Systems Software for Rich Client Services via Persistent Memory

Sudarsun Kannan,
Ada Gavrilovska, Karsten Schwan
Georgia Institute of Technology

Georgia Institute of Technology

Motivation – Client Memory Usage

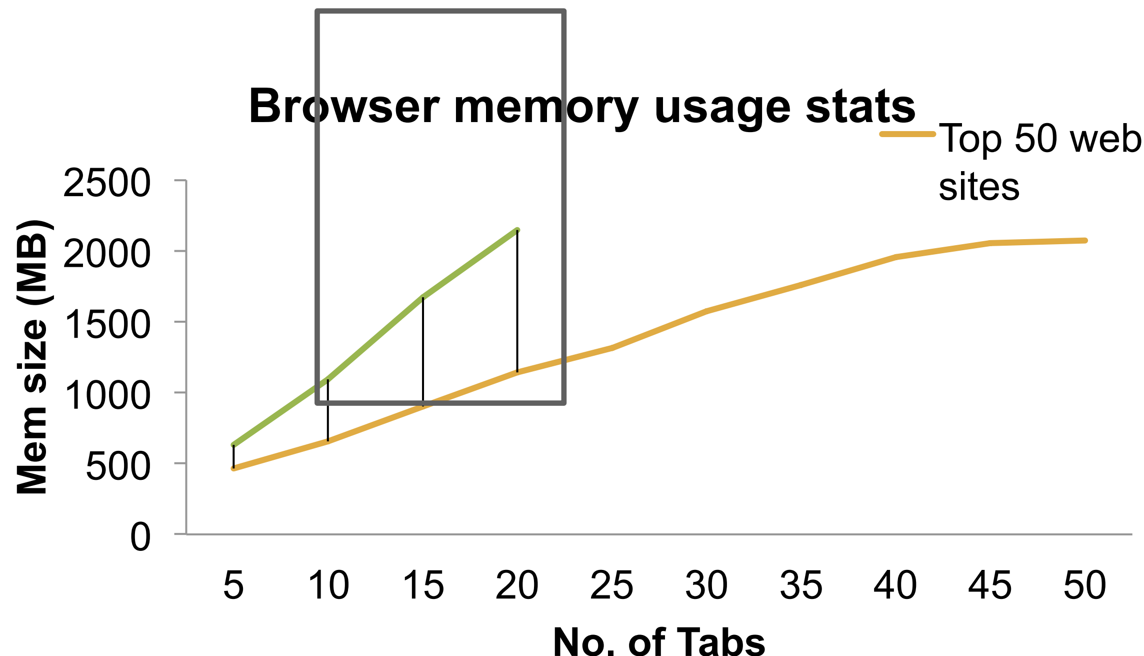
- Growing number of end client apps
  e.g., Webstore -33 million users, ~1 million apps

- Data-rich apps
  Picasa, Digikam, Face/Voice recognition, etc.
- Multi-threaded apps, to exploit increasing core counts
- Increased app memory usage
  App. features and data
  Browsers and plugins are memory hungry
  Google Chrome native client, Intel parallel JavaScript

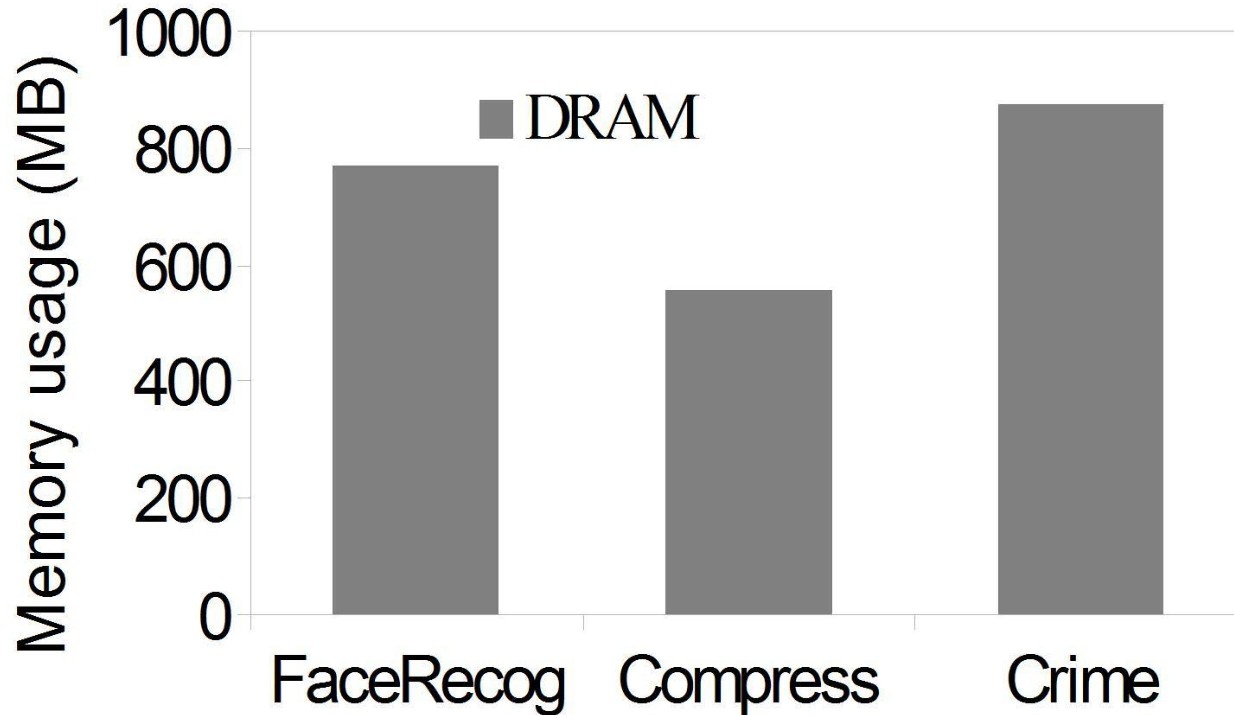
- Severe persistent data storage bottlenecks (and overhead)
  External Flash ~4- 16 MB/Sec (FAST' 11, Kim et al.)
  Browsers - substantial sandboxing overheads

Motivation – Memory Usage

- Membust benchmark in Google Chrome
  - Experiments using Alexa Top 50 and Webstore apps.
  - Average memory usage (RSS) 900 – 1500 MB!



Motivation – Memory Usage

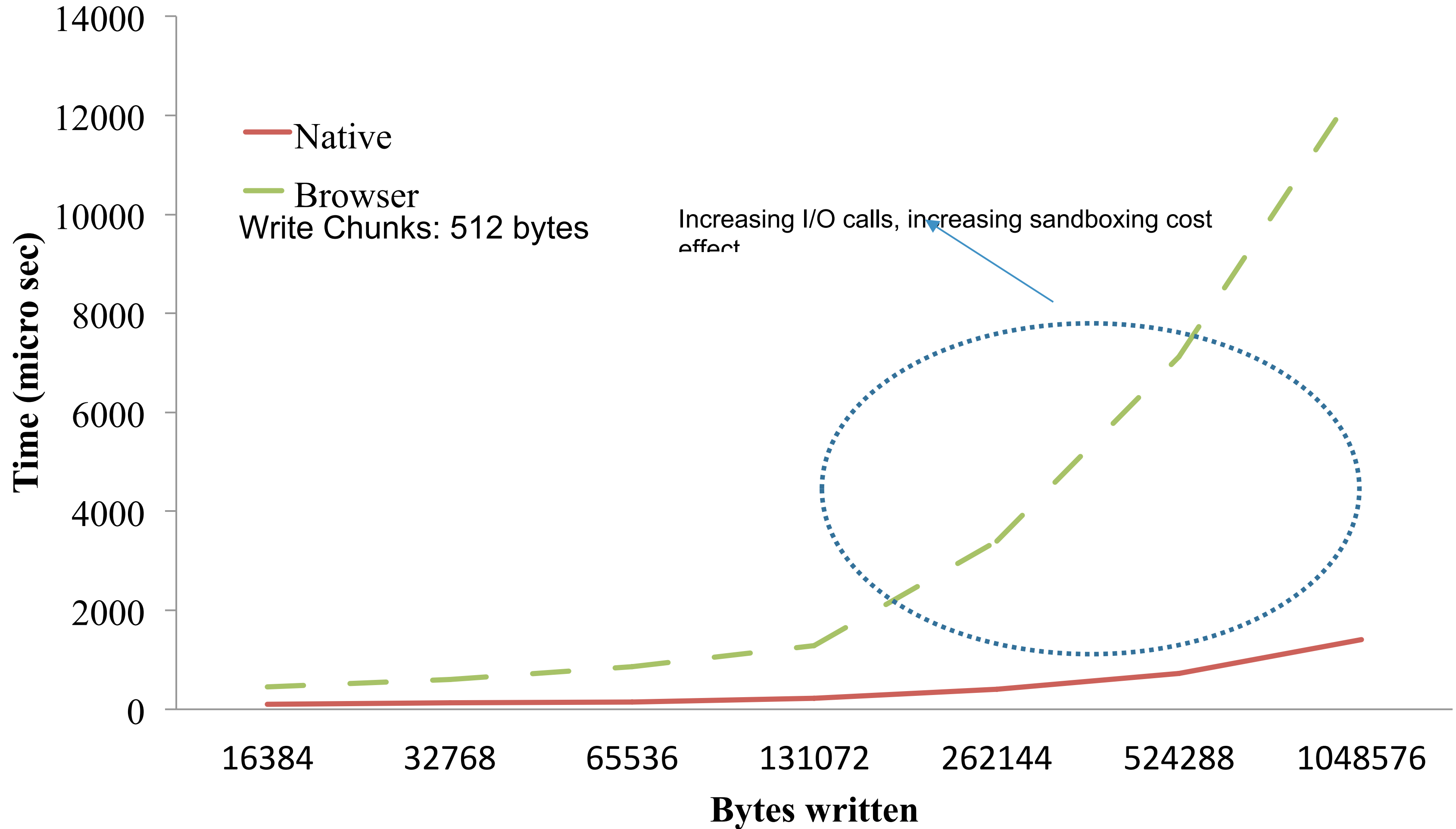


FaceRecog: Memory usage dominated by input data sets

Compress: X264 compression, parallel threads, memory usage

Crime mash-up: Simple multithreaded parallel search on public crime database

Motivation – I/O Sandboxing



Motivation: I/O S/W overheads

- High software overheads for block-based I/O interfaces
- End Client Apps: low per call data sizes, hence more calls
- Rarely use 'mmap' based interfaces
- Problems with 'mmap':
  - Every mmap/munmap call results in user/kernel transition
  - Requires several 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

Research Approach
NVM for Client Memory Capacity and Persistent State Challenges

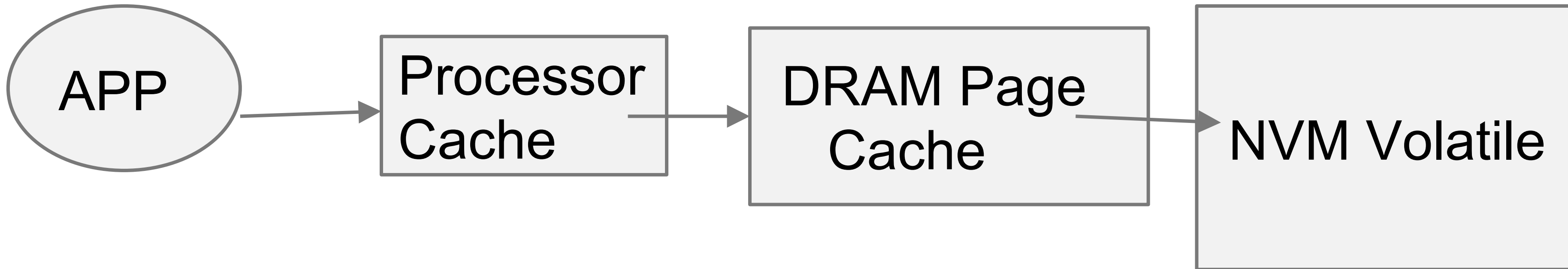
NVM technologies

- Byte addressable and persistent
- 2X-4X higher density compared to DRAM
- 100X faster compared to SSD
- Less power due to absence of refresh
- Byte addressability - (can be connected across memory bus and accessed with load/stores)

Limitations

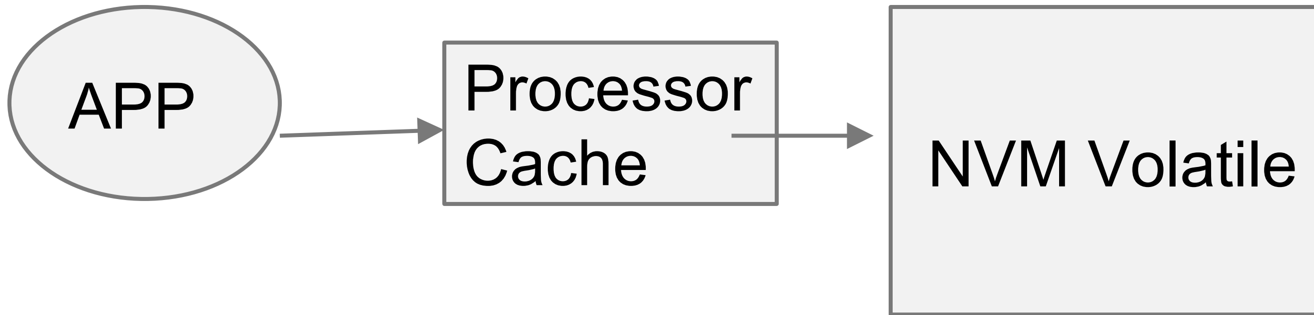
- Hight write latencies compared to DRAM
- 4X - 10X slower writes
- Limited endurance (approx. 10^8 writes/cell)
- Limited bandwidth: interface and device bottlenecks

Prior Work: NVM with DRAM Cache



- DRAM acts like a page cache
- Works well for server machines with TBs of DRAM
- ‘Capacity’ benefits

Prior Work: Fast Non Volatile Heap

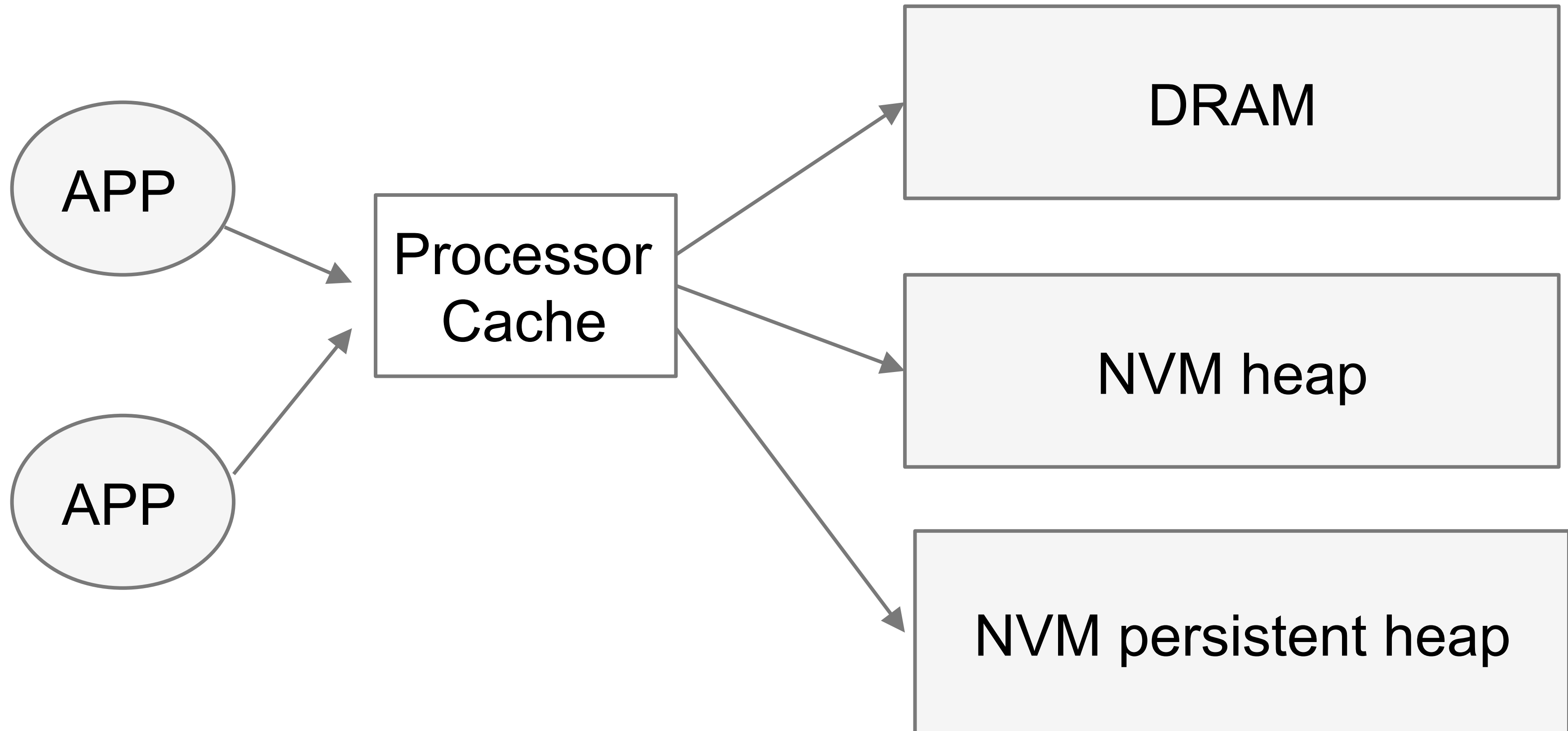


- Provides persistence, but

- Strong persistence guarantees require:
  - frequent cache flushing, NVM writes, memory fencing

- Outcome: high persistence management overheads
  - user and kernel level

Our Approach: pMem: Dual-Use NVM Capacity + Persistence



Processor cache plays crucial role in reducing write latency

Proposed: pMem: Dual-Use NVM

Key implementation ideas

- NVM as OS NUMA node
- `NVM node' dynamically partitioned into capacity + persistent heaps
- New applications APIs:
  - Applications explicitly use capacity/persistent NVM

=> NVM not exposed as I/O calls

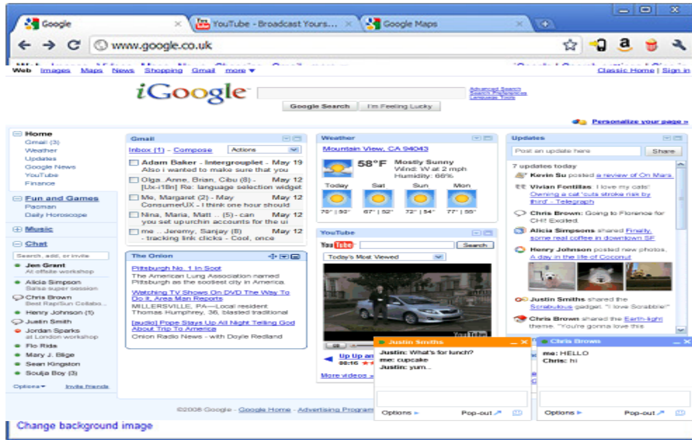
  - Goal: minimize software interactions for NVM access

Advantages

- Dual benefits: capacity + fast persistence
- Leverage hardware memory management support for NVM access

pMem - High Level View

Rich browser based client services



Native Client

NVM user library

DRAM manager

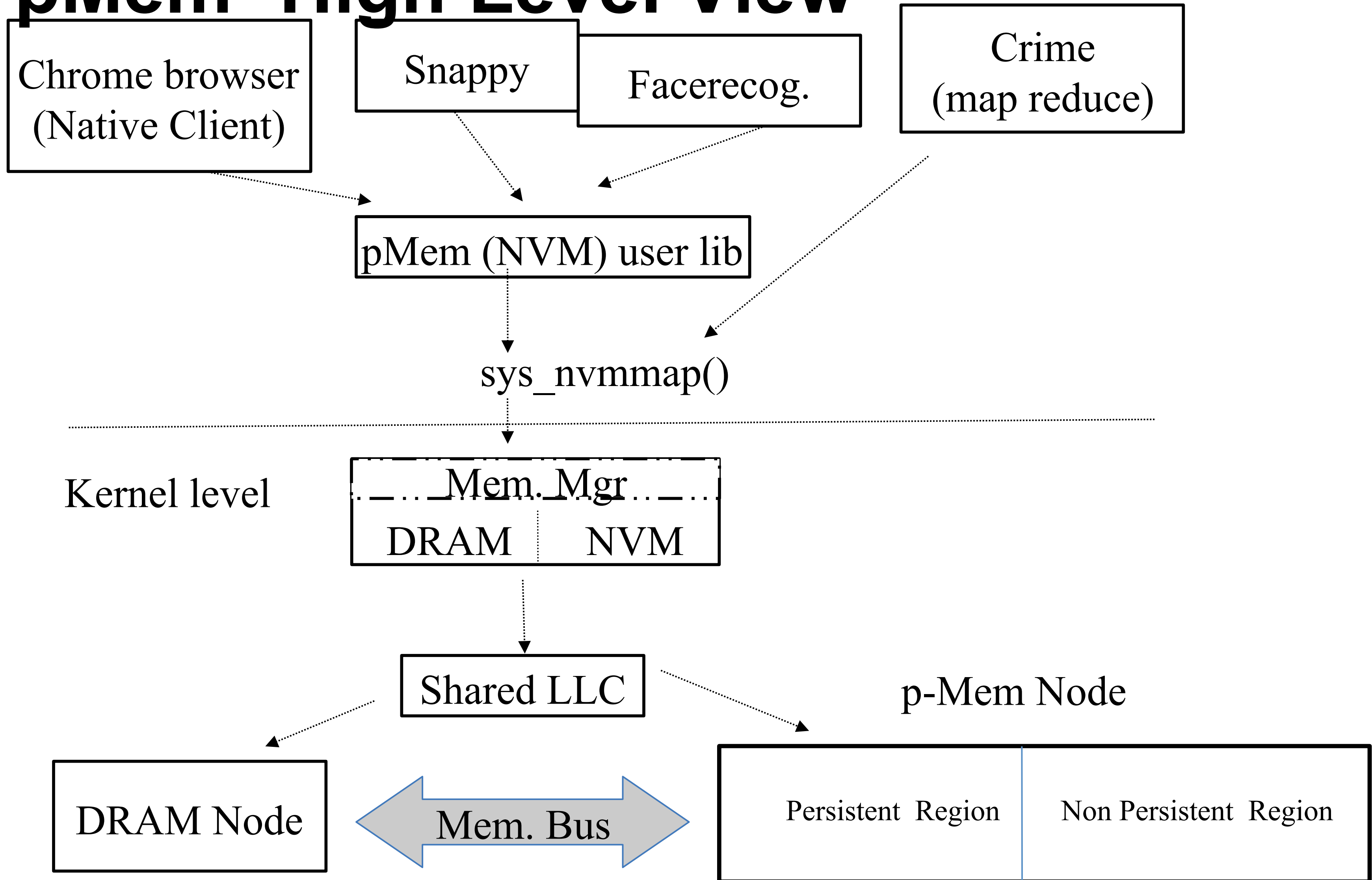
NVM manager

DRAM Node

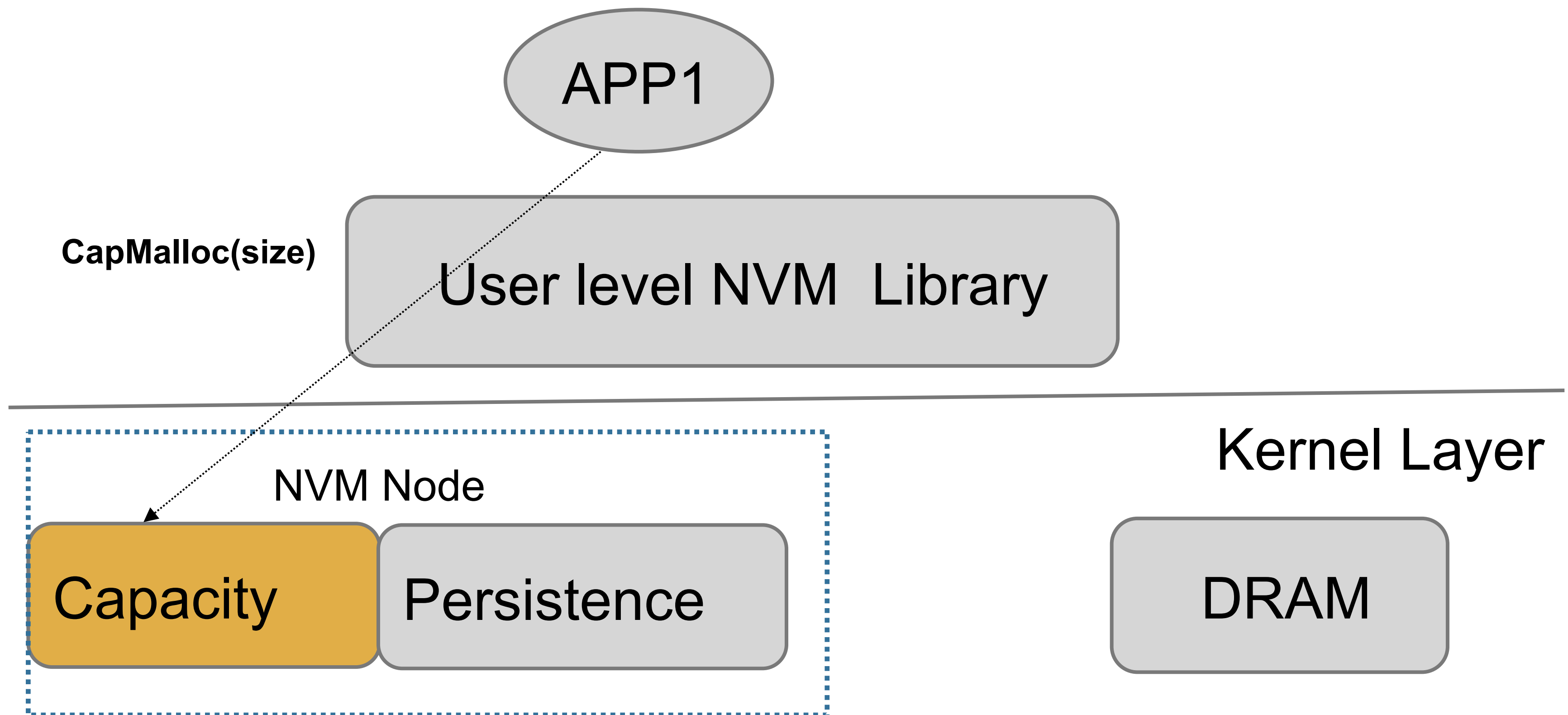
NVM node

With HIGHMEM and KERNEL Zones

pMem- High Level View

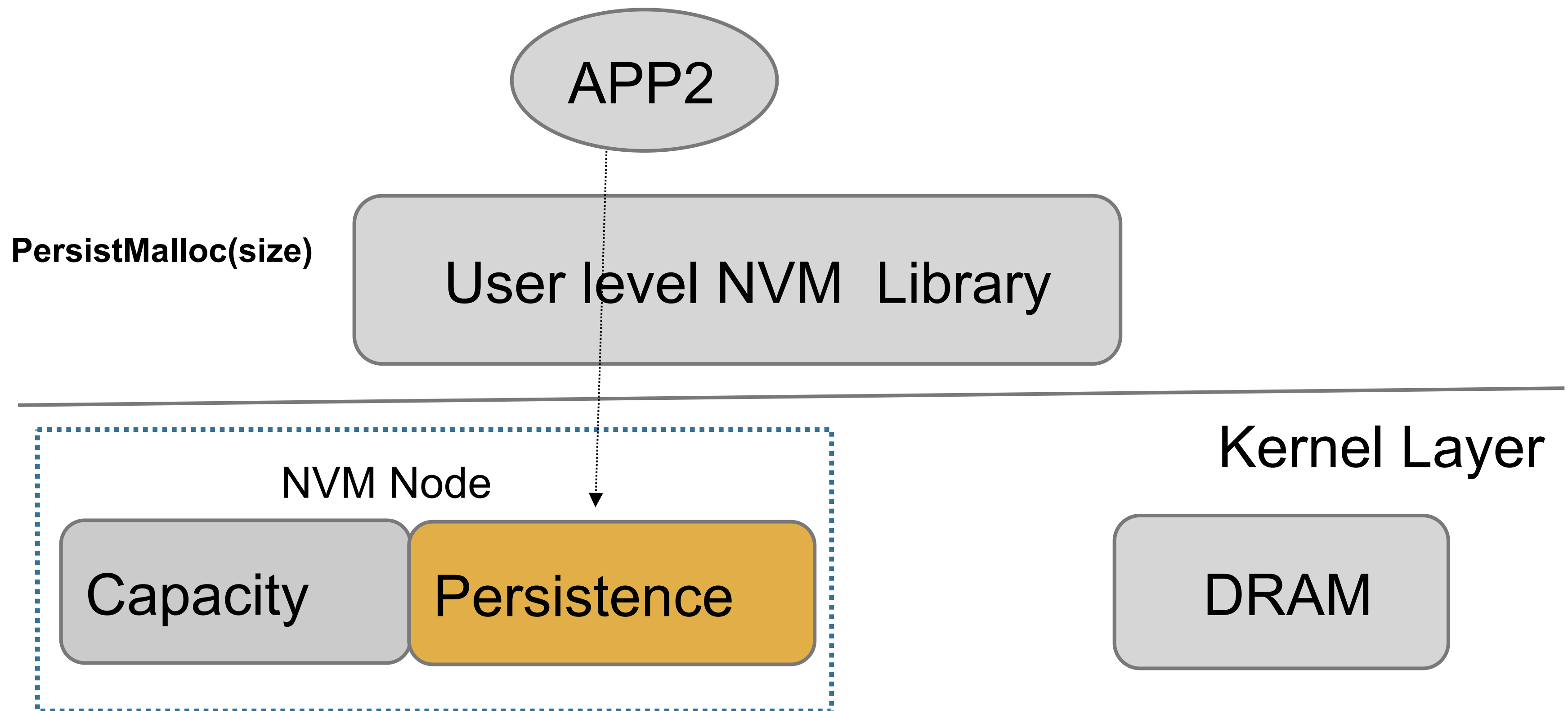


Using pMem: Capacity



- User and Kernel managers route application calls
- Application decides when to use NVM for capacity
  - NVM used as heap

Using pMem: Persistence



- Application decides when to use NVM for persistence
  - API calls
- Persistence metadata only maintained when needed

Proposed: Dual Use using pMem

Example: Persistent Hashtable using pMem

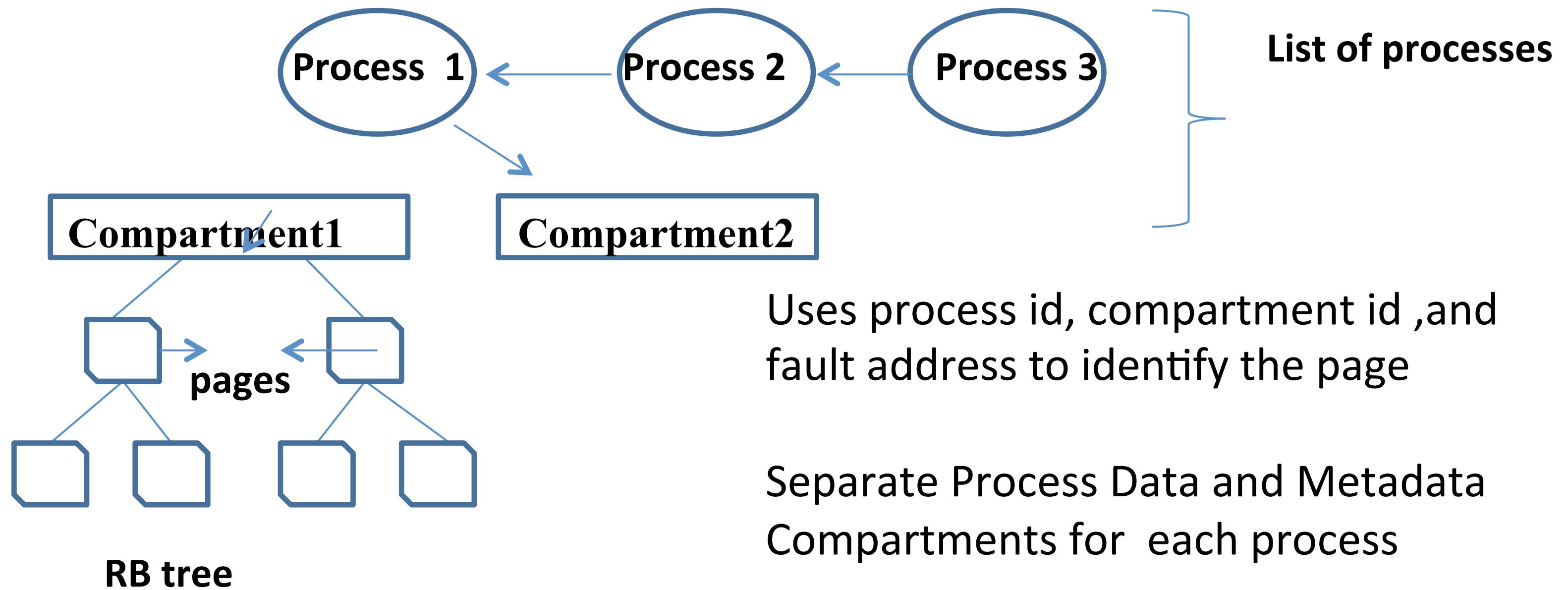
```
hash *table = PersistMalloc(entries, "tableroot");
    for each new entry:
            entry_s *entry = PersistMalloc(size, NULL);
            hashtable[count] =  entry;
            count++
```

Only root pointer of a data structure needs a name

pMem Software Architecture Design Principles

- OS supports separate NVM node
  - Clean system level abstraction for heterogeneous memory device
- Lightweight NVM memory manager
  - Handles NVM memory pages and maintains persistence structures
- NVM-specific allocation policies
  - Scalability and isolation from interference

Software Architecture – Kernel

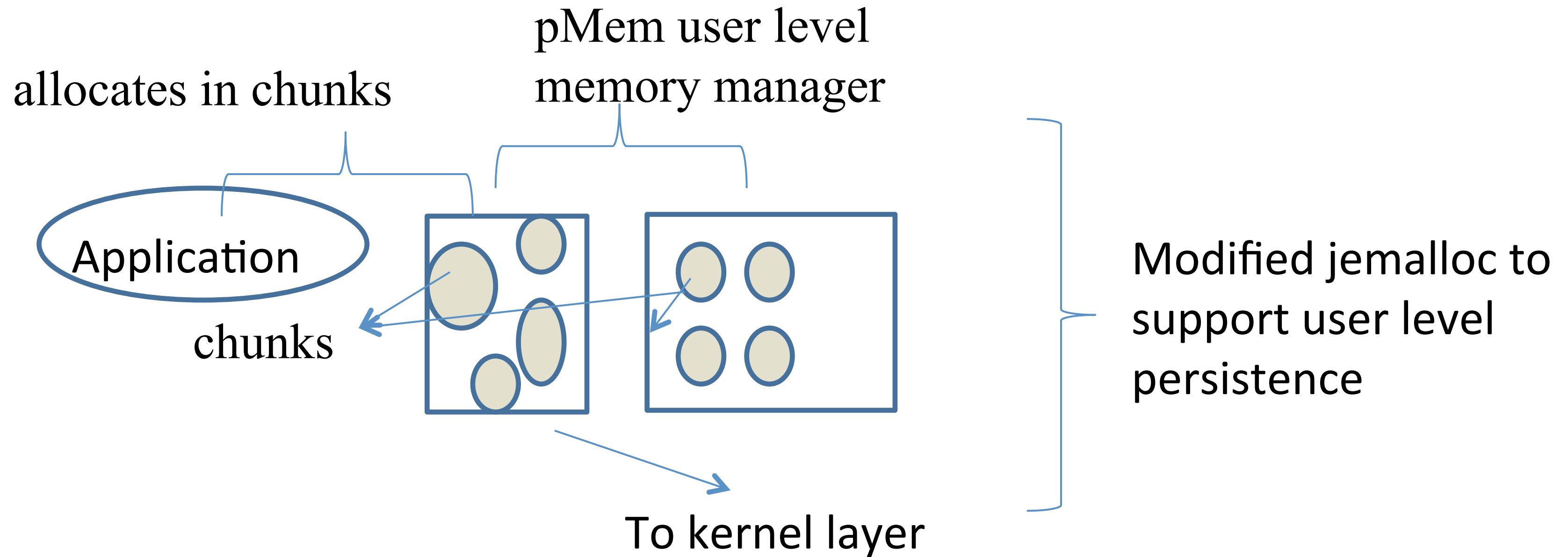


1 bit for each NVM page flag and 1 bit flush flag

Compartments:

- large region of NVM allocated by the user level NVM manager using *nvmmap*
- are virtual memory area structures (VMA),
- apps. can explicitly request separate compartments ('nvmmap')
- provides isolation b/w persistent and non-persistent NVM regions

Software Architecture – Allocator

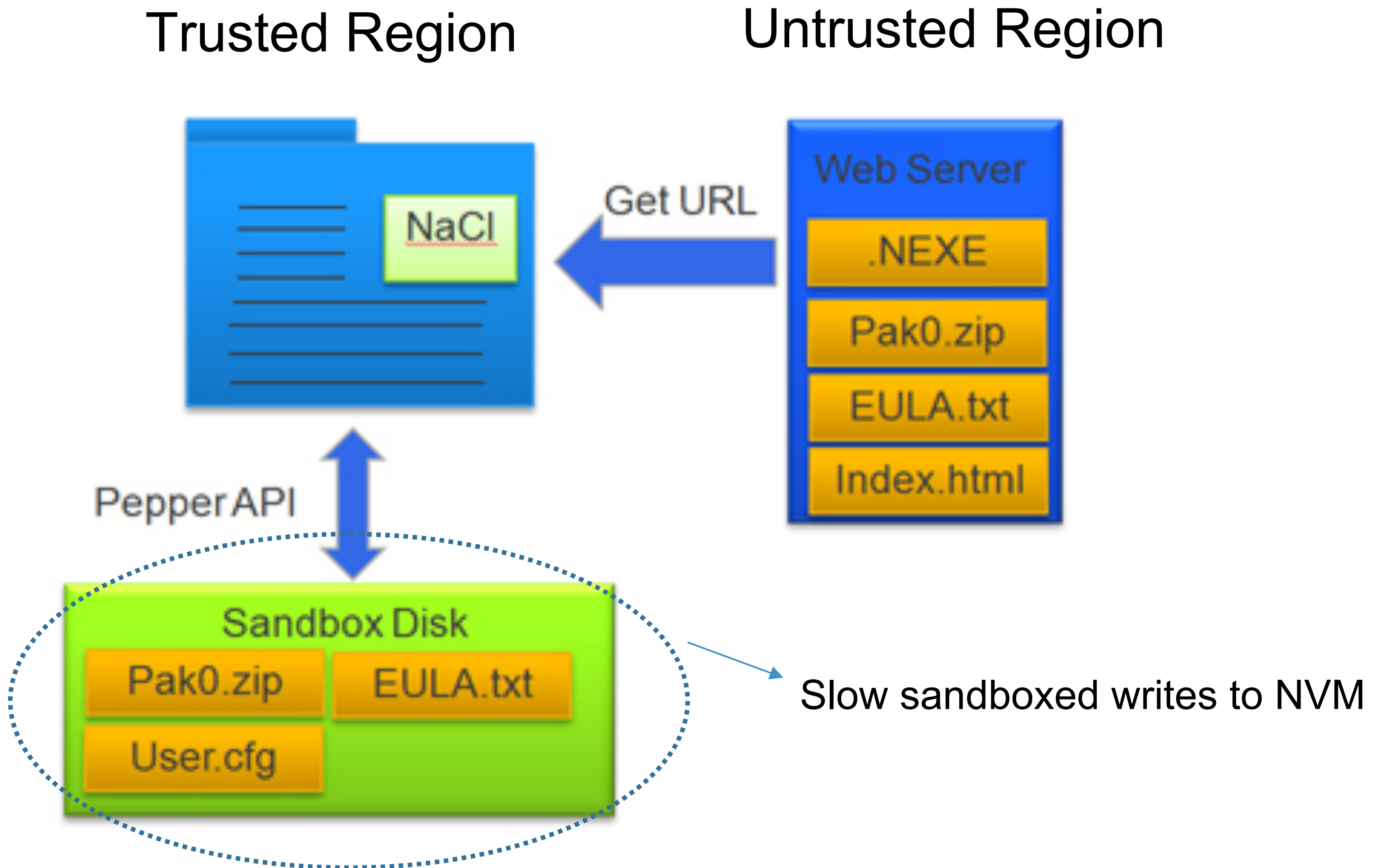


- Provides application interfaces like "capmalloc", "persistmalloc", "flushnvm"
- Manages application data in chunks
- Implemented by extending the jemalloc library

Consistency and Recovery

- Logging and lock-based transactions
- Lock-based transactions instead of STM
- Logging supports durability, pMem support UNDO and REDO logs, and hybrid (word + object-based) logs
- UNDO logging reduces code changes for heap-based use of pMem
- Recovery accomplished via lazy pointer swizzling

Support for Browsers



Browser Sandboxing and NVM

Key Idea

- By providing byte addressable heap, no need to trap every load/store software-controlled read/write
  - Create NVM heap for each untrusted plugin
  - Plugin can access any data within its heap
  - Only accesses outside its heap trap
- Avoids sandboxing each read/write call
- Performance results below

Implementation Comments

- Configure an OS NUMA node to emulate NVM
- Use ‘allocate on write’ policy
- All NVM pages locked, and swapping disabled

- For persistence:
  - All NVM pages are locked, swapping is disabled
    => persistence across application sessions
  - For persistence across boots, use SSD

Summary

- pMem addresses capacity + persistence needs
- Provides flexible interfaces to applications (capmalloc, persistmalloc)
- Treats NVM as a NUMA node, and exploits NUMA based allocation policies
- Provides support for browsers to reduce I/O overheads.

pMem Experimental Evaluation

Experiment Setup:

- Emulate NVM with DRAM-based NUMA node
- Persistence across sessions: prevent OS from reclaiming pages
- Account for NVM read/writes using PIN based instrumentation
- Use hardware counters to capture cache misses
- Additional use of simulations (MACSim) to understand cache misses

Intel Atom : Dual core, 1MB LLC, (8 way, Write Back, Shared LLC)
Applications pinned to cores

pMem Experimental Evaluation

Use cases

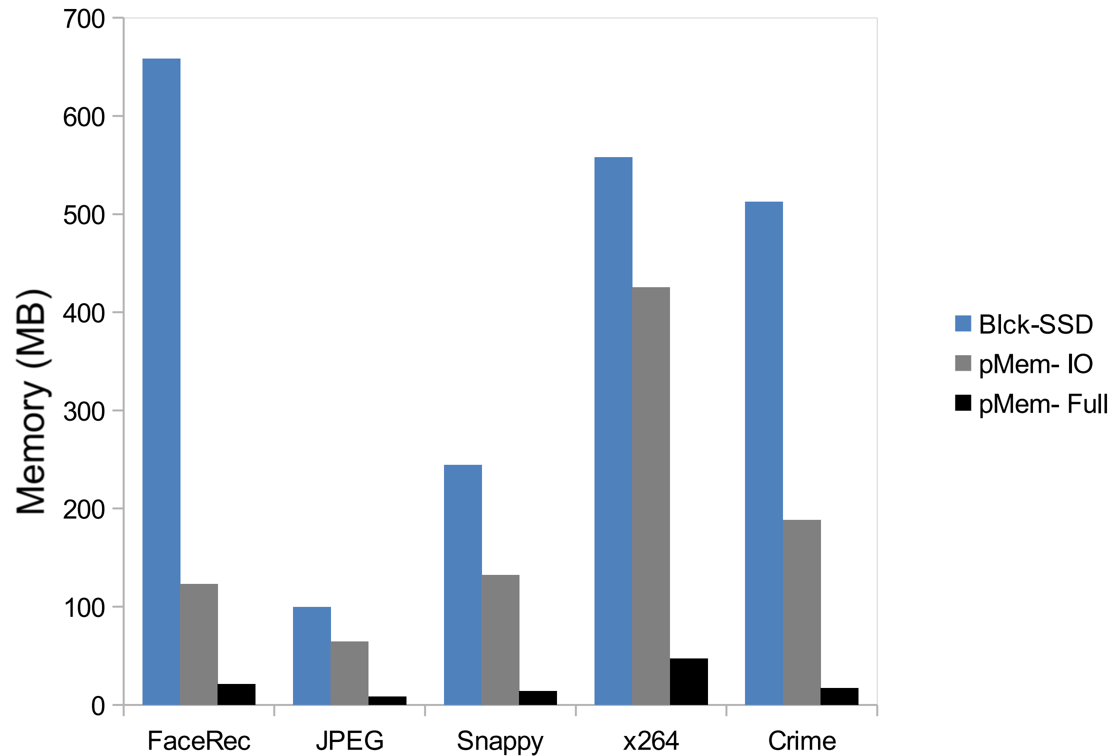
Scalability:
    Linux scalability benchmark for page allocation

Memory Capacity:
    Face recognition, Compression, Crime

Persistence:
    User behavior/preferences while browsing
      - persistent cross-session state
        compiled using ML methods

pMem DRAM Memory Usage

Performance: 4%-6% overhead



pMem for Persistence – Performance Gains

Execution time (sec)

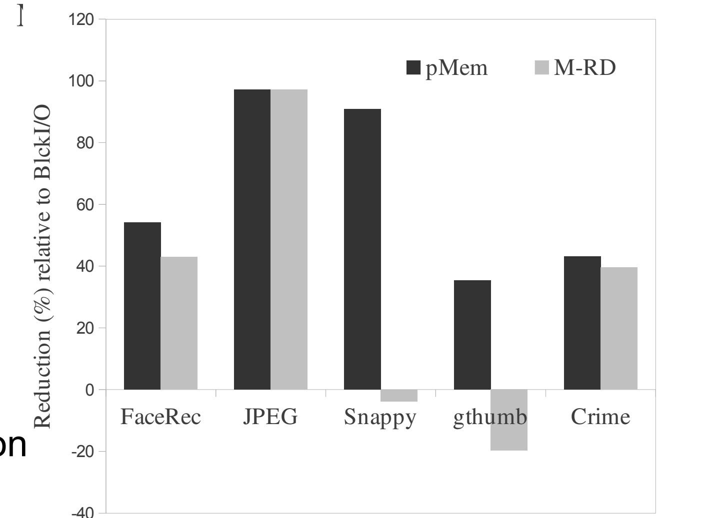
70
60
50
40
30
20
10
0

System time(sec)
User time (sec)

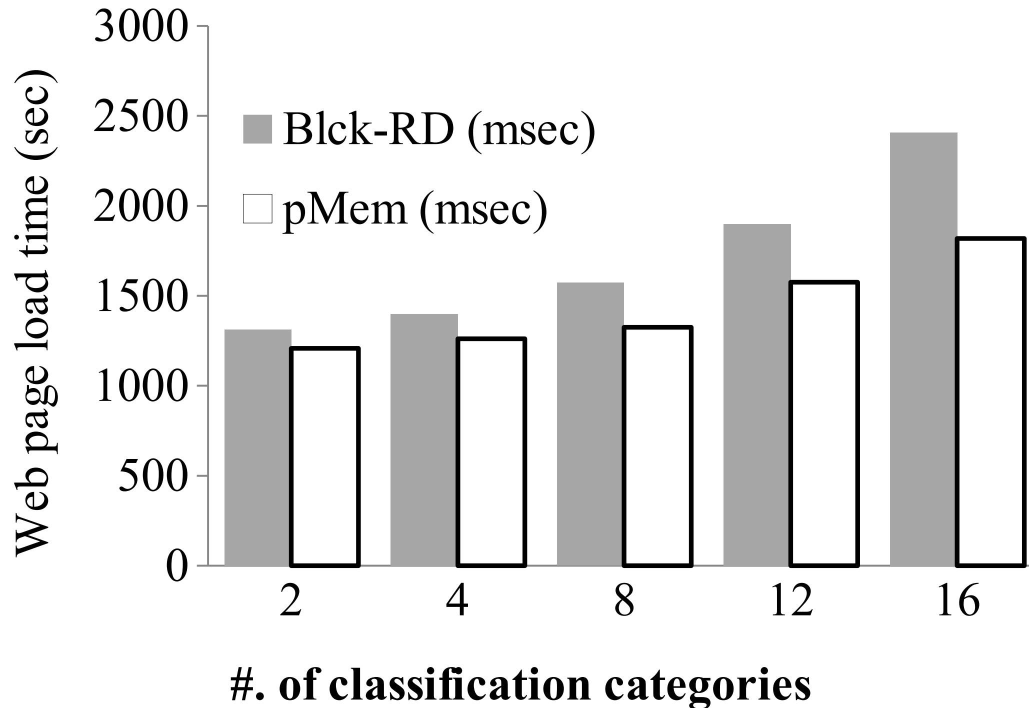
M-RD Blck-RD M-RD Blck-RD M-RD Blck-RD M-RD Blck-RD M-RD Blck-RD
M-SSD pMem M-SSD pMem M-SSD pMem

FaceRec JPEG Snappy

User-kernel switch reduction relative to Blck-IO



Cost of Recovery Mechansims



~45% improved performance compared to using SSD
~62% improvement for persistent hashtables
Increased data size => increased persistence cost

Function	pMem	Block
Learn	8.337921	12.3453
Logging	1.22304	-NA-
Cache Flush	0.00232	-NA-

Table 5: Cost of Logging.

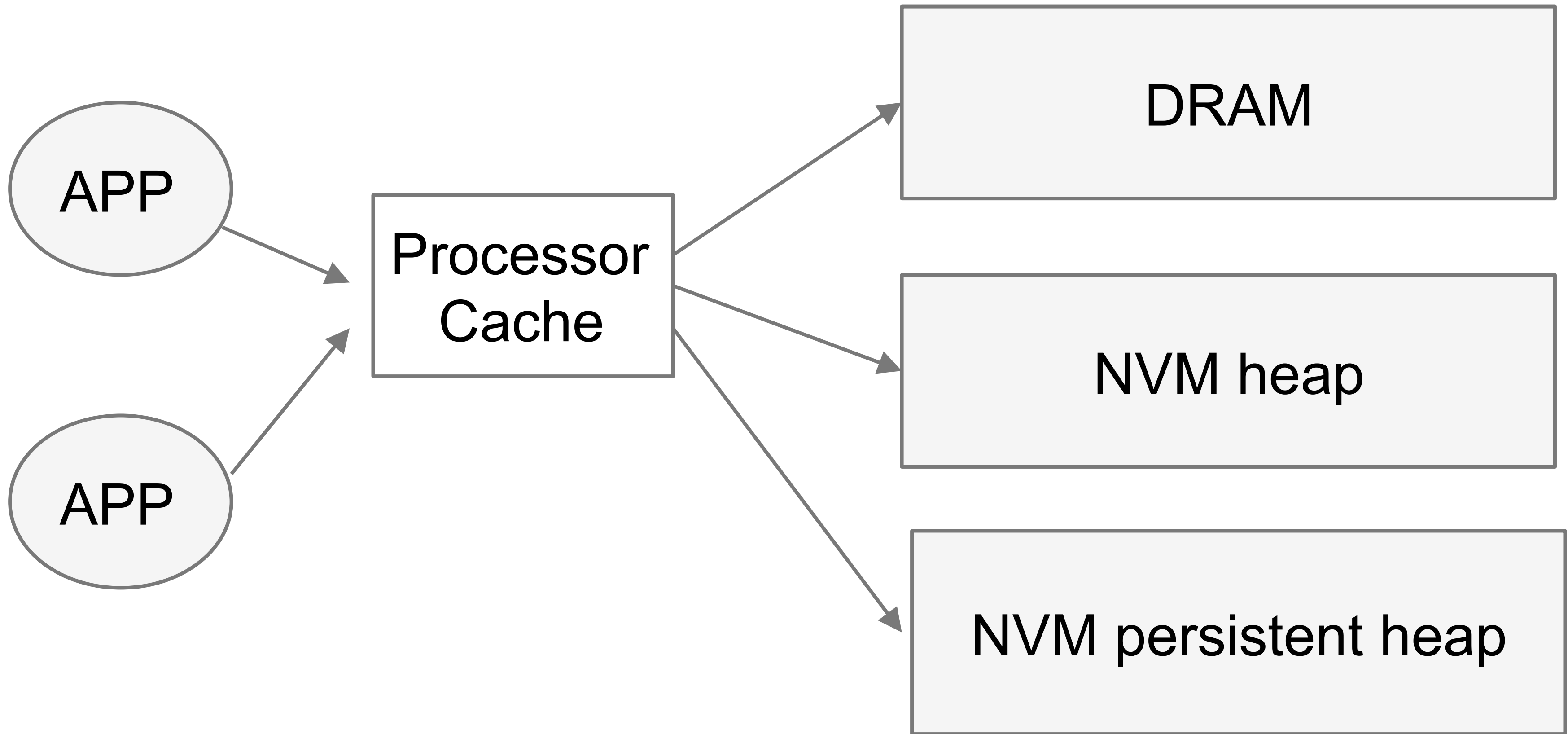
Summary of Results

Partitioned NVM: Capacity vs. Persistence

- up to 91% memory capacity benefits
- ~45% faster I/O for end client apps
- less that 6%-7% runtime overhead on some apps, compared to using DRAM

- But NVM should be ~100x faster!

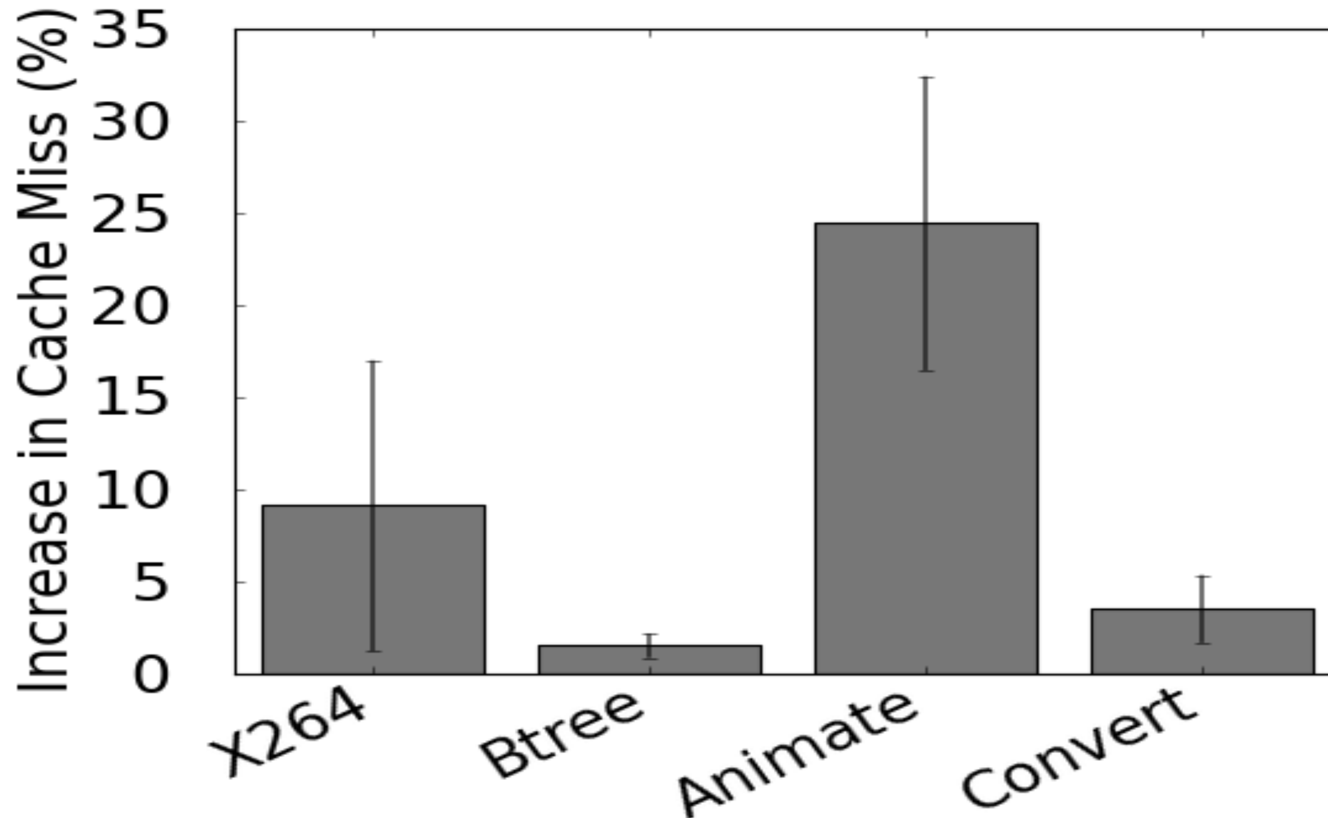
Next Steps: Persistence Overheads



Persistence requires frequent cache line flushing

=> cache sharing a problem?

Cache Sharing – Performance Effects



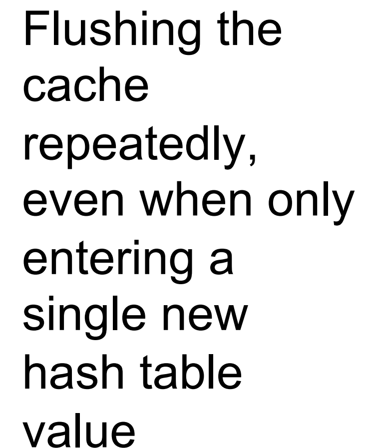
Persistent application: Hashtable with 1M Operations (puts and gets)
Intel Atom : dual core, 1MB LLC, (8 way, Write Back, Shared LLC)
Persistent and capacity applications pinned to their cores

Frequent Cache Flushes

```
AddHash_Entry() {
//Fence and Flush log (in NVM).
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);
```



Additional Persistence Overheads

```
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);
```

Transactional overhead

Allocator overhead

Transactional overhead

Persistence Overheads - Summary

- Cache Flushes
  - Cache partitioning? Logging and bundling?

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

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

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

Next steps

- Many interesting open questions

- Power model
- Client vs. datacenter/server vs. HPC pMem stack
- From single node/single NVM node to multi node heterogeneous systems.

Questions/Comments

Thanks!

Hybrid logging

```
AddHashEntry() {
  ID = begin_trans(”word”);
   ++(table->entrycnt);
   commit_trans(ID, &table-> entrycnt);

  key = (char *)nvalloc(64);
  val= (char *)nvalloc(4096);

  ID1 = begin_trans(”object”);
  memcpy(val, page, 4096);
  commit_trans(ID1, value);

  ID2 = begin_trans();
   table->k = key;
   table->v = val;
   commit_trans(ID2,table);
}
```