21 – Persistent Memory

CS 736

Storage: Past, Present, Future

Storage-Class Memory
(slow, block, persistent)

Memory
(fast, byte, volatile)

NVM Technologies

Flash-backed DRAM
Phase-Change Memory

Spin Torque MRAM
Resistive RAM
Flash

Latency

ns
μs

• Persistent
• Short access time
• Byte Addressable

Flash-backed DRAM (NV-DIMM)

• Store data within regular DRAM backed by NAND Flash

Transfer data upon power failure

Normal Operation

supercaps
Phase-Change Memory (PCM)

- Store data within phase-change material
  - Amorphous phase: high resistivity (0)
  - Crystalline phase: low resistivity (1)
- Set phase via current pulse
  - Fast cooling → Amorphous
  - Slow cooling → Crystalline

Resistive RAM (RRAM)

- Store data by dissolving ions within electrolyte memristive material (e.g., TiOx)

Phase-Change Memory

- Key properties
  - Reads: 100 – 300 ns
  - Writes: 10 – 150 us
  - Endurance: $10^8$
  - Density (expected): Medium/High
  - Cost: Medium (few $/Gb$)

Resistive RAM (RRAM)

- Key properties
  - Reads: 30 ns – 2 us
  - Writes: 100 ns – 2 us
  - Endurance: $10^{10}$
  - Density (expected): High
  - Cost: Very high ($5,000/Gb$)
Implications to Software

- Persistent
- Short access time **Software overhead matters**
- Byte Addressable **Accessible via loads/stores**

Devices: NVM-based SSDs

- Challenges
  - Hardware interface
  - Software latency
  - Protection and user-mode access
- Examples:
  - Moneta-D
  - NVMExpress

NVM as SSD

NVM SSD Challenges

- Software overheads in kernel

Baseline Latencies:
- Hardware: 8.2 us
- Software: 13.4 us

Software is Critical

[Caulfield, SC’10]
OS Scheduling Overhead

- I/O scheduling
- Locking
- Interrupts and waiting
- Example: Moneta [MICRO’10]

User-mode Data Access

- Challenge: metadata, I/O Latency

Remove FS Overhead

- Metadata calls still in kernel
- Kernel grants program access to data
- Program issues DMA directly to device
- Example: Moneta-D [ASPLOS’12]

NVM in the PC

- NVM and DRAM are addressable by the CPU
- Physical address space is partitioned
- Assume 64-bit atomic write
- NVM data may be cached in L1/L2

[Caulfield, SC’10]

[Caulfield, ASPLOS’12]

[Condit, SOSP’09]
Direct VS Peripheral Access

- App System call interface
  
  VFS

  ~Const. latency to NVM

  Variable latency to disk

<table>
<thead>
<tr>
<th>Load/store interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU access protected by VM</td>
</tr>
<tr>
<td>DMA not protected</td>
</tr>
</tbody>
</table>

User Persistent Memory

- Challenge: consistency

Traditional Approach to Durability

- In-memory objects
  
  Serialize

  File or Database

  Deserialize

  • Separate object and persistent formats
  
  • Translation code
  
  • Programmability and performance issues

Persistent Regions in a nutshell

- Address Space
  
  QUEUE

  DRAM

  NVM
Persistent Regions in a nutshell

Virtual memory segments stored in NVM
- Optionally virtualized to a region file on disk

Implementation
- Linux DAX: on `mmap()`, NVM pages mapped to user space

Allocating Persistent Memory

- `pstatic var`
  - Allocates a persistent variable in a static region
- `pmalloc(sz, addr)`
  - Allocates a persistent memory chunk

Avoiding Persistent Memory Leaks

- Programmer shall ensure reachability
- Mnemosyne API helps programmer
  - Atomically sets pointer to newly allocated memory: `pmalloc(sz, addr)`
Avoiding memory corruption

- Techniques:
  - Write protect pages until written
  - Hide NVM pages in memory
  - FSCK-style checks to fix corruption

Central challenge: Consistency

- Data only durable in NVM, not in cache
- Need ordering to maintain consistency.
- How maintain ordering in a CPU cache?
  - Disable caching
  - Extra hardware to remember ordering?
  - Explicitly instructions to force writebacks?

