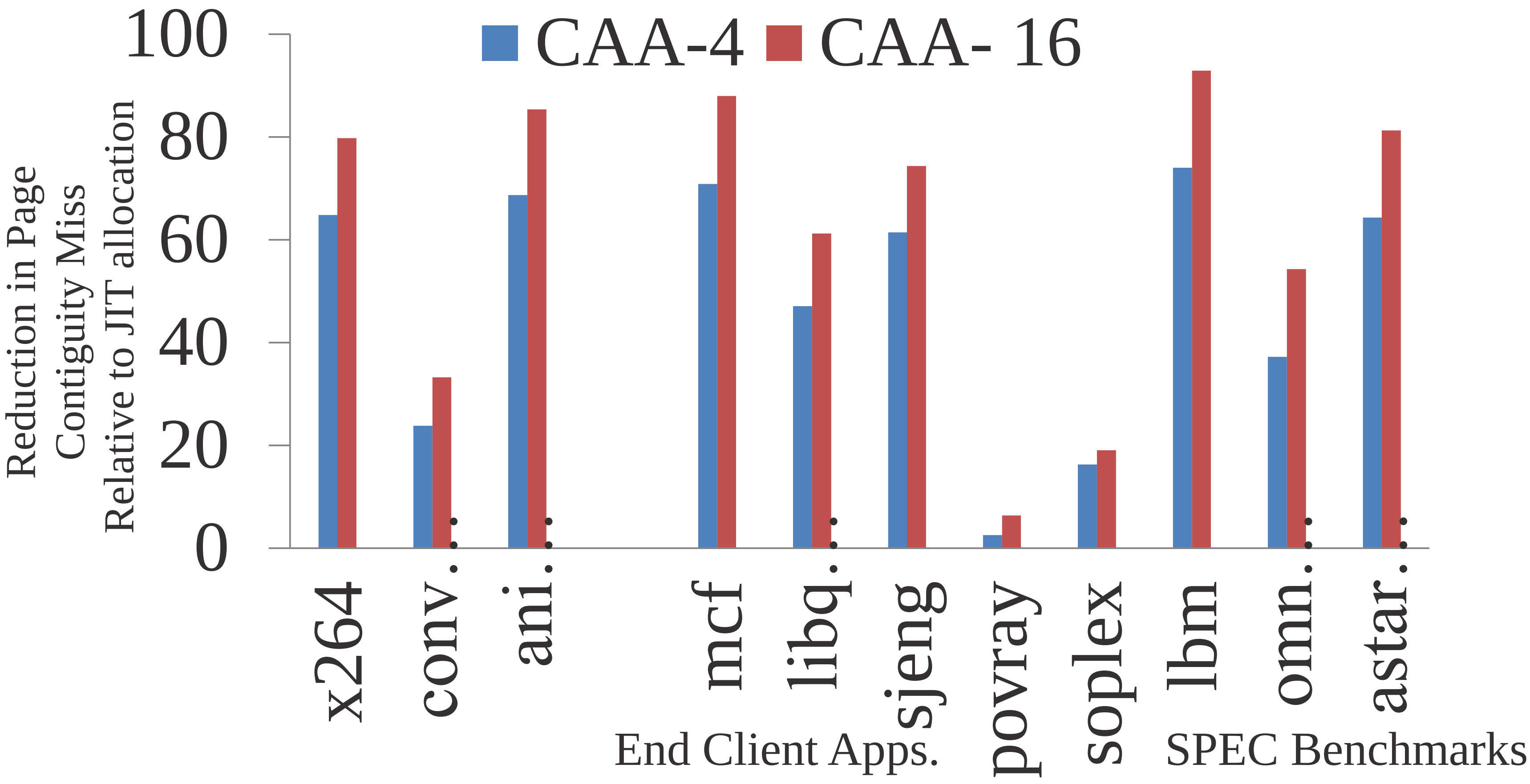
Phys Frames
NVMCap, Pg 1
NVMCap, Pg 2
NVMCap, Pg 3
NVMPersist, Pg 4
NVMPersist, Pg 5
NVMPersist, Pg 6
….

No Conflicts

Tag	Way1	Way 2
NVMCap, Pg 1		
NVMCap, Pg 1		
NVMCap, Pg 2		
NVMCap, Pg 2		
NVMCap, Pg 3		
NVMCap, Pg 3		
NVMPersist, Pg 4		
NVMPersist, Pg 4		

- ✦ Physically contiguous page allocation reduces conflicts
- ✦ We propose a simple design to achieve contiguity

