pMem- Achieving Dual Benefits of PCM/NVM by Reducing Persistence Overheads

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan

Motivation

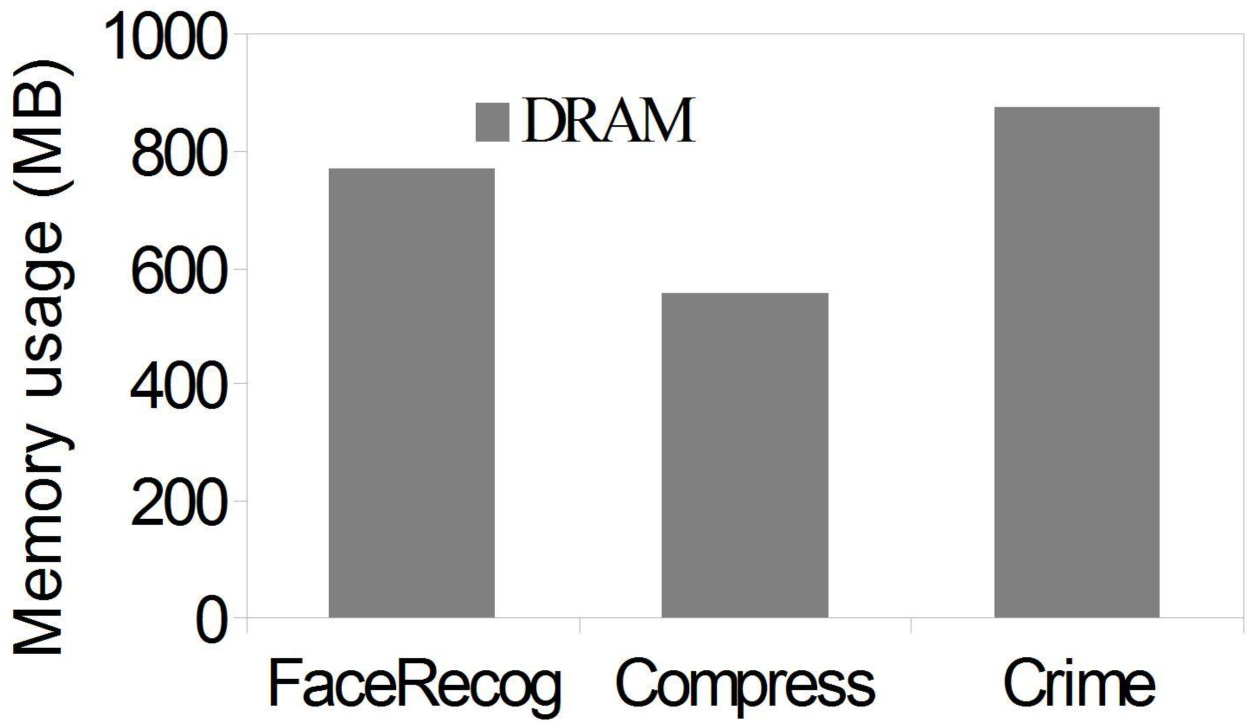
Growing number of end client apps
    E.g., Webstore -33 million users. ~1 Million apps

Lots of Data-intensive applications
    Picasa, Digikam, Facebook, Face/Voice recognition etc.

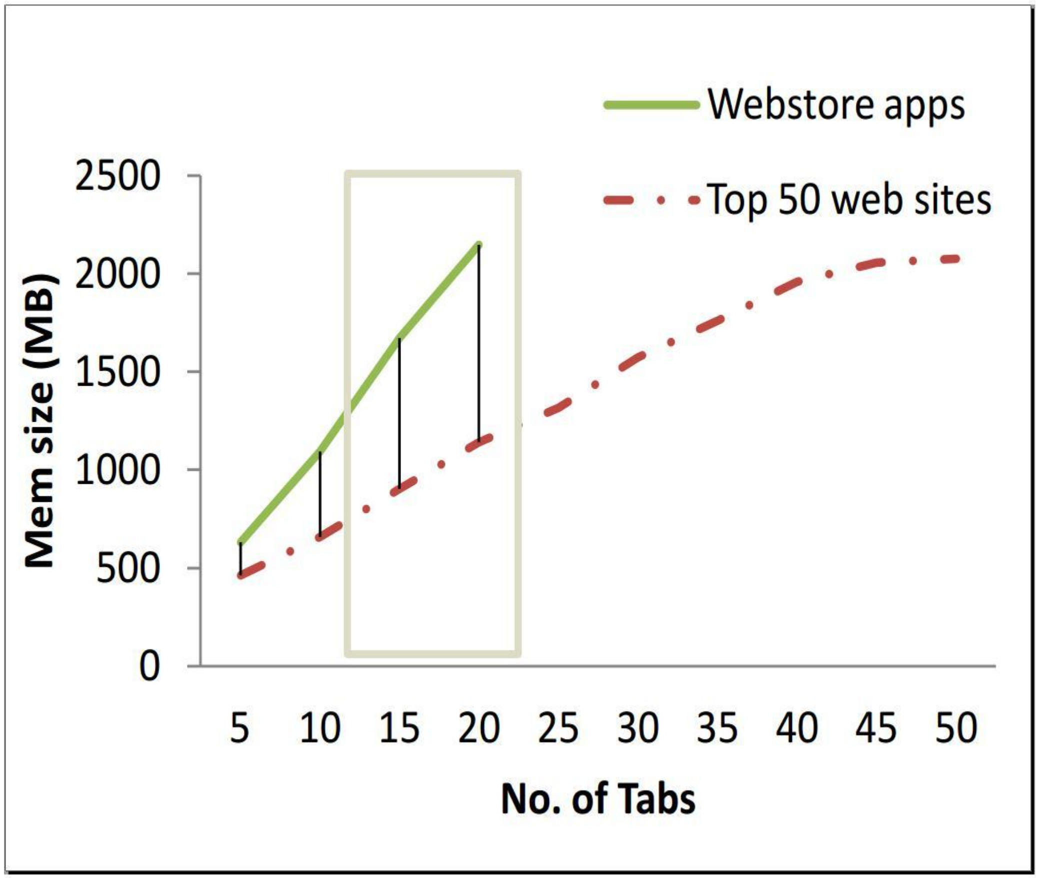
Increasing number of cores and multi-threaded applications

Effective memory capacity + persistent storage bottlenecks
    - MDRAM has limited scalability
    - External flash ~4- 16 MB/Sec (FAST' 11, Kim et al.)

Motivation - Memory Capacity



Motivation - Memory Capacity

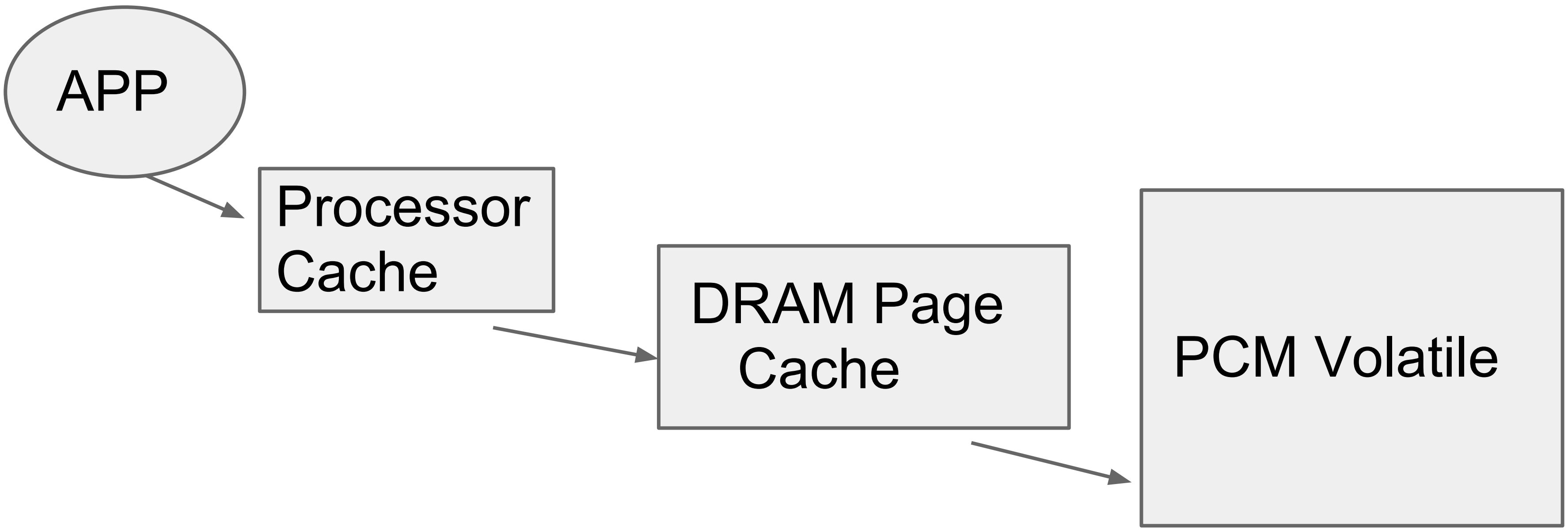


Byte addressable storage?

NVM technologies like PCM
- Byte addressable and persistent
- 2X-4X higher density compared to DRAMs
- 100X faster compared to SSDs
- Less power due to absence of refresh
- Byte addressability - (Can be connected across memory bus and accessed with load/stores)

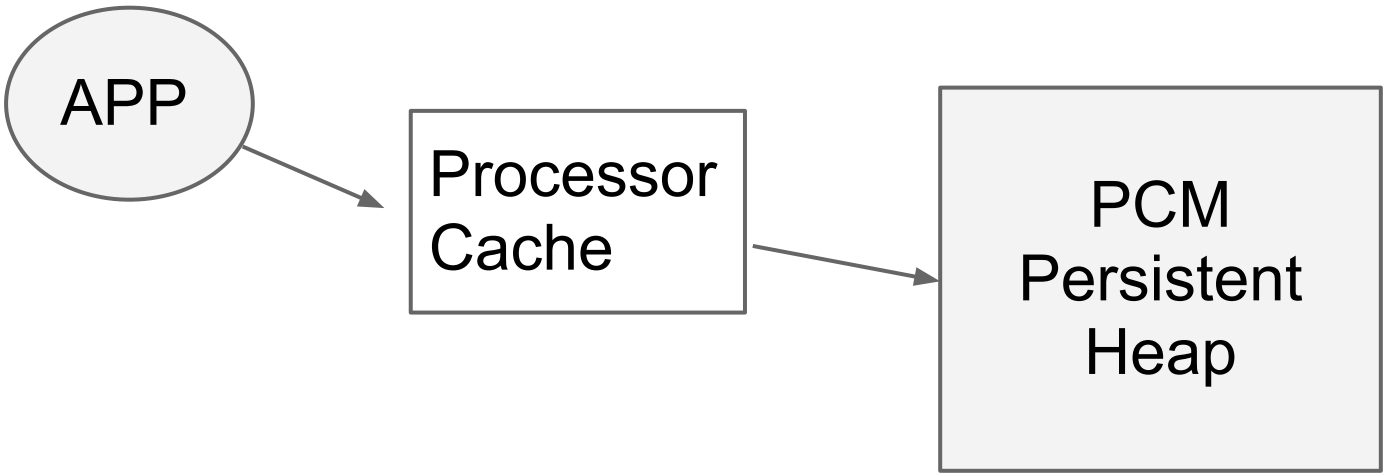
Limitations:
- Hight write latencies compared to DRAMS
  - (4X - 10X slower around a microsec)
- Limited endurance (approx. 10^8 writes/cell)
- Limited bandwidth: interface and device bottlenecks

Prior Work: DRAM as Cache



Good for high end server

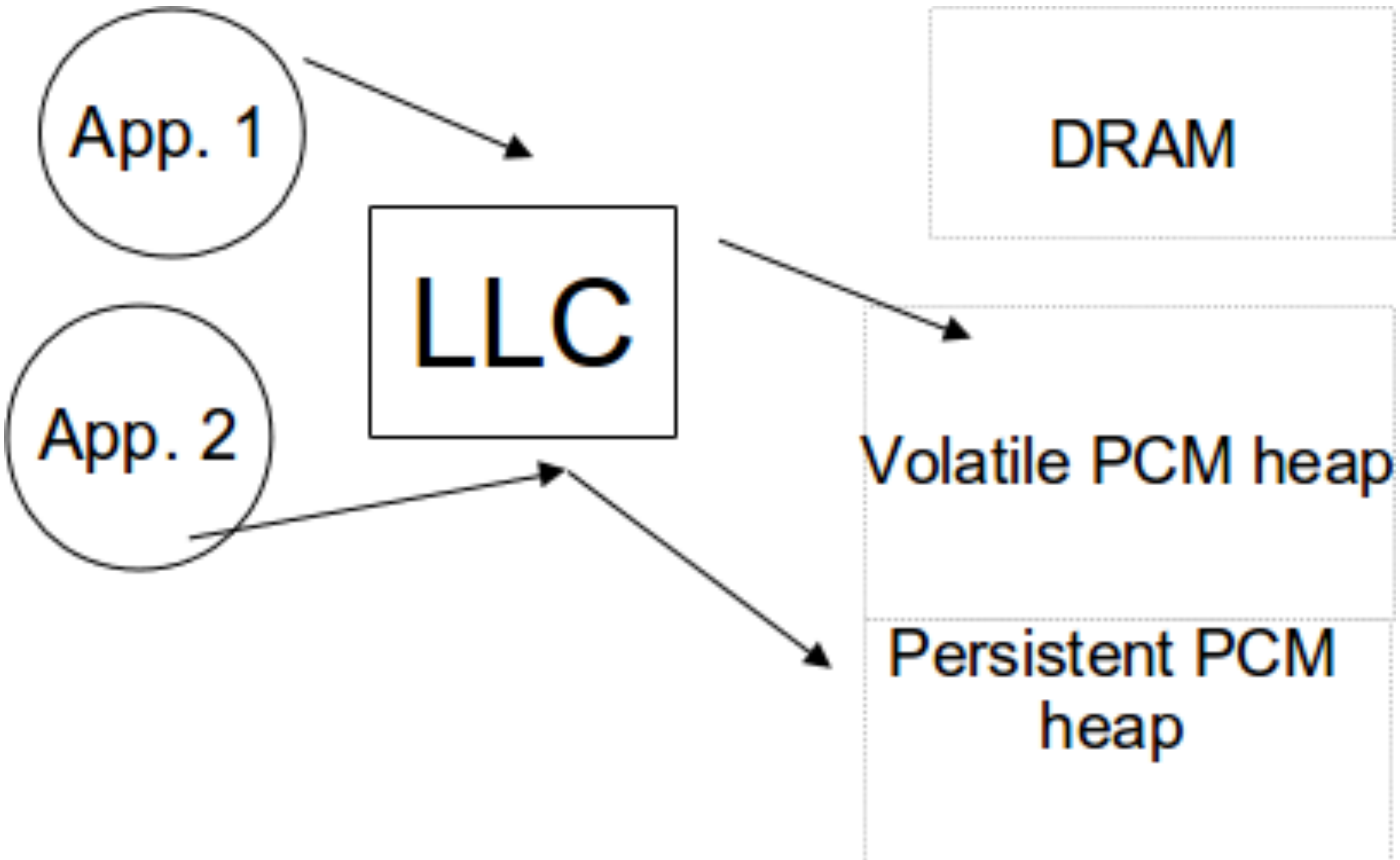
Prior Work: Fast Non Volatile Heap



High Persistence Guarantees:

- Frequent cache flushing, memory fencing, writes to PCM
- High persistent management overhead
  - (user + kernel layers)

Proposed: Capacity + Persistence



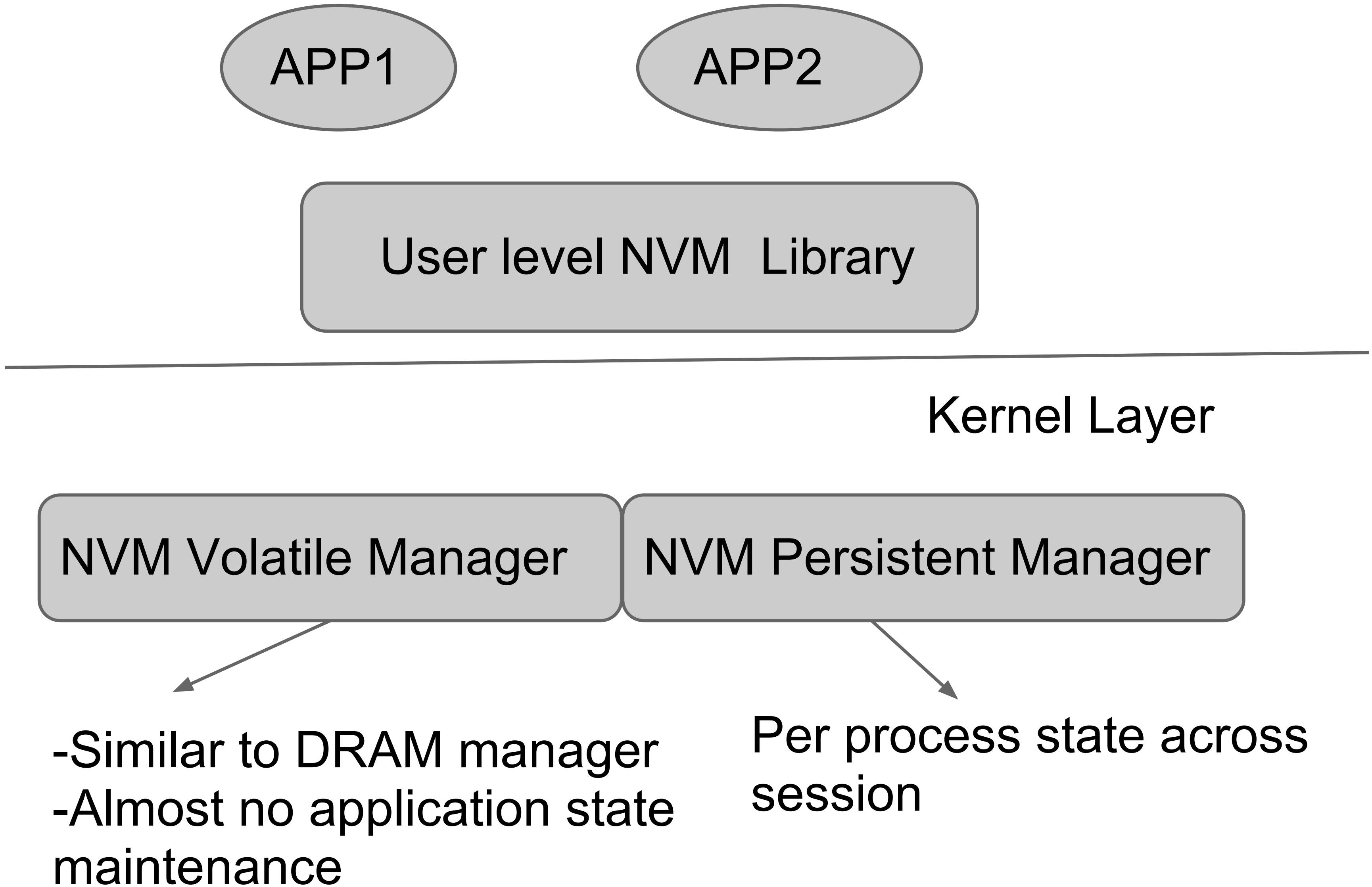
Processor cache plays crucial role in reducing write latency

Proposed: Dual Use using pMem

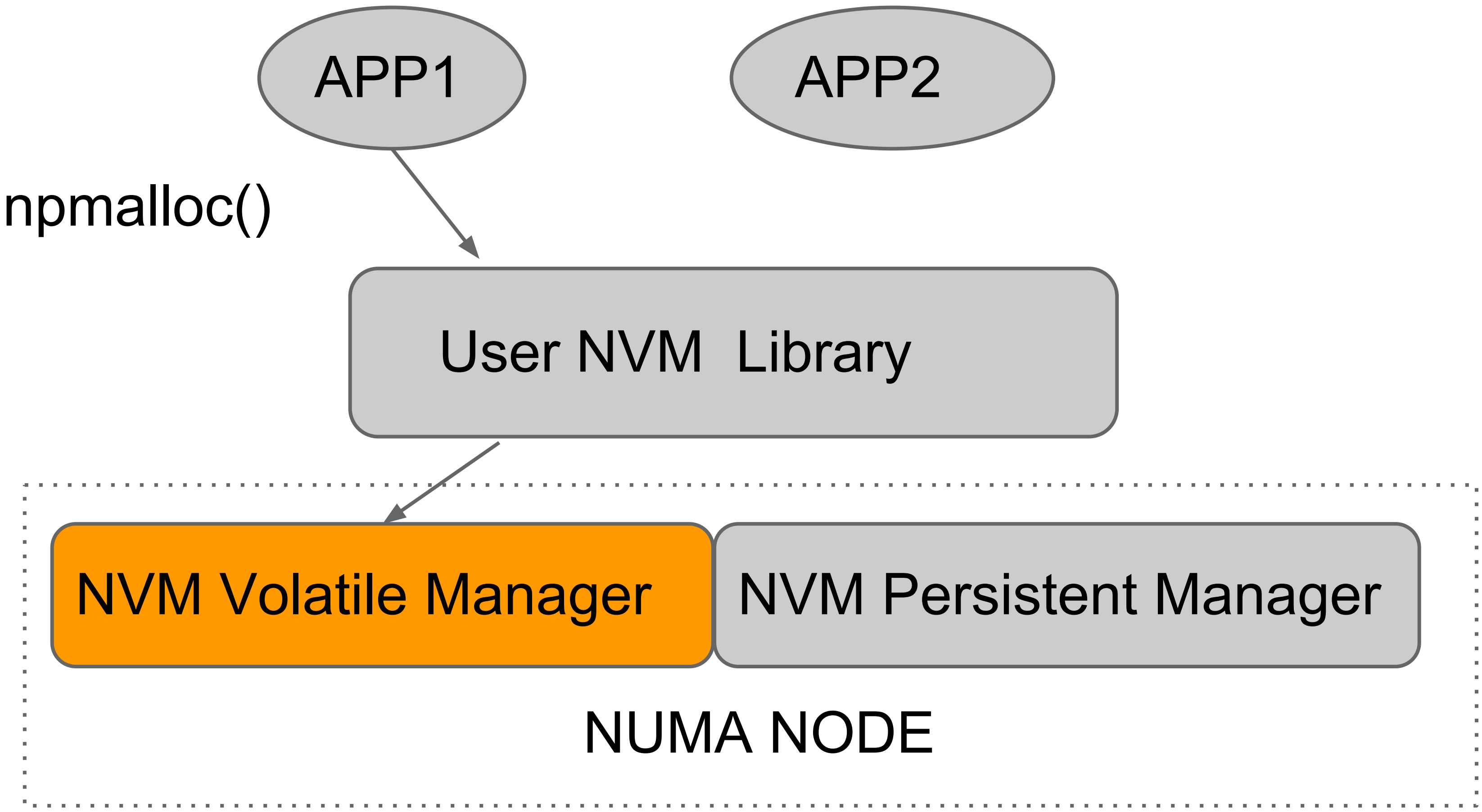
- Advantages
  - Dual benefits: Capacity + fast persistence

- Key Idea
  - Use PCM as NUMA node
  - PCM 'Node' partitioned to volatile + persistent heap
  - Applications are provided with suitable interfaces
    - Application control persistent/non persistent data
  - Throw/ stay way from traditional I/O calls
    - Goal: Reduce software interaction (includes OS)

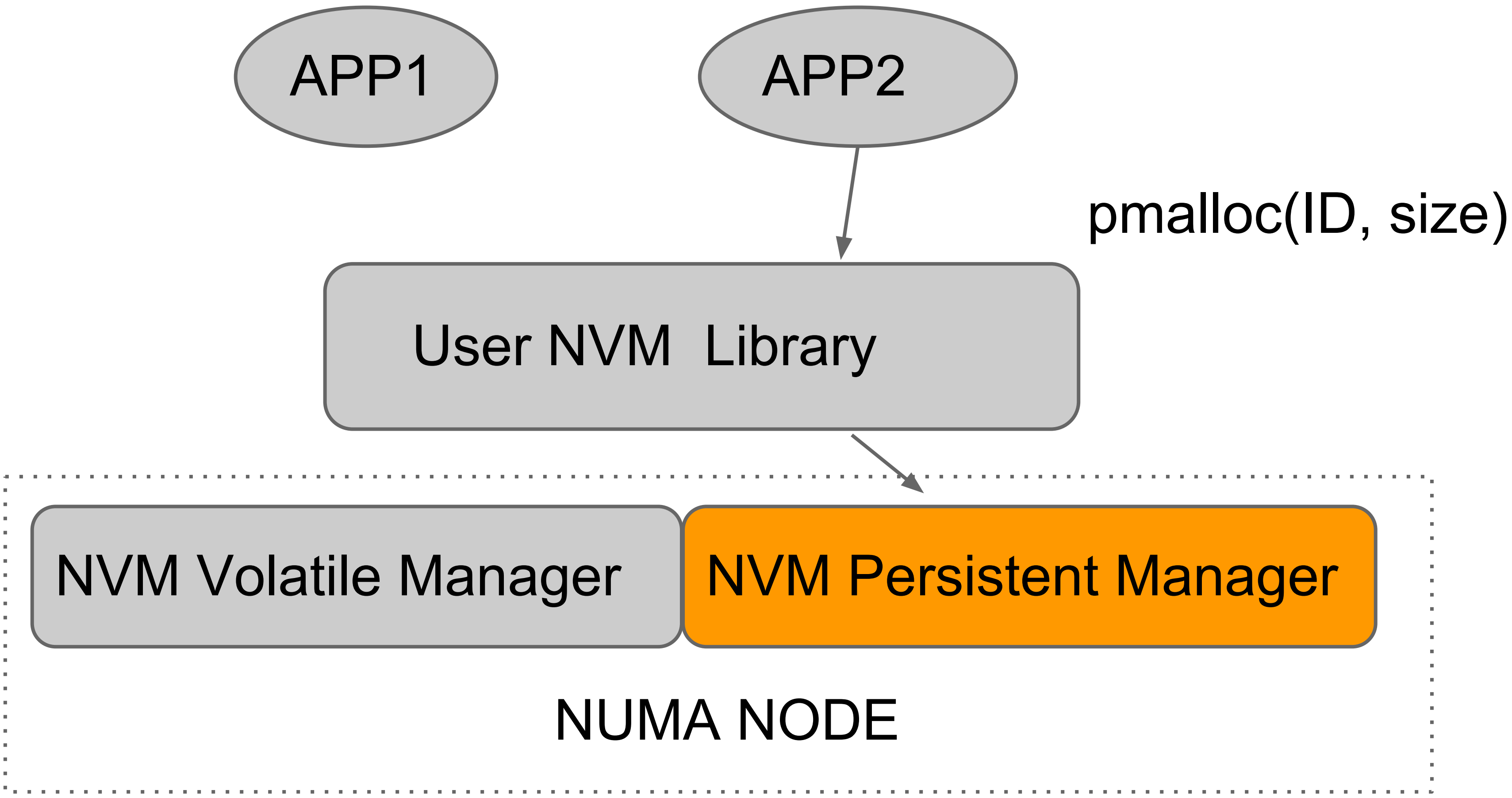
Proposed: Dual Use using pMem



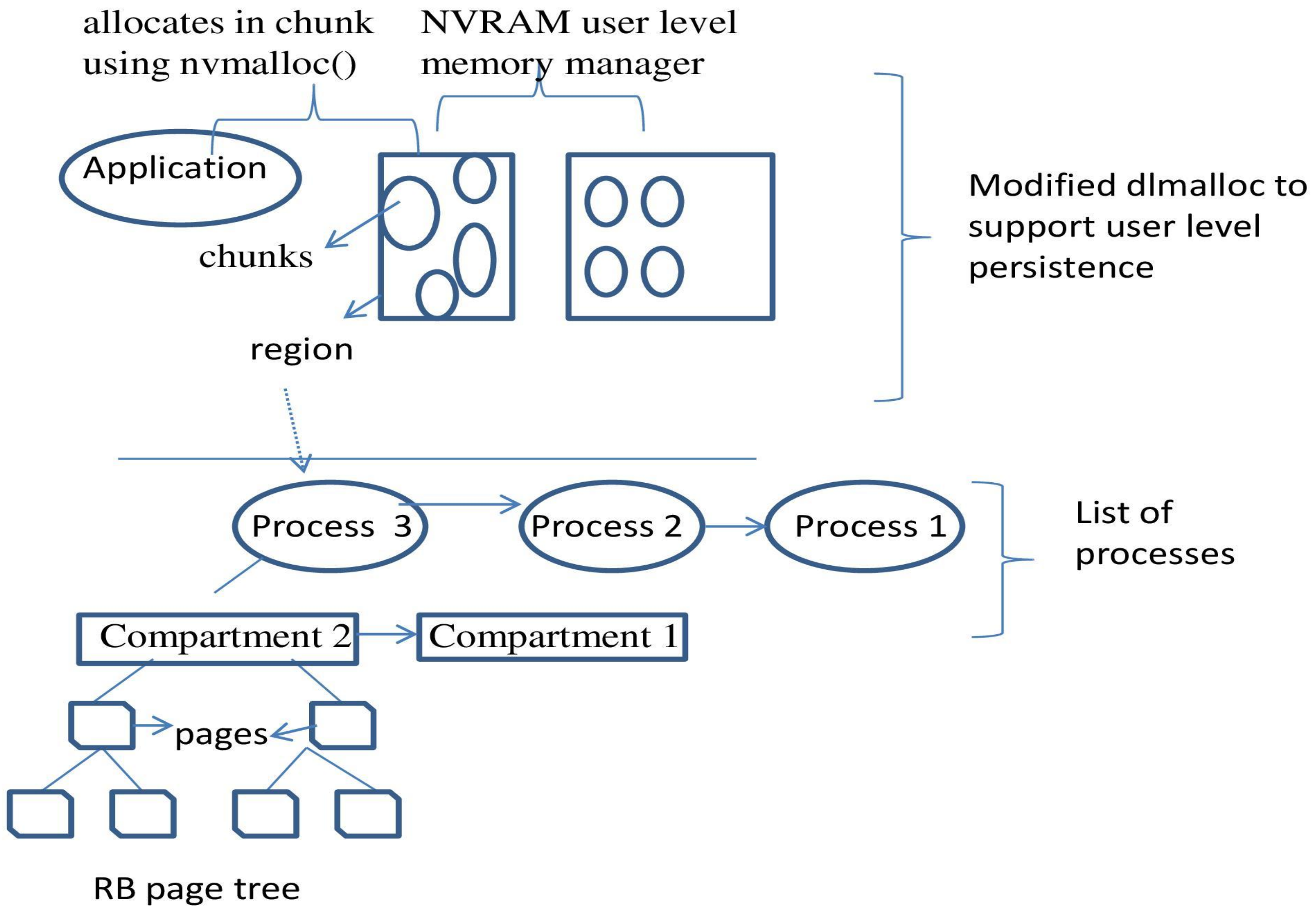
Proposed: Dual Use using pMem



Proposed: Dual Use using pMem



pMem System Structure



pMem Experimental Results

Experimental Method:

- DRAM as NVM with a NUMA node as PCM
- Persistence across sessions avoiding OS to reclaim pages
- Accounting for NVM read/writes using PIN based instrumentation
- Hardware counters to understand cache misses
- Also architectural simulations (MACSim)

Experimental Results

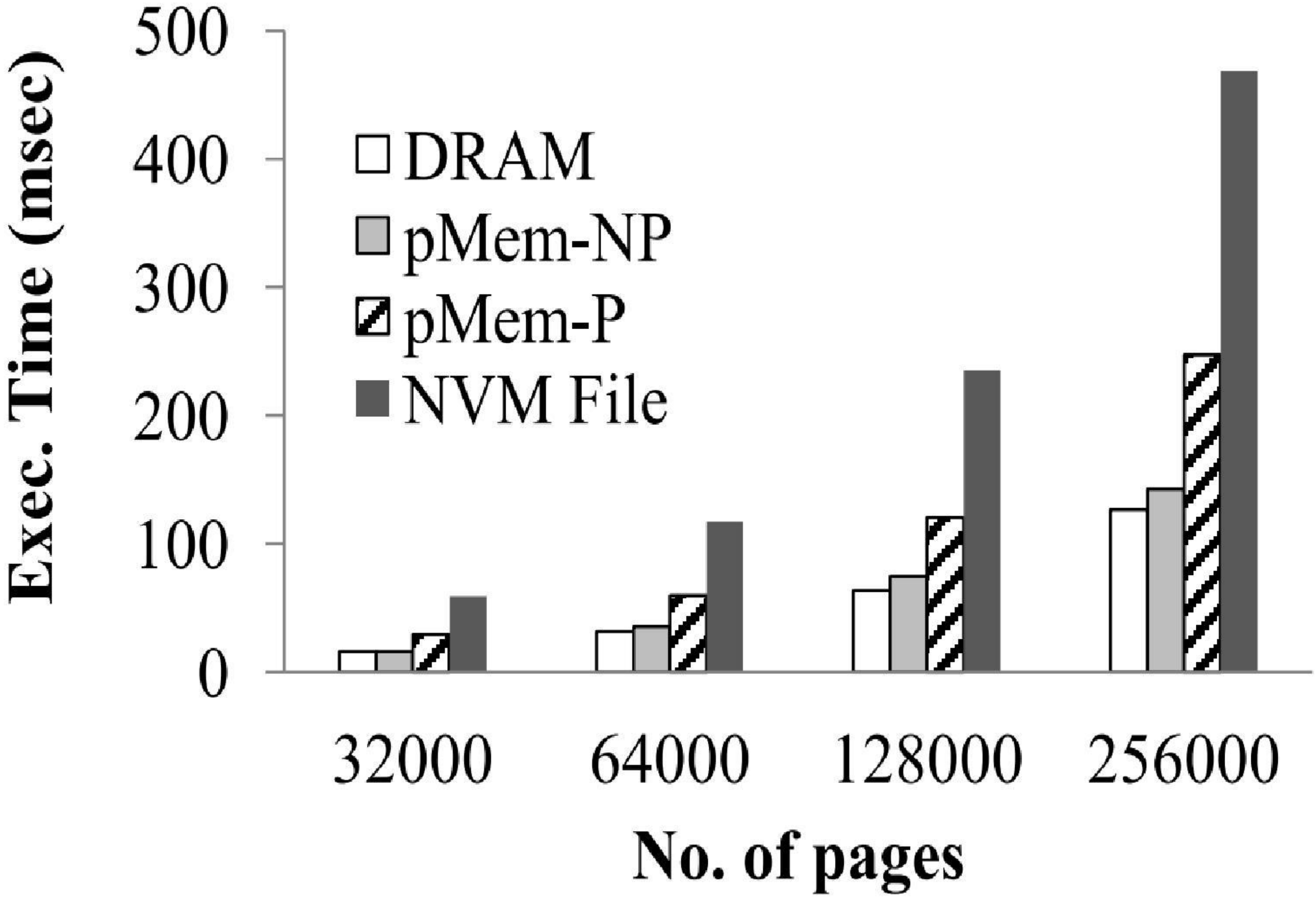
Experimental Use cases

Scalability:
    Linux Scalability benchmark for paging/allocation

Memory Capacity:
    Face recognition, Compression, Crime

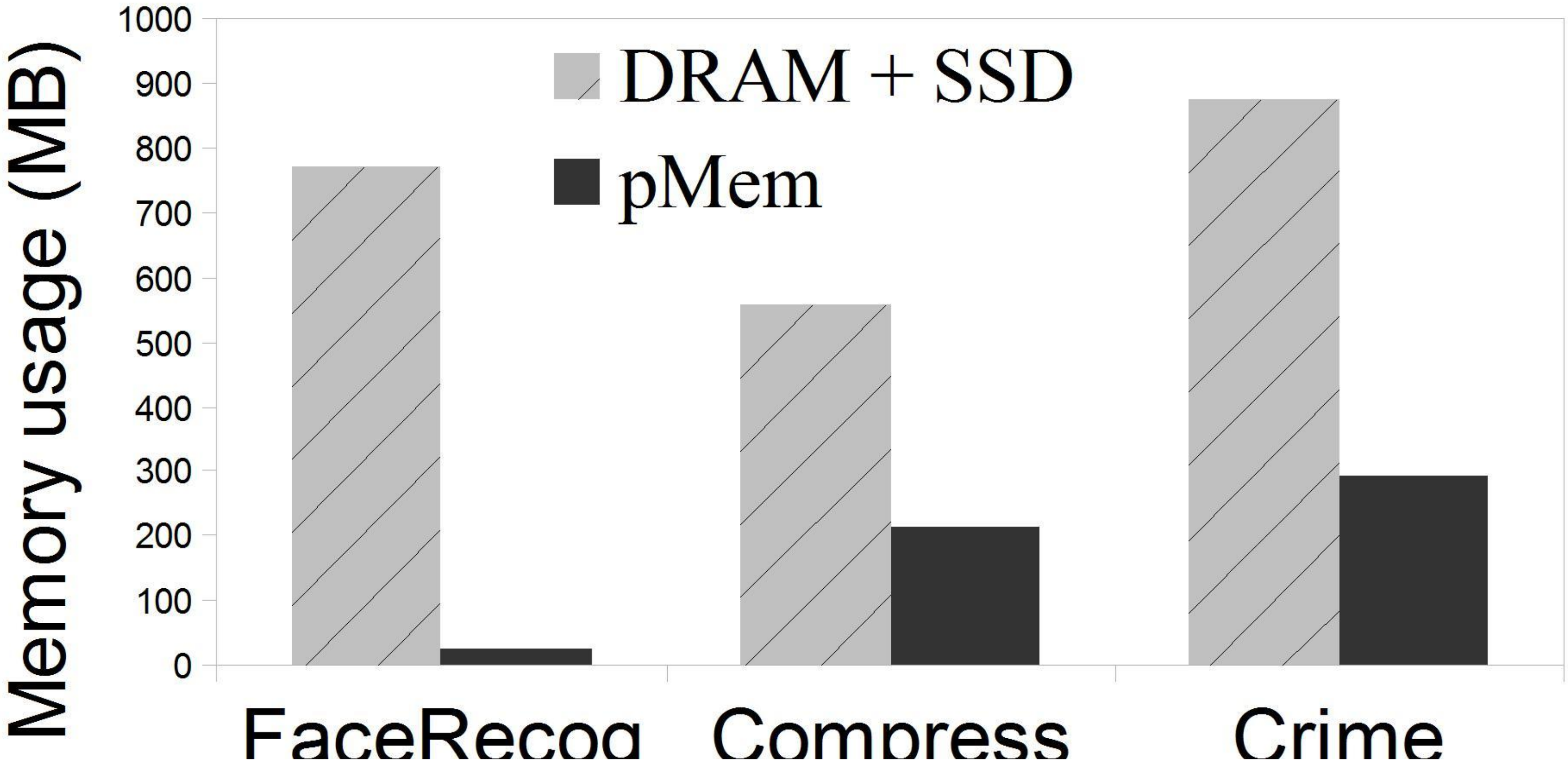
Persistence:
    Machine learning application to load user preferences during browser page time

pMem Paging Performance

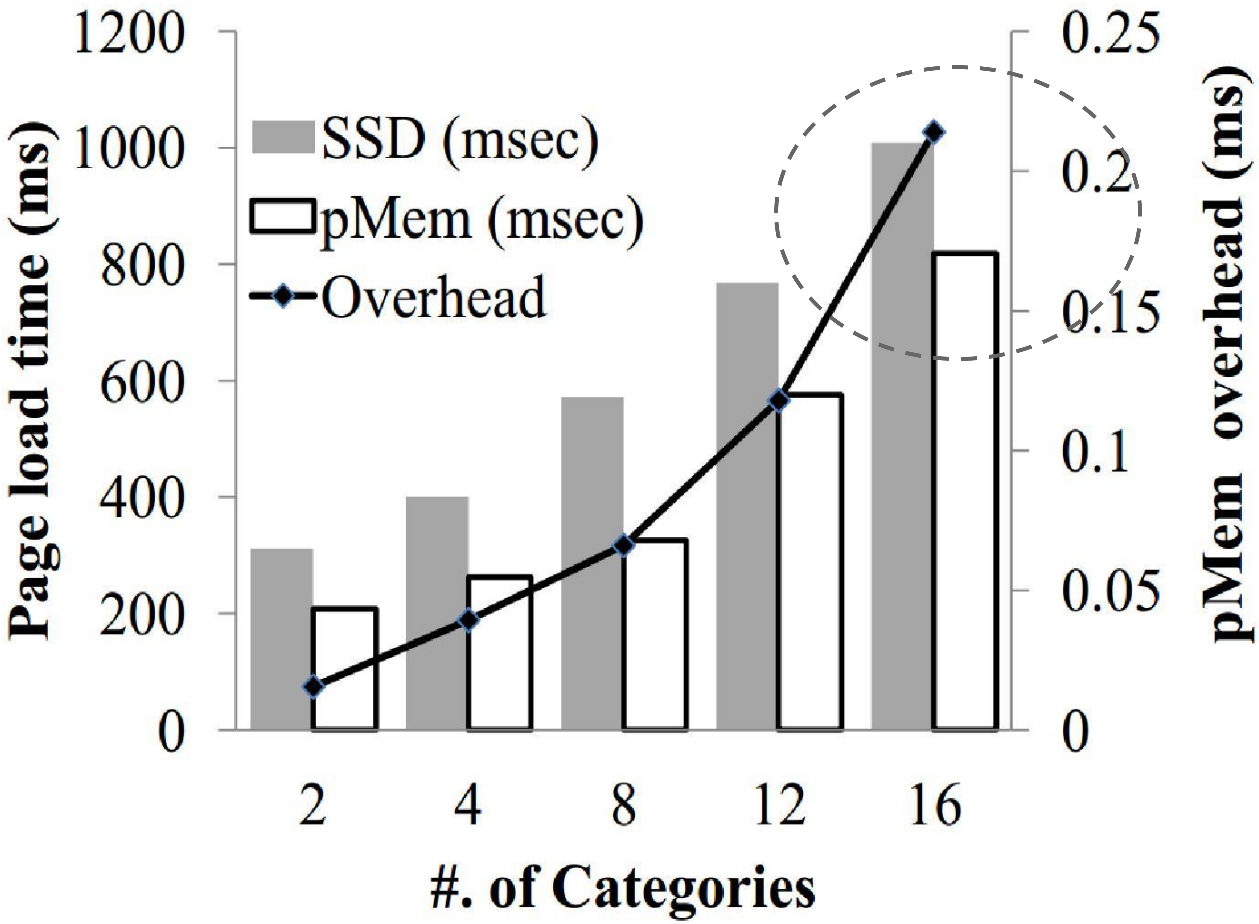


pMem Memory Usage

Performance 4%-6% overhead



pMem Persistent Storage



45% Improved I/O compared to SSD

With increasing data, cost of persistence increases

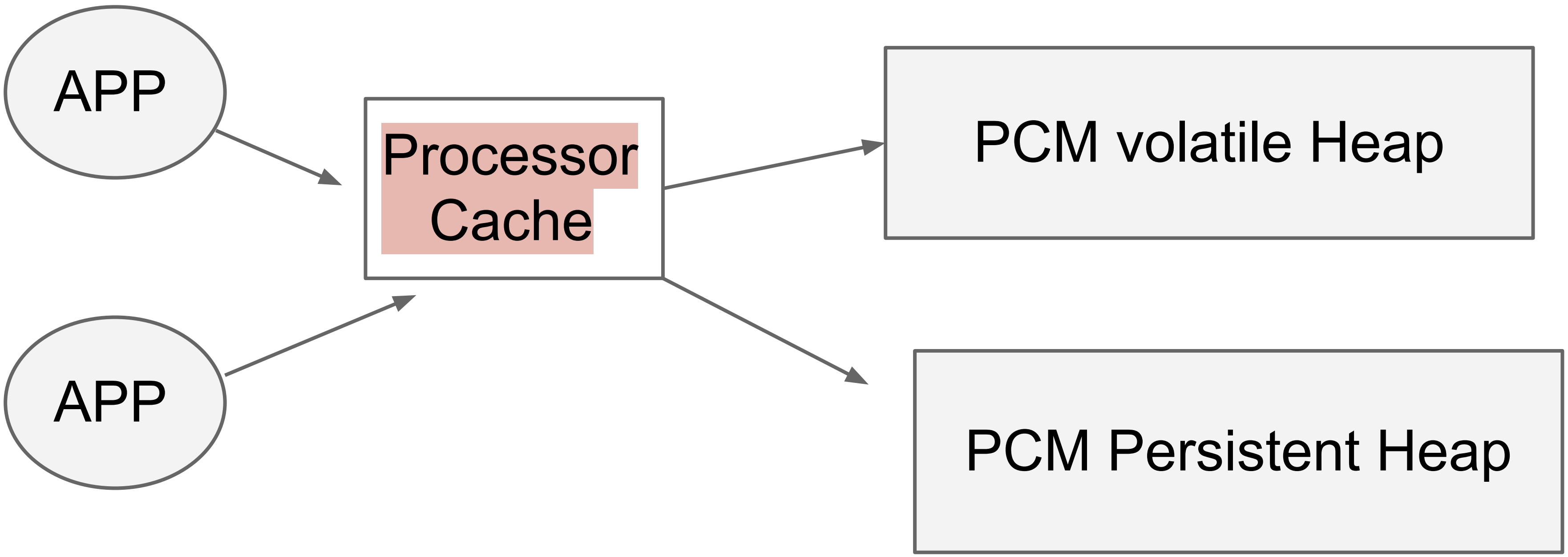
~62% improvement in persistent hashtables

Summary

- Volatile-Persistent heap partitioning
- Idea: Use PCM as persistent NUMA node
- Upto 91% memory capacity benefits
- ~45% faster I/O for end client apps.
- Less that 6%-7% runtime overhead on some apps

But PCM/NVMs are theoretically 100x faster :-)

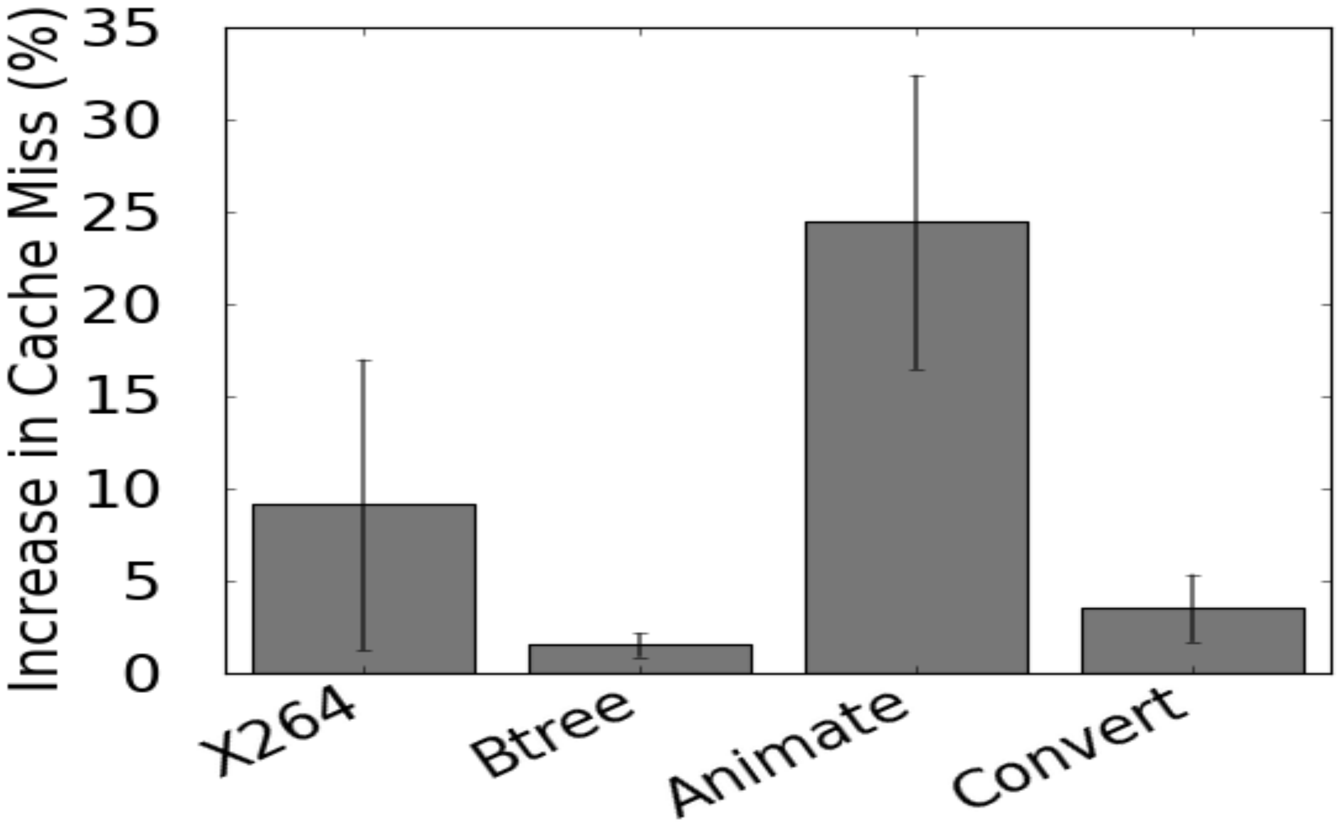
Persistence Overheads



Persistence requires constant barrier, cache line flushing

Is sharing cache a problem?

Effects of Persistence



Persistent Application: Hashtable with 1M Operations (puts and gets)
Intel Atom : Dual core, 1MB LLC, (8 way, Write Back, Shared LLC)
Persistent and volatile applications pinned to their cores

Effects of Persistence

```
AddHash_Entry() {
//Fence and Flush log (in PCM).
BEGINTRANS((void *)table,0);
 ++(table->entrycount);

//Fence and flush
e = nvalloc(sizeof(struct entry));

//Fence and flush
BEGINTRANS((void *)e,0);
     e->h = hash(h,k);
    e->k = k;
    e->v = v;
    table->table[index] = e;
//Fence and flush
 COMMIT((void *)e, (void *)table, 0);
```

Effects of Persistence

```
AddHash_Entry() {
//Fence and Flush log (in PCM).
BEGINTRANS((void *)table,0);
 ++(table->entrycount);

//Fence and flush
e = nvalloc(sizeof(struct entry));

//Fence and flush
BEGINTRANS((void *)e,0);
     e->h = hash(h,k);
    e->k = k;
    e->v = v;
    table->table[index] = e;
//Fence and flush
 COMMIT((void *)e, (void *)table, 0);
```

BEGINTRANS((void *)table,0); → Transactional overhead

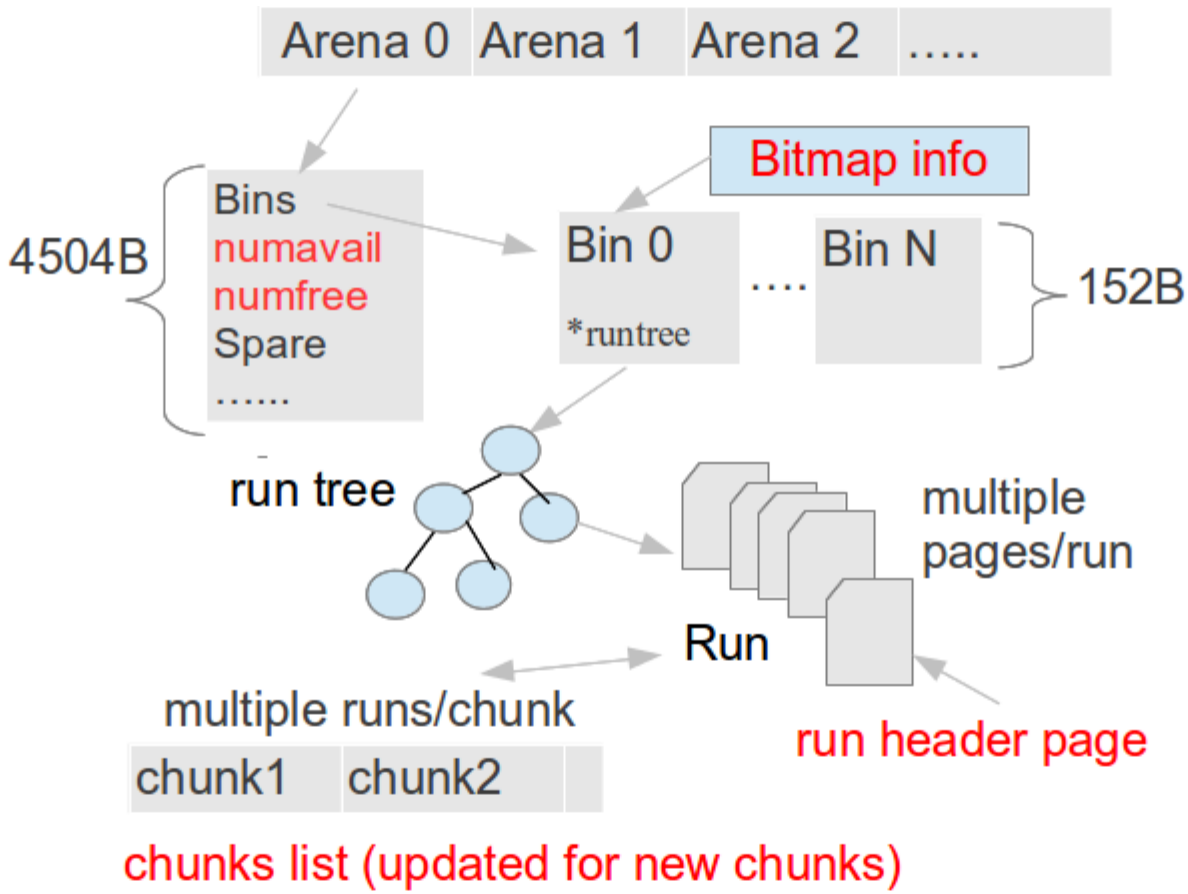
e = nvalloc(sizeof(struct entry)); → Allocator overhead

BEGINTRANS((void *)e,0); → Transactional overhead

Cost of Persistence

- User level Overheads
  - **Allocator metadata maintenance**
  - Restart/ Recovery Swizzling
- Transactional (Durability) Overheads
  - Logging
  - Substantial code changes
  -
- Kernel level Overheads
  - Kernel metadata maintenance
  - Kernel metadata swizzling

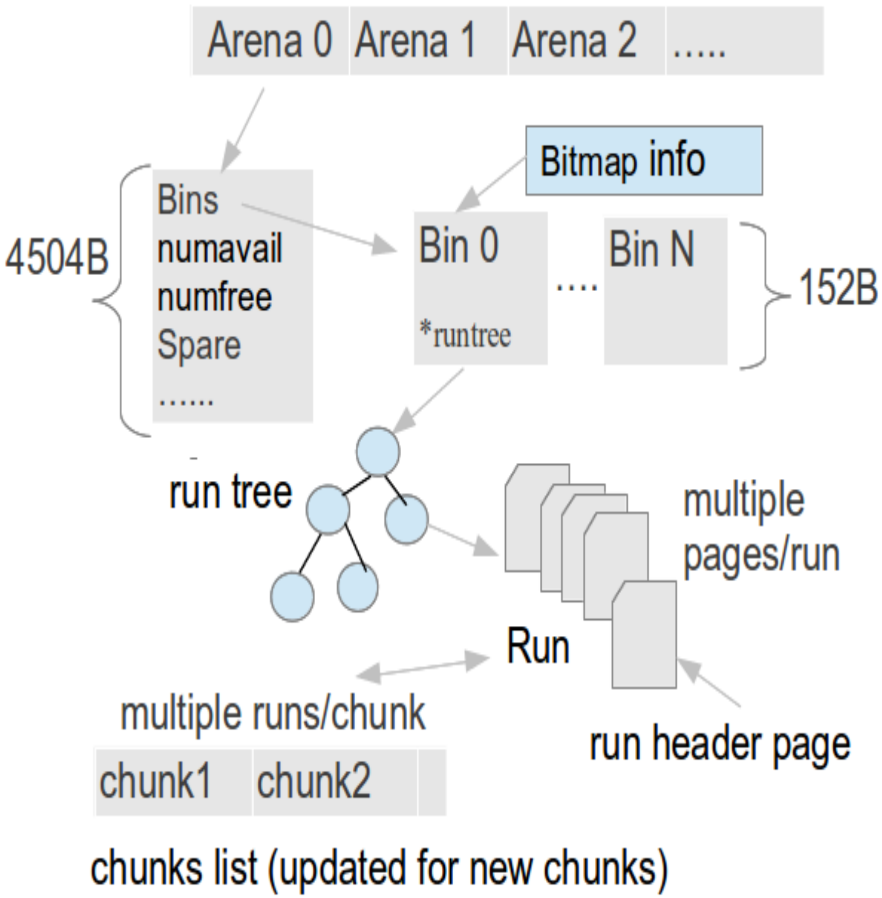
Allocator Overhead



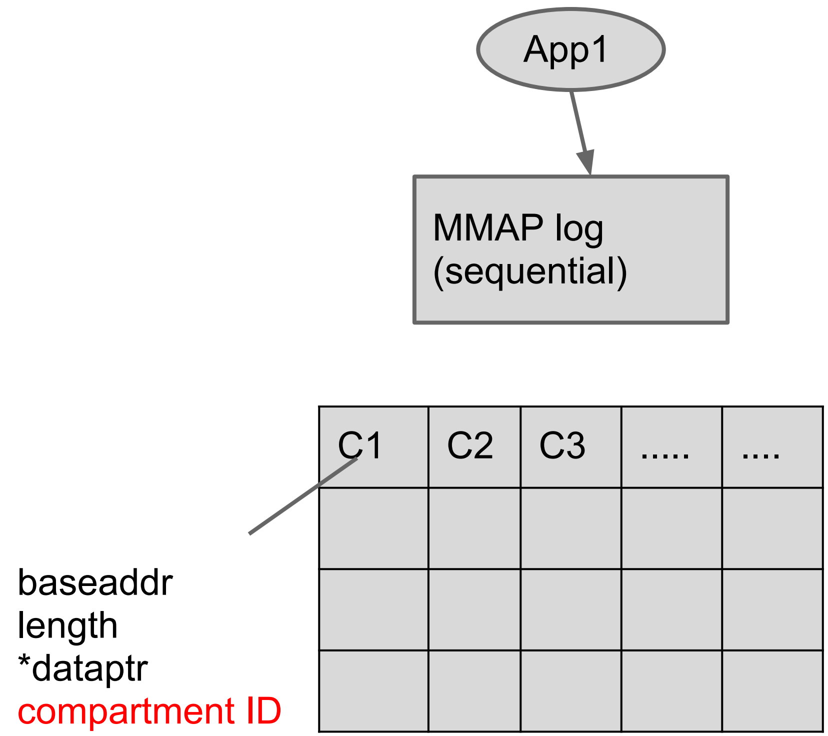
Problem: Complex allocator metadata in PCM, High random writes,
High Cache miss rate

Proposed Allocation

Complex allocator state in DRAM



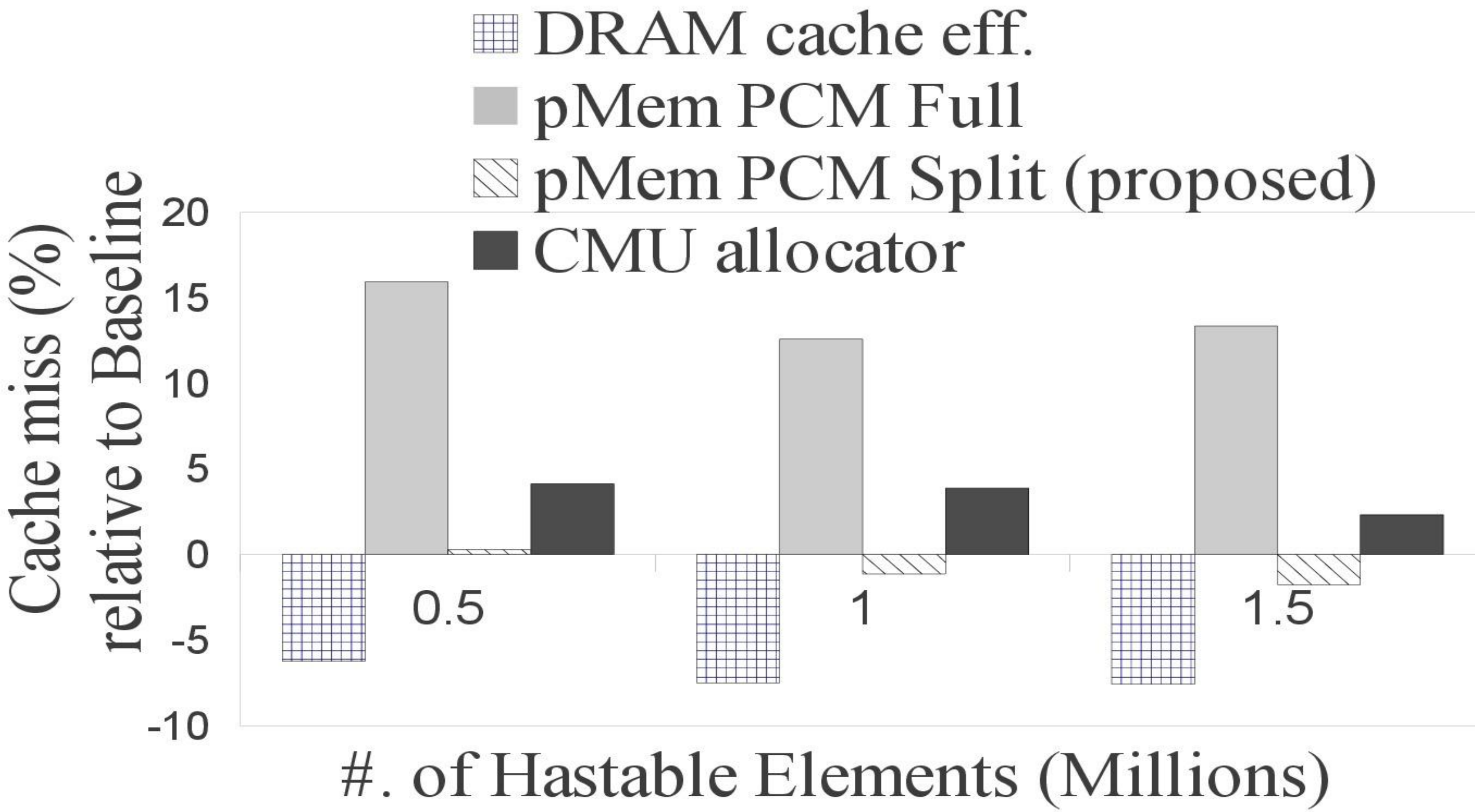
2 level allocator
Metadata log in PCM



Chunk log - sequential

Insert - O(n)
Lookup- O(log n) + C

Proposed Allocation



Reduction in Cache Flush: 8X

Cost of Persistence

- User level Overheads
  - Allocator metadata maintenance
  - **Restart/ Recovery Swizzling**