Consistent updates

- Rely on hardware for 1-word atomic update

  - CPU cache may reorder writes to NVM
  - Breaks “crash-consistent” update protocols

Primitive operation: ordering writes

- Why?
  - Ensures ability to commit a change
- How?
  - Flush – MOVNTQ/CLFLUSH
  - Fence – MFENCE
- Inefficiencies:
  - Removes recent data from cache
Consistent Updates

- Support updating data without risking correctness after a failure

- Four update operations

<table>
<thead>
<tr>
<th>Update</th>
<th>API</th>
<th>Usage example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-variable</td>
<td>HW primitive</td>
<td>Set flag</td>
</tr>
<tr>
<td>Append</td>
<td>log_append</td>
<td>Append to journal</td>
</tr>
<tr>
<td>Shadow</td>
<td>pmalloc + single-var</td>
<td>Update tree node</td>
</tr>
<tr>
<td>In-place</td>
<td>patomic {}</td>
<td>Update double linked list</td>
</tr>
</tbody>
</table>

Single-variable Updates

- Example: Set a flag

```cpp
// update data
... node->valid_data = 1;
```

Limitation: Single-word atomicity

Shadow Update

- Example: Update a tree node

```cpp
pmalloc(..., &tmp);
tmp->a = 6;
tmp->b = 7;
flush(&tmp);
fence();
parent->child = tmp;
```
Log-based Journaling

• Log intentions

log_append(NEW_NODE);
log_append(parent);
log_append(old_node);
log_append(new_node);
log_flush(); // make log stable
// update node
log_truncate()

• Inspect log upon recovery and undo incomplete actions

Undo vs Redo Logging

• Write old data durably to log before in-place update

<table>
<thead>
<tr>
<th>Log</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>CLFLUSH</td>
<td>CLFLUSH</td>
</tr>
</tbody>
</table>

• Write new data durably to log before in-place update

<table>
<thead>
<tr>
<th>Log</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>ST</td>
<td>CLFLUSH</td>
</tr>
<tr>
<td>CLFLUSH</td>
<td>CLFLUSH</td>
</tr>
</tbody>
</table>

Programming Persistence Levels

- Transactions:
  patomic { ... }

- Data structures:
  log, heap

- Persistent variables:
  pstatic var

- Persistent regions

- Ordered writes:
  flush, fence

Durable memory transactions

```
patomic {
  B.next = C;
  C.prev = B;
}
```
Durable memory transactions

- **TM Compiler** instruments atomic blocks
  
  ```
  
  patomic {
  B.next = C;
  C.prev = B;
  }
  begin_transaction();
  stm_store(&B.next, C);
  stm_store(&C.prev, B);
  commit_transaction();
  ```