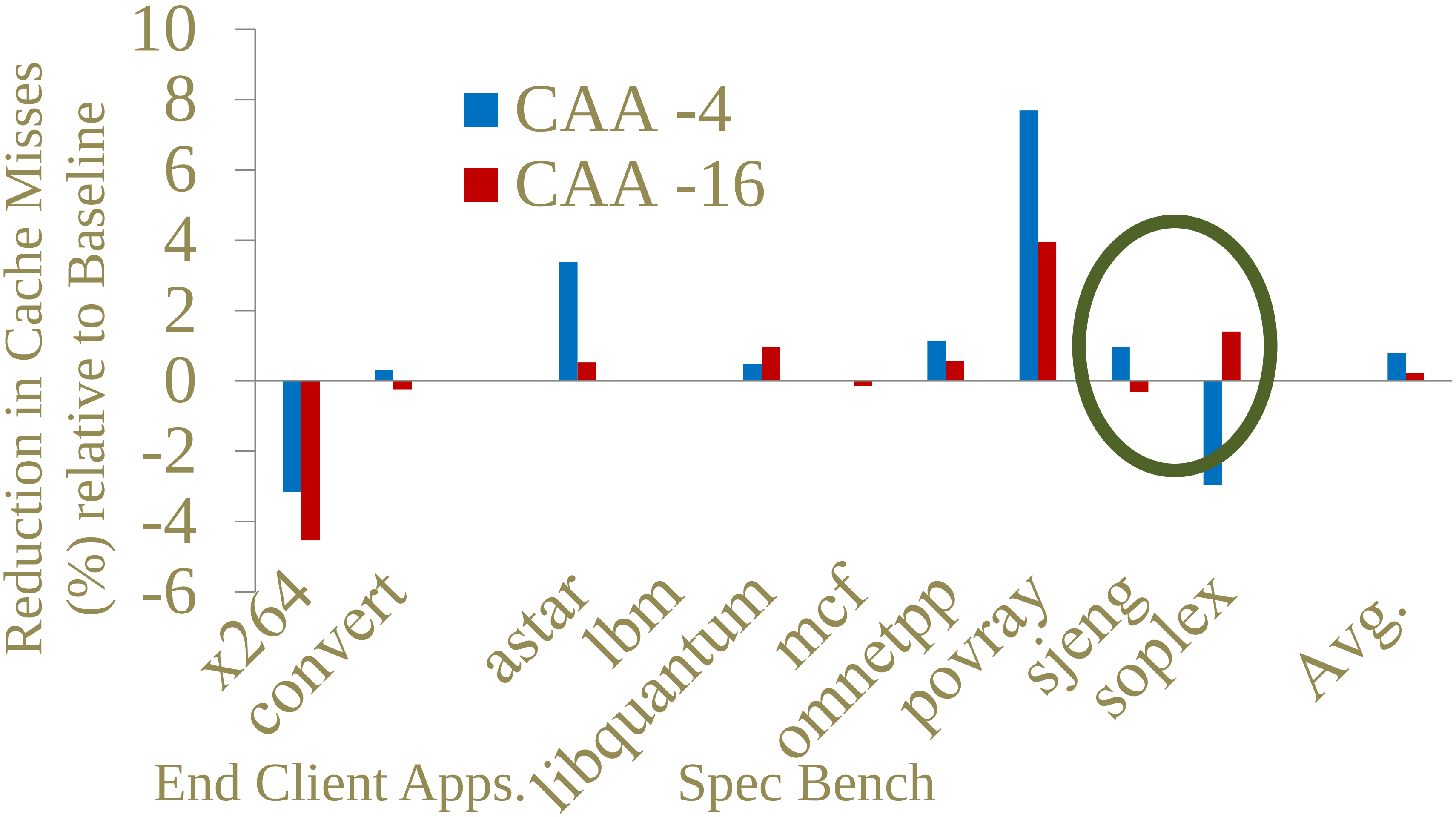
CAA - Reduction in Contiguity Misses



Contiguity-Bucket-based Page Allocator

- Reduces page contiguity misses by 89%

NVMCap Cache Miss Reduction



- ✦ Beneficial for apps with large memory footprints
- ✦ Adding more pages to bucket can increase cache misses due to linked list traversal

Agenda

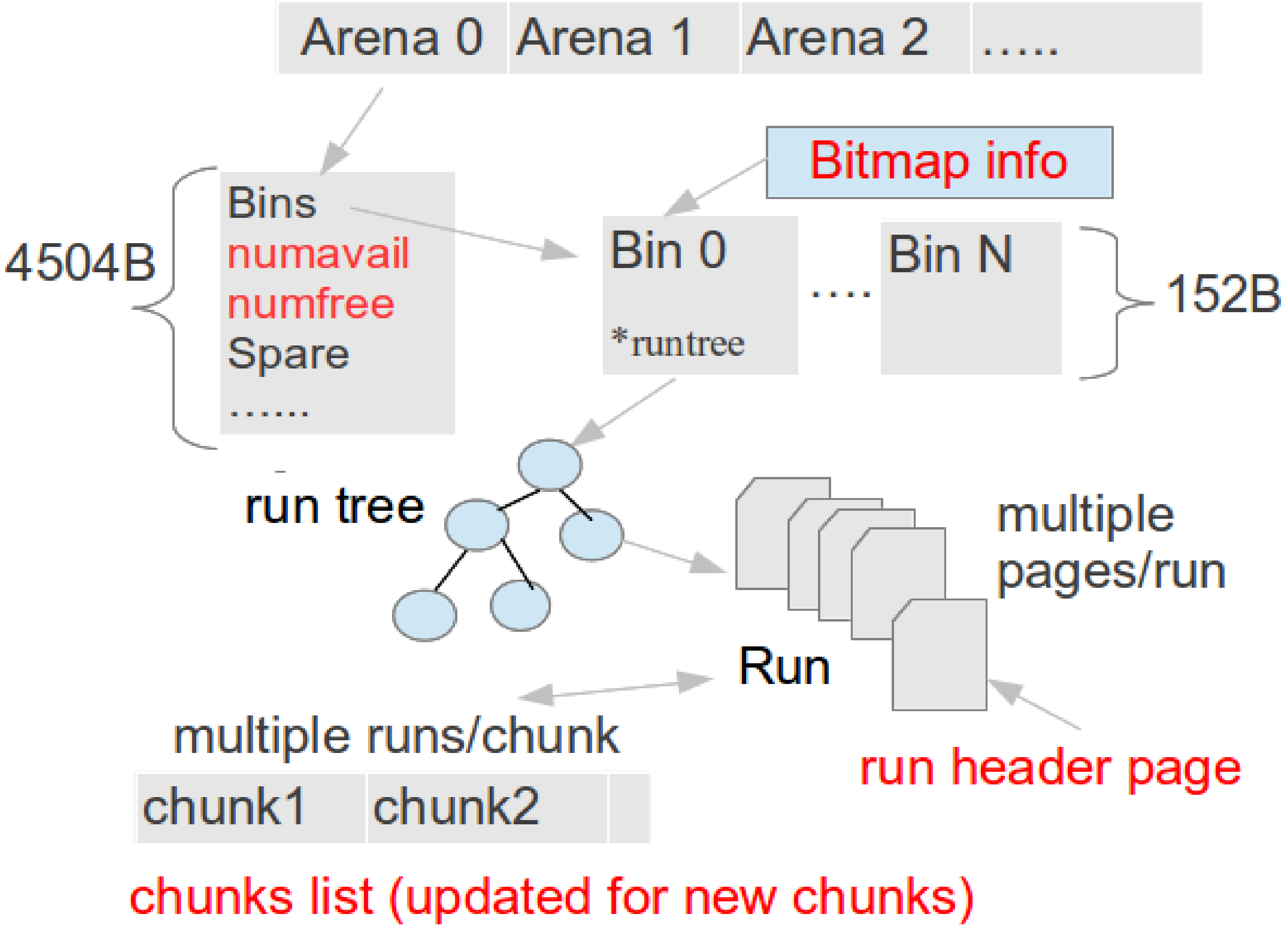
- Motivation
- Dual use of NVM Heap
  - High level Design
  - Programming Interface
- Sources of Persistence cost in dual use of NVM
- **Optimizations to Reduce Persistence Cost**
  - Cache conflict aware allocation
  - **Library allocator optimization**
  - **Hybrid Logging**
- Conclusions and Future Work

Library Allocator Overhead

- ✦ Nonvolatile heaps require user-level allocator
- ✦ Modern allocators use complex data structures
- ✦ Placing complex allocator structures in NVM requires multiple cache line flushes
- ✦ Increases cache misses and NVM writes for NVMPersist and NVMCap

Porting DRAM Allocators for NVM

Persistence support for JEMalloc ~4 CLFlush/ alloc.



NVM Write Aware Allocator (NVMA)

- Allocator complexity “independent of” NVM support

- Idea: place complex allocator structures into DRAM

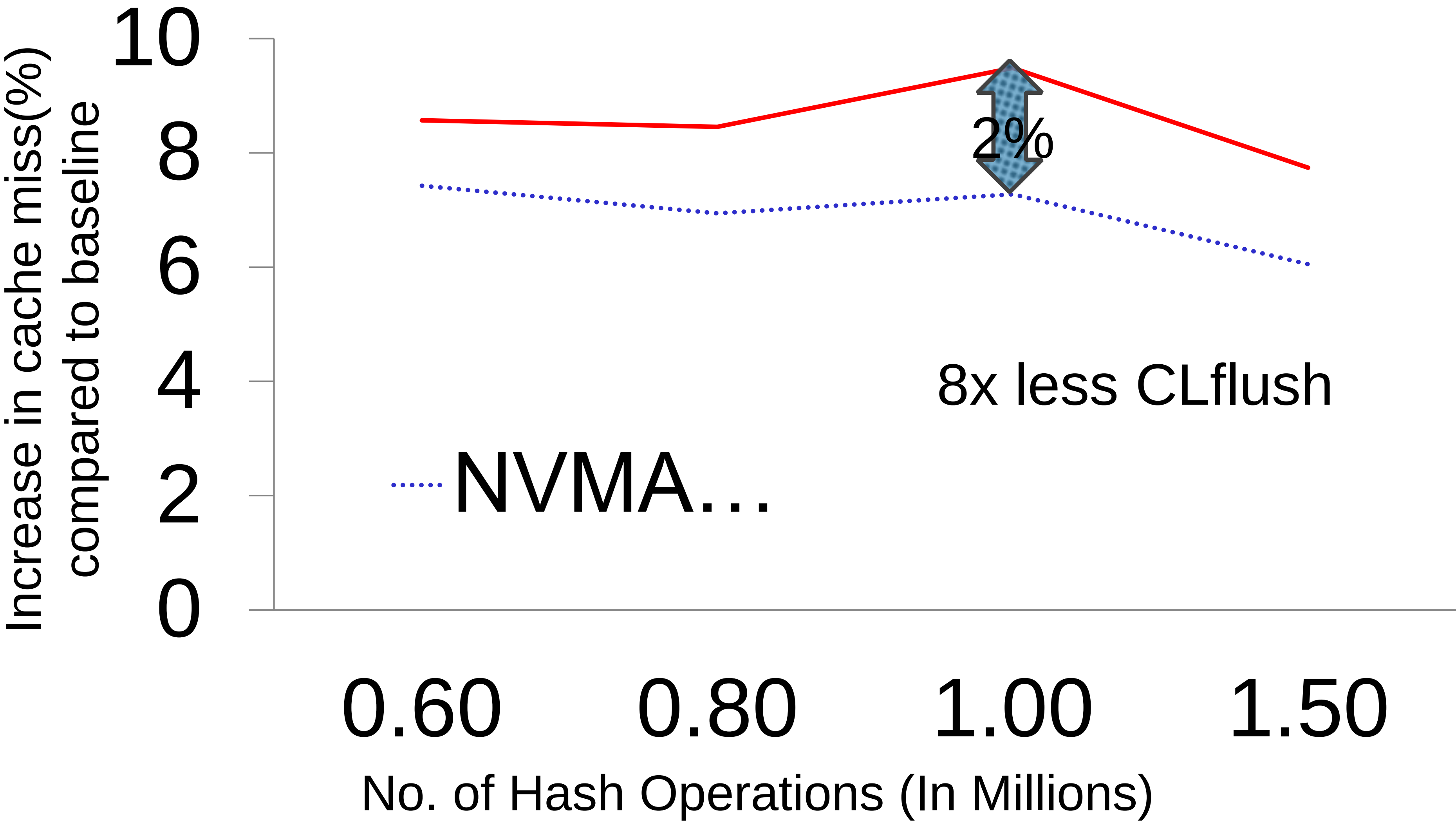
- NVM contains only log of allocations and deletions

C1	***C2***	***C3***	***.....***	***....***

C1,C2 indicates log of allocated chunks

- Flush only log information to NVM (~2 lines)

NVMA – Cache Flush Reduction

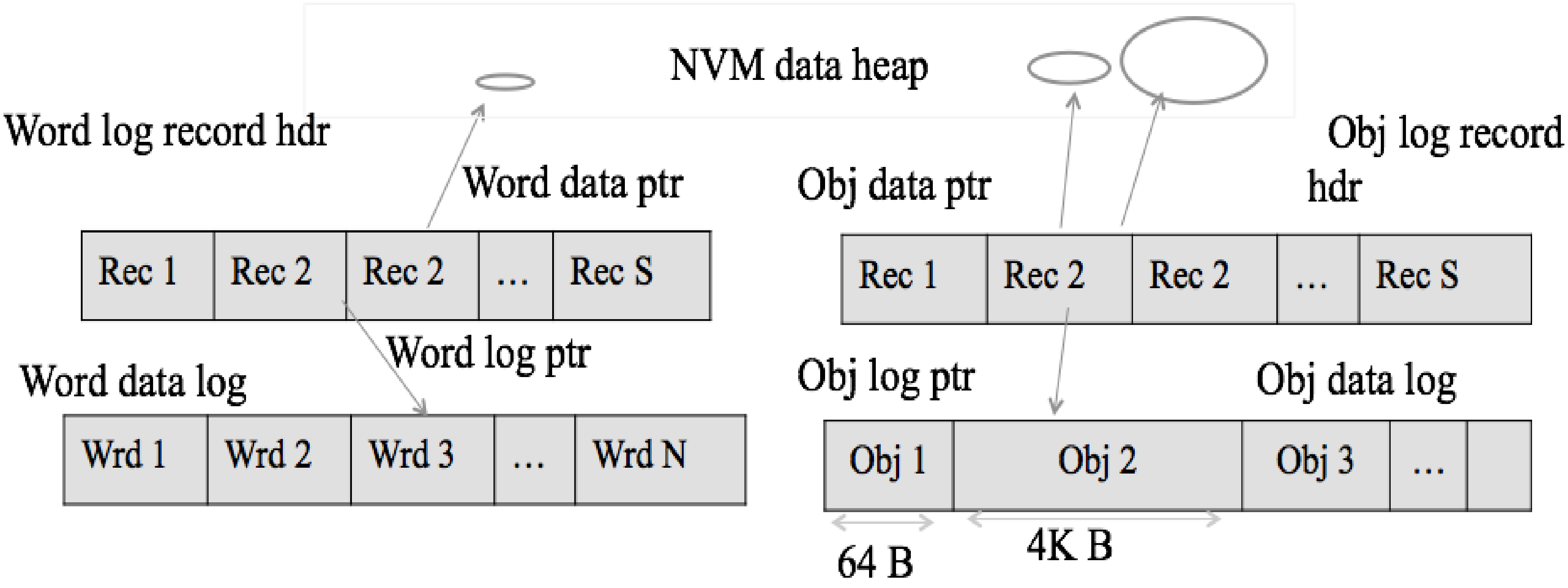


Logging Overheads

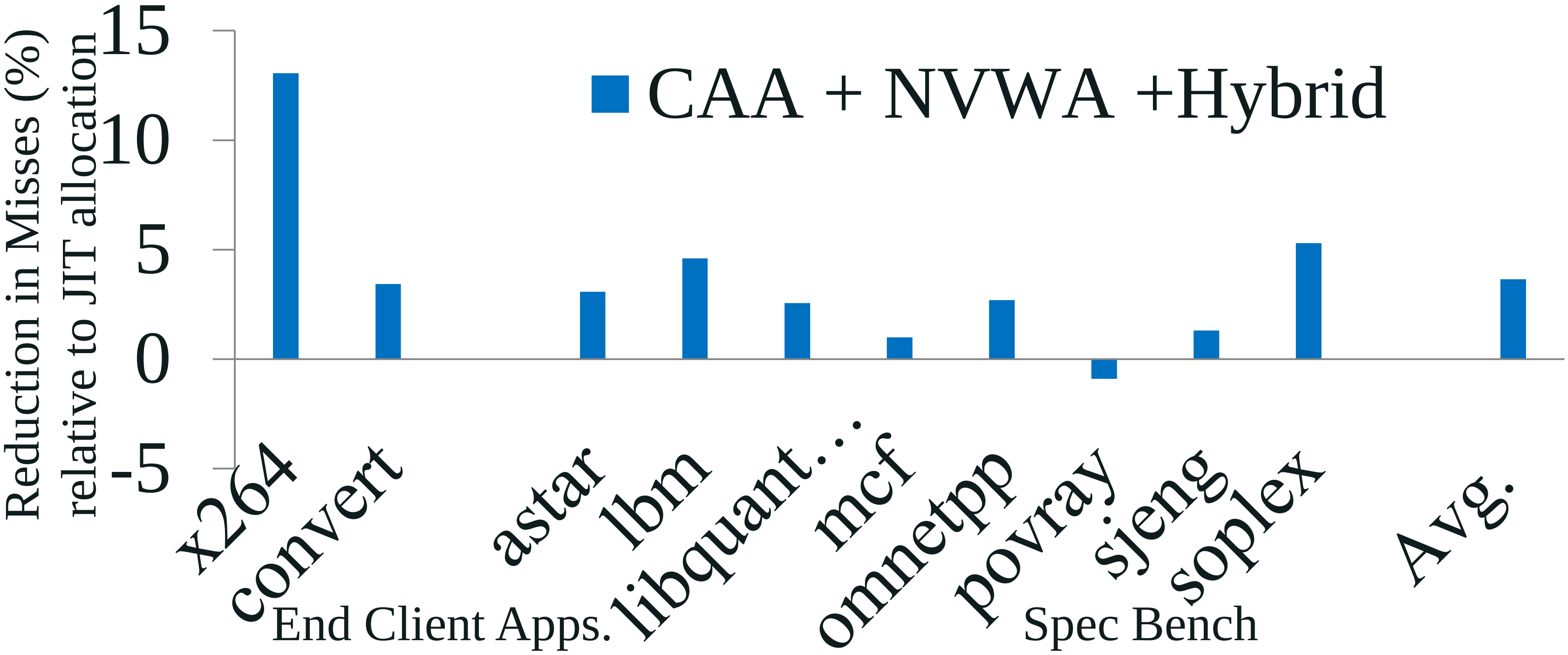
- Logging required for apps. with strong durability requirements
- Logs must be frequently flushed to NVM
- Current Word/Object logs increase NVM writes
- Word based logs: High log metadata/ log data ratio
- Object log: Logs entire object even for a word change

Hybrid Log Design

- ✦ Hybrid log to address word/object granularity tradeoffs
- ✦ Flexible use of word/object logs in same transaction
- ✦ Applications specify the transaction type
- ✦ Word- and object-based logs are maintained separately

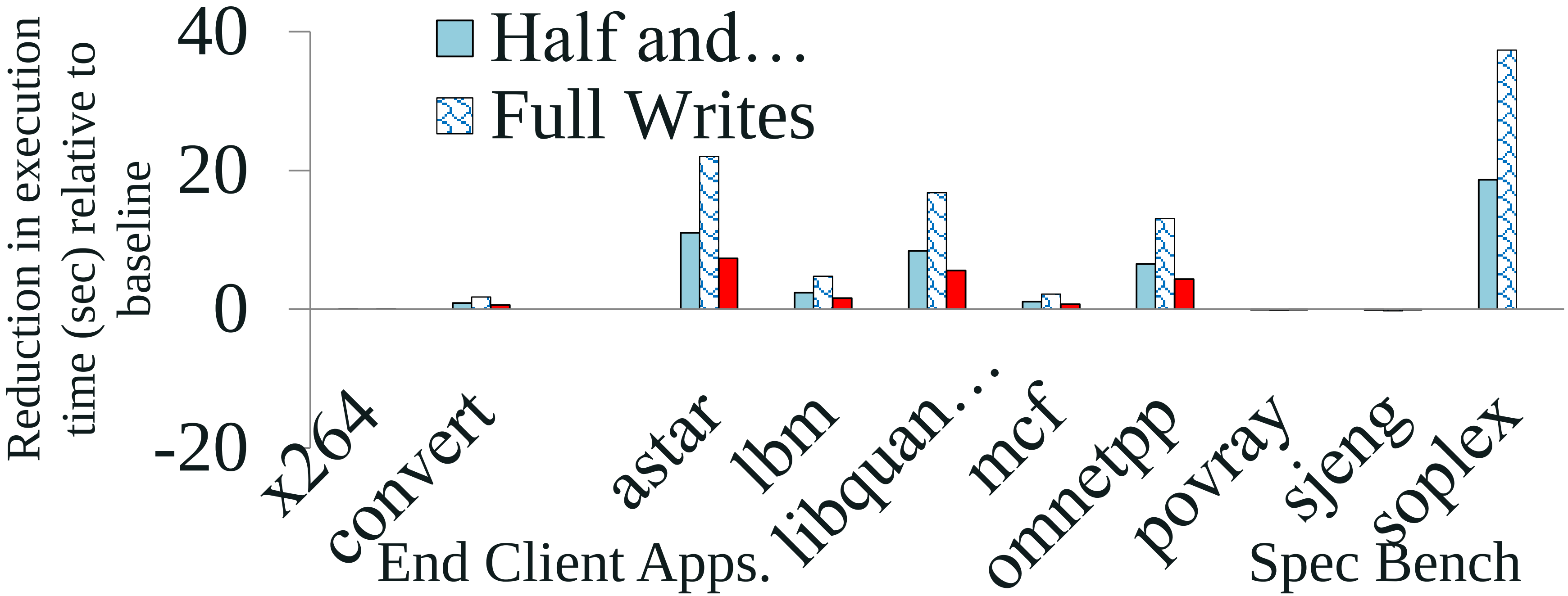


Optimization - Miss Reduction



- ✦Reduces misses by 1-2% compared to CAA+ NVWA
- ✦With increasing rate of hash operations, more gains

Estimated Impact on Runtime



Gains in runtime improvement can be substantial with optimizations that reduce misses by ~2%

- Half-Half : Half the misses reduced are NVM writes
- Full Writes: All misses reduced are NVM writes
- One-third: 1/3 of misses reduced are NVM writes

Summary & Future Work

✦ Efficient use of NVM requires cross-stack changes to applications and systems

✦ Analysis of dual use NVM shows potential high impact of NVMPersist on NVMCap

✦ Impact reduced via: page contiguity, NVM-aware user library allocator, and hybrid logging

✦ Improvements result in ~12%-13% reduced cache misses, with consequent substantial gains in applications performance

✦ Future work: DRAM vs. NVM data structures (e.g., OS allocator)

✦ Analysis of power implications

=> ‘Think Memory’, not cores!

Shared Platforms: Performance Effects

- Current hypervisors are limited in their ability to meet performance needs and isolation for ***multiple*** hosted Applications

- Application performance depends on resources beyond CPU + Memory: shared resources: *Memory Bandwidth*, I/O*

- Shared resources *not as easily partitioned* as CPU, Memory in hardware (limited support e.g., NUMA)
- Application resource requirements are *elastic along multiple resource dimensions*

- *State of art hypervisor resource managers and arbitration offer inadequate solutions to manage elasticity with isolation*

Shared Platforms: Performance Effects

- Arbitrary *interference* for resource shares may have detrimental performance implications

- Some applications more *sensitive* to interference than others

- Interference: an application’s resource shares imposing on another’s sensitive resource shares

Challenges for Resource Managers of Shared Platforms

How to improve ***isolation*** of multiple performance properties, leveraging limited support in hardware?

How to further efficiently arbitrate and manage ***elastic*** application resource demands when multiple varying performance requirements need to be met?

Costs of resource-reallocation to maintain elasticity may be non-trivial.

Thanks!

‘’Think Memory’: encouraging you to rethink ‘storage, I/O, and ‘memory’ (usage and management) for future multicore platforms

Applications Analyzed…

OpenCV based FaceRecognition *

- Generates training database from images
- Database is in a XML format
- For recognition, Eigen vector analysis of source with database (150 image database with train/recognize)

Snappy Compression *

- Fast compression library from google.
- Importance on speed rather than compression ratio (2GB)

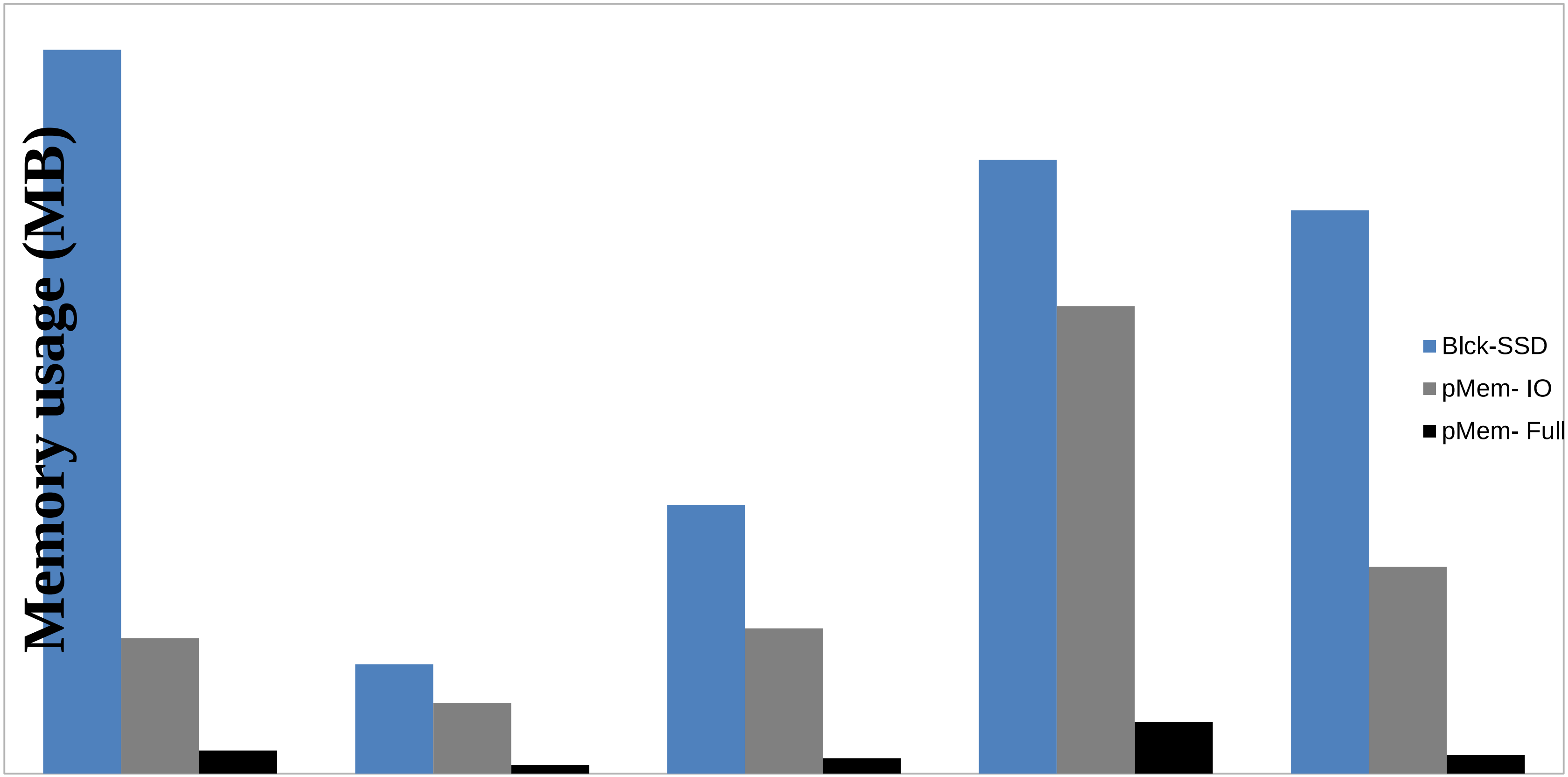
JPEG Conversion Library

- Standard Linux/Windows JPEG library
- We use the JPEG to BMP conversion utility
- Most time spent on image decoding (similar to X264)
- Read and Write Intensive (5000 images)

Crime/DBACL – Diagramic Bayesian Classification (Machine Learning)

- Classified user emails/documents (500 MB of documents)

pMem - Memory (DRAM) Usage

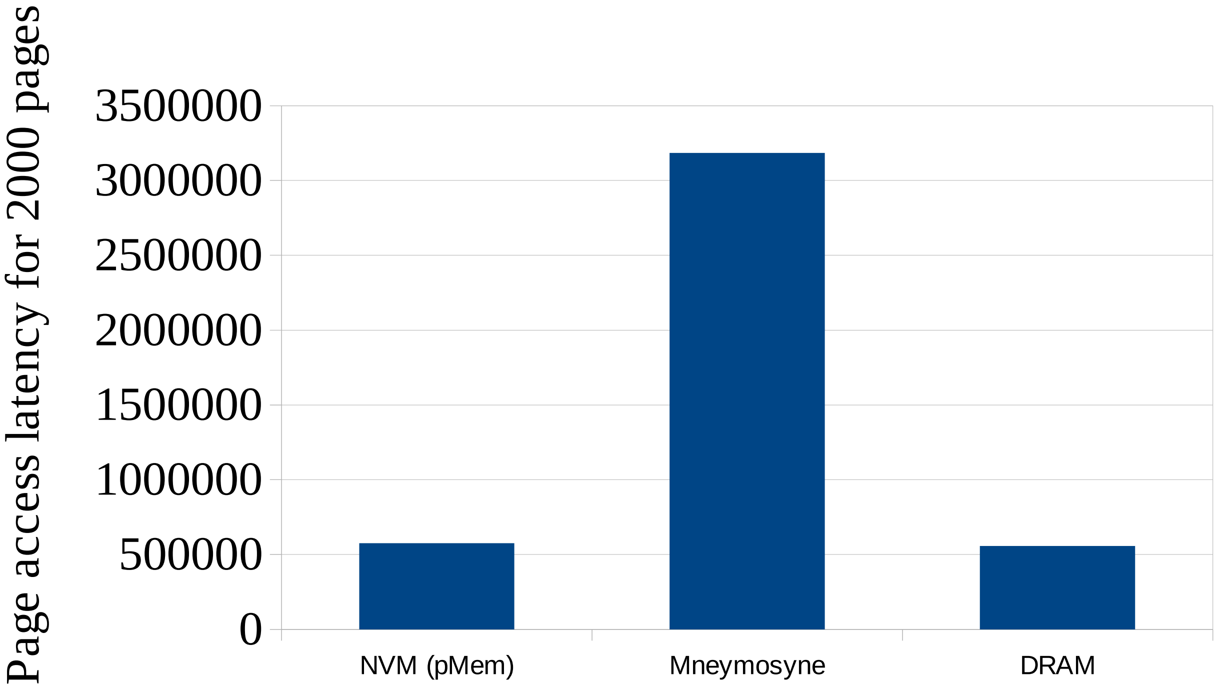


pMem I/O – NVM only for persistence

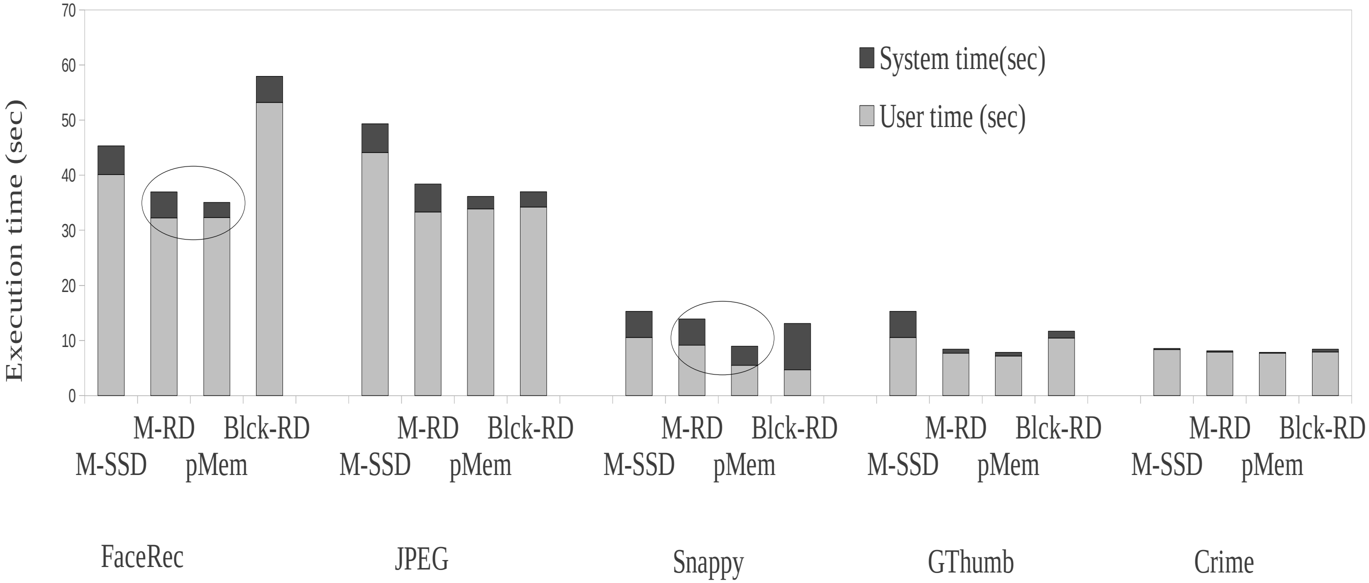
pMem- Full – NVM for persistence and as additional memory

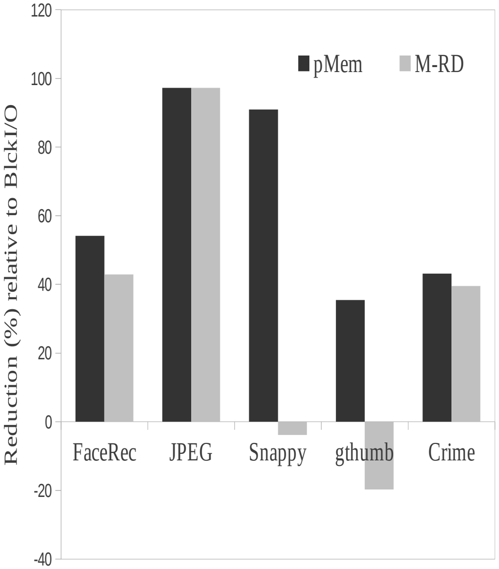
Blck-SSD – Block-based SSD usage

pMem – Page Access Latencies



pMem for Persistence - Performance Gains





User-kernel switch reduction
relative to Blck-IO

- RD – RamDisk, M-mmap, Blck- Block based access
- Worst case: 4%-6% overhead compared to DRAM when using NVM for execution and storage
- Avoids high context switch costs compared to 'mmap'

Optimizing Checkpoints Using NVM as Virtual Memory

Sudarsun Kannan,
Ada Gavrilovska,
Karsten Schwan
CERCS - *Georgia Tech*

Dejan Milojicic
HP Labs (Palo Alto)