- Transactional (Durability) Overheads
  - Logging
  - Substantial code changes
  -

- Kernel level Overheads
  - Kernel metadata maintenance
  - Kernel metadata swizzling

Swizzling - Recovery overheads

During Reboot,

Lets say process heap starting address is 2000

```
hash_s *hashtable =  load_entire_hashtable("hashtable_root")

cout << "hashtable ptr << endl;
```

prints incorrectly 1000, should be >= 2000

Swizzling - Recovery overheads

Normal Execution:

```
hash_s *hashtable  = nvmalloc( size, "hashtable_root");
for each new entry:
        entry_s *entry =  nvmalloc( size);
        hashtable[count] =  entry;
        count++

cout << "hashtable ptr << endl; ──────► prints 1000

SYSTEM CRASH
```

Traditional Recovery - Serialization

Requires extensive modification of datastructures

Substantial I/O calls, and more OS interaction

Two phase overhead:

1. serialization when saving data
2. deserializationfor recovery
3. kills byte addressability
4. Can increase overhead upto 20% each phase

Prior Work: Swizzling during application execution

Proposed Solution - Lazy Swizzling

- Lazy/ On demand pointer swizzling
- Use allocator metadata as history of previous allocation
- On restart, when a chunk is accessed, get its stale pointer value.
- See if stale pointer is in history (allocator log)
- If yes, map the state pointers to get new virtual address
- Convert the old state pointer to new pointer

Proposed Solution - Lazy Swizzling

```
h =   (struct hashtable *)nvalloc_("root_hash");

  for each entry in hash:

        LOADNVPTR(&key);
        LOADNVPTR(&value);
```

Benefits:

- No serialization of pointers required during commit
- Application decides what to load during restart
- Multiple level of pointer can be recovered
- Less than 10 % performance overhead during restart

Constant Virtual address

- Use same virtual address across sessions
- No requirement of pointer swizzling
- Requires static partitioning of NVM/PCM

Cost of Persistence

- User level Overheads
  - Allocator metadata maintenance
  - Restart/ Recovery Swizzling
- **Transactional (Durability) Overheads**
  - Logging
  - Substantial code changes
  -
- Kernel level Overheads
  - Kernel metadata maintenance
  - Kernel metadata swizzling

Durability overheads - Logging types

Log every write (in PCM) to overcome failures

Undo Logging

- Create a log, and copy the original data to log
- Modify the data in-place
- Upon failure before commit, restore stable log version
- Problems
    - Two writes for every single write
    - Random Writes

Durability overheads - Logging types

Write Ahead logging (most favoured and widely used)

- Create log and write sequentially to log
- When log fills up, log committed to original data

- Problems
  - Usually for heaps, every word is logged
  - High Log Metadata/ Log Data overhead
  - Metadata: 24bytes even for 8 bytes
  - Substantial Code changes

Prior Work: Word based or Object based logging

Write Ahead logging (WAL) in Heap

```
i = (unsigned int)LOAD(&h->entrycount);
STORE(&h->entrycount, i++);

if (LOAD(&h->entrycount) > h->loadlimit)
{
   hashtable_expand(h);

}
e = (struct entry *)nvmalloc(sizeof(struct entry));

STORE(&e->h, hash(h,k));
STORE(&e->v, v);
STORE(&e->next, h->table[index]);
STORE(&h->table[index], e);

COMMIT;
```

Proposed: Hybrid logging Heap

- Using only Word or Object based logging granularity not optimal (Why?)
- Combine Object and Word based logging with Undo Logging
- Maintain separate Object and Word based logs
- **Object based log: Less Log Metadata/ Log Data ratio**
- Word based log: Convenient for small changes (e.g., hash entry count)

Benefits: Hybrid logging Heap

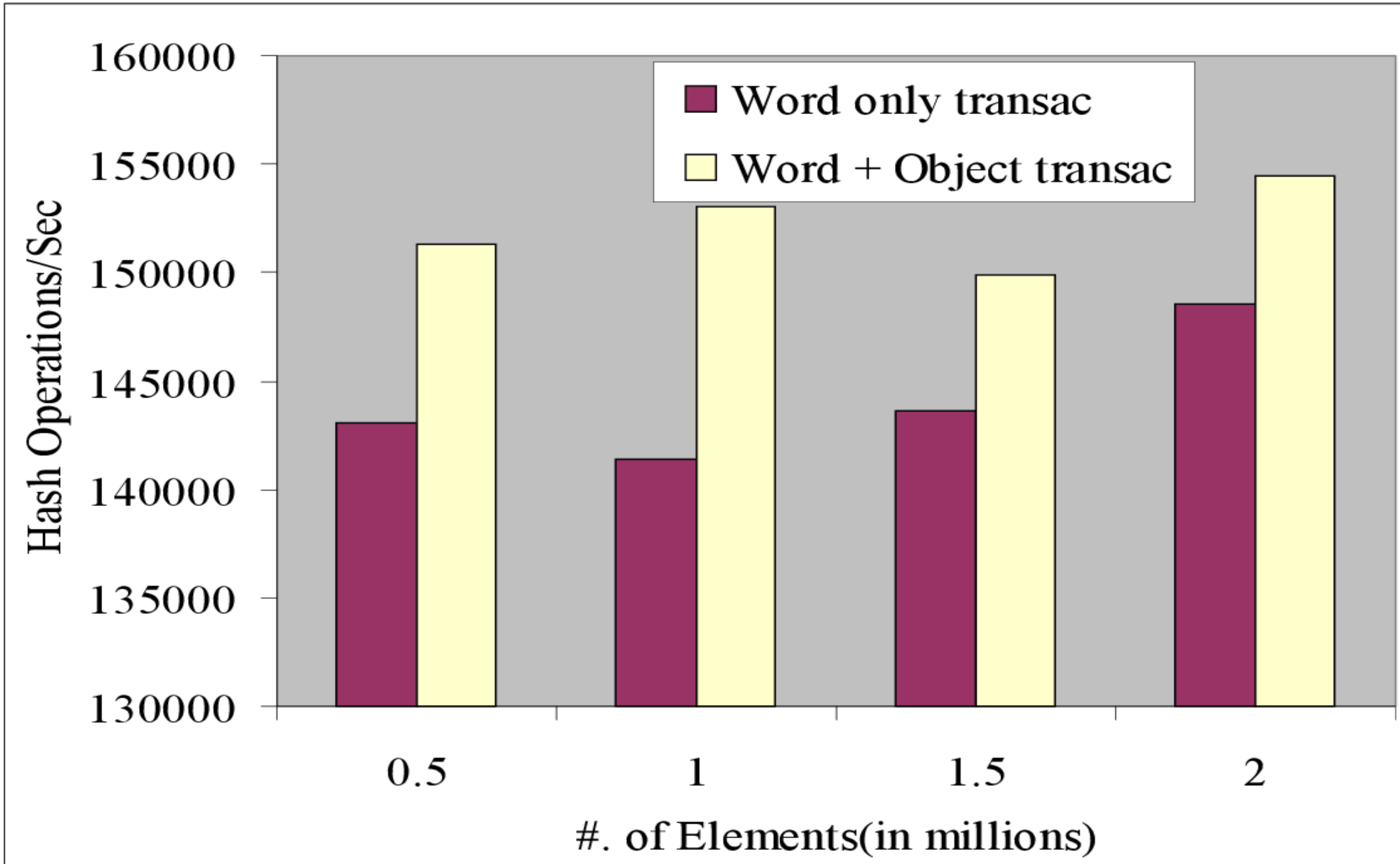
For Object based undo logging, easy dirt checking

- e.g, first time inserts

Object based allocator metadata used also for logging

No separate log metadata is required

Benefits: Hybrid logging Heap



Summary

Goal to reduce persistence overheads
Cache efficient NVM allocator
Lazy pointer swizzling to reduce serialization cost
Less than 10% swizzling overhead
Novel hybrid logging (Object + Word)
Improved I/O performance by 63%

More opportunities:
    Reducing Kernel Overheads
    Compiler optimizations

Questions / Comments

Thanks!