- **TM** runtime supports ACID transactions
  - Log-based recovery after crash

---

### Hash table example

```c
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(sizeof(*bucket), &htRoot);
}

update_hash(key, value) {
  patomic {
    pmalloc(sizeof(*bucket), &bucket);
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```

---

### NV-Heaps

- NV-Heaps programming interface supports
  - Persistent objects as the underlying data model
  - Referential integrity via specialized pointer types
  - Crash-consistent updates via transactions

- NV-Heaps exist as self-contained relocatable memory-mapped files

---

### Smart Pointer Types

- Prevent unsafe references
  - NO non-volatile pointers to volatile data
  - NO inter-heap pointers

- Prevent persistent memory leaks
  - Pointers maintain per-object reference count
  - Objects have to be reachable from the heap’s root

- Support NV-Heaps relocation
  - Hold offset rather than absolute address

---

[Colburn, ASPLOS'11]
NV-Heap API Example

class NVList: public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
}  

NVHeap* nv = NWHOpen("Foo.nvheap");
NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
AtomicBegin {
  ...
  a->get_next();
  a->set_next(...);
}  

Performance Evaluation

• NVM performance model based on DRAM
  FLUSH (addr) → CLFLUSH (addr)
  MODEL_DELAY_NS (150)

• Configurations

  NVM-disk
  Mnemosyne

• Platform: Intel Core 2 2.5GHz (4 cores)

Applications

• TokyoCabinet: Key-value store
  – Original: msyncs B-tree to a mmap’d file
  – Modified: keeps B-tree in persistent memory

• OpenLDAP: Directory service
  – Original: stores dir-entries in Berkeley DB
  – Modified: keeps dir-entries in persistent memory

Application performance

Write-mostly workloads

• TokyoCabinet: 1024-byte ins/del queries
• OpenLDAP: template-based update queries
Hash table performance

Summary

• When is persistent memory good?
  – Fine grained, frequent updates
  – Low latency required
  – Pointer-based data structures

Extra Material

Consistent Durable Data Structures

• Every update creates a new version
• Data representing older versions are never overwritten during an update
  – Modified atomically or copy-on-write.
  – Flushed/fence after all updates written
• Last consistent version
  – Stored in a well-known location
  – Fence before update
  – Used by reader threads and for recovery

[Venkataraman, FAST’11]
Garbage Collection

- CDDS tracks oldest version with non-zero references
- Collect data with older version numbers

CDDS B-Tree Node

- Key
  - [start, end]
- □ Live entry
- □ Deleted entry

B – Size of a B-Tree node

<table>
<thead>
<tr>
<th>5 [1,-)</th>
<th>10 [2,-)</th>
<th>20 [3,-)</th>
<th>30 [4,6)</th>
<th>40 [5,7)</th>
<th>50 [8,-)</th>
<th>60 [9,-)</th>
<th>70 [10,-)</th>
</tr>
</thead>
</table>

B-Tree Insert/Split

- Insert / Split

CDDS B-Tree Lookup

- Find key 20 at version 5
Recovery

- On program restart, programs restart as normal; not from a checkpoint
  - Integrate persistent memory,
  - Replay logs, garbage-collect data structures

Aerie

Why enter the kernel to access file data? Why not map all file data into user address space?

Example: Web proxy

Characteristics
- Flat namespace
- Immutable files
- Infrequent sharing

Example: Web proxy

Web Proxy Cache

POSIX FS

Web Proxy Cache

Proxy FS
Aerie in a nutshell: Components

APP

libFS

LibFS (layout, logic)

User

Kernel

HW

SCM Manager (allocation, protection, addressing)

SCM

Aerie in a nutshell: Operational view

APP

libFS

User

Kernel

HW

SCM

Aerie in a nutshell: Operational view

open

lib -S

User

lib

shared

HW

Aerie in a nutshell: Trusted File Service

APP

libFS

LibFS (layout, logic)

Trusted FS Service (TFS)

APP

libFS

LibFS (layout, logic)

Kernel

SCM Manager (allocation, protection, addressing)

SCM
Aerie in a nutshell: Reducing communication

File system libraries
- **PXFS**: POSIX-style FS
  - POSIX interface
  - Hierarchical DAG namespace
  - Lock per file
- **KVFS**: Key-value FS
  - Put/Get/Erase
  - Flat key-based namespace
  - Global file-system lock
  - Short, immutable files

**Application-workload performance**
- PXFS performs as well or better than kernel-mode FS

Note:
- PXFS performs better than kernel-mode FS by 37%.
- kvfs performs better than ext3 by 18%.
- RamFS performs better than ext3 by 34%.
Micro-benchmarks (Operations)

- PXFS performs as well or better than kernel-mode FS
- KVFS exploits app semantics to improve performance

Application-workload performance

Future directions: SCM-system reliability

- Wide memory interface is unsafe
  - Use hardware protection?
  - Use data structures resilient to faults?
  - Use a managed language?

- HW system failure makes SCM inaccessible
  - Replicate to storage array or remote node?
Open Research Questions

• What to relax in namespace semantics?
  – Example: Plan-9’s remove-on-close

• How limiting is a library interface?
  – Example: sendfile()

• How efficient is virtual-memory hardware?
  – Mapping TB of data increases page-table pressure

Conclusions

• SCM promises low-latