Systems Software for Rich Client Services via Persistent Memory

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan
Georgina 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!
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

Browser I/O vs. Native I/O

- Increasing I/O calls, increasing sandboxing cost effect
- Write Chunks: 512 bytes

Bytes written

<table>
<thead>
<tr>
<th>Time (micro sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>16384</td>
</tr>
<tr>
<td>32768</td>
</tr>
<tr>
<td>65536</td>
</tr>
<tr>
<td>131072</td>
</tr>
<tr>
<td>262144</td>
</tr>
<tr>
<td>524288</td>
</tr>
<tr>
<td>1048576</td>
</tr>
</tbody>
</table>

- Native
- Browser
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.

<table>
<thead>
<tr>
<th>App.</th>
<th>Avg. Write Size</th>
<th>Avg. Read Size</th>
<th>Read Count</th>
<th>Write Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG</td>
<td>27</td>
<td>4096</td>
<td>146212</td>
<td>10000</td>
</tr>
<tr>
<td>OpenCV</td>
<td>0</td>
<td>1045256</td>
<td>765</td>
<td>0</td>
</tr>
<tr>
<td>Snappy</td>
<td>121307</td>
<td>121307</td>
<td>11108</td>
<td>11108</td>
</tr>
<tr>
<td>x264</td>
<td>152792</td>
<td>153600</td>
<td>1164</td>
<td>388</td>
</tr>
<tr>
<td>Mapreduce</td>
<td>0</td>
<td>67108864</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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
- High 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

Chrome browser (Native Client) -> Snappy -> Facerecog. -> pMem (NVM) user lib -> sys_nvmmmap() -> Kernel level

Kernel level: Mem. Mgr
- DRAM
- NVM

DRAM Node <-> Mem. Bus <-> Shared LLC

p-Mem Node
- Persistent Region
- Non Persistent Region
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, "tablroot");
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

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

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

Separate Process Data and Metadata Compartments for each process

1 bit for each NVM page flag and 1 bit flush flag
Software Architecture – Allocator

- Allocates in chunks
- Managed by a pMem user level memory manager
- Modified jemalloc to support user level persistence
- To kernel layer

- 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

Trusted Region

Untrusted Region

Slow sandboxed writes to 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

![Graph showing memory usage for different applications.](image)
pMem for Persistence – Performance Gains

- Comparing pMem with current SSD devices (1) RAMDisk-based mmap interface (M-RD), (2) SSD, compared to all applications (see Figure 7): using a better understanding of memory capacity benefits, with four cases.

- The use cases are the same as those used for traditional approaches when applications access and execute directly on NVM. The use cases are the same as those used for unbuffered approaches.

- 5.4 Impact on Performance:

- Input size.

- Buffers are required and the benefits increase with the database. When using pMem, no such intermediate buffer is needed. Each thread, which is approximately 2/3 of the input database, is stored in a portion of intermediate storage buffer for each thread, which is approximately 2/3 of the input database.

- Database to a portion of intermediate storage buffer for each thread, which is approximately 2/3 of the input database.

- In case of the multi-threaded crime application using map-reduce, for the calculation of the training database to a portion of intermediate storage buffer for each thread, which is approximately 2/3 of the input database.

- The ratio of NVM loads to stores is around 17:1 (Figure 8). Read-intensive workloads are less impacted by noise.

- Although FaceRec is a computationally intensive task, the user-time for both approaches is the same. One thing to note is that, due to lesser page access latency, the gains from pMem's NVM-full for intermediate data can reach up to 91.5% of consumption for the majority of DRAM space. Here, using pMem's NVM-full for intermediate data can result in a significant reduction in memory usage for the input data set, whereas current block-based methods do not offer the same level of benefits.

- x264 uses several shared data structures for each frame, which consumes the majority of DRAM space. Here, using pMem to reduce memory usage by 91.5% of x264's shared data structures can result in a significant reduction in memory usage for the input data set, whereas current block-based methods do not offer the same level of benefits.

- pMem-DB outperforms the M-SSD approach, data needs to be loaded from a database to a portion of intermediate storage buffer for each thread, which is approximately 2/3 of the input database.

- The left graph in Figure 7 shows the execution time (sec) for each application, and the right graph shows the user-kernel block time reduction (in percent) relative to Blck-IO. For Snappy compression, although the total input data size is same as for the JPEG application, file sizes vary, with a total input data size of approx. 2x for the pMem approach compared to M-SSD. For FaceRec, the input size is dependent on the input size, and hence when using pMem, the benefits are moderate (approx. 45.9%), whereas for Snappy and JPEG, the benefits are significant (approx. 45.9% for Snappy and 35.33% for JPEG). In case of the multi-threaded crime application using map-reduce, the benefits are moderate (approx. 45.9%).
## Impact of Logging

We use the same approach, the number increases by 10% for Snappy. This explains the 34% performance improvement of pMem over M-RD.

Clearly, the pMem based interface reduces the number of transitions by up to 90%, whereas for the MMAP approach, the number increases by 10% for Snappy. This further improved with asynchronous log flushing, like what is done with 'fsync'.

We observe that pMem results in superior performance mainly because the block based approach makes several heap based library based interfaces, there is no need for system calls for every input. To understand the difference in runtime, we evaluate the cost of providing persistence with transition data and kernel persistent data structure log-mechanism. When using pMem, we enable both application data and kernel data.

### 5.6 Summary of Results

<table>
<thead>
<tr>
<th>Function</th>
<th>pMem</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn</td>
<td>8.337921</td>
<td>12.3453</td>
</tr>
<tr>
<td>Logging</td>
<td>1.22304</td>
<td>-NA-</td>
</tr>
<tr>
<td>Cache Flush</td>
<td>0.00232</td>
<td>-NA-</td>
</tr>
</tbody>
</table>

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
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);

Flushing the cache repeatedly, even when only entering a single new hash table value.
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);
}
Persistence Overheads - Summary

• Cache Flushes
  - Cache partitioning? Logging and bundling?

• User level Overheads
  o Allocator metadata maintenance
  o Restart/Recovery Pointer Swizzling

• Transactional (Durability) Overheads
  o Logging
  o Substantial code changes

• Kernel level Overheads
  o Kernel metadata maintenance
  o 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);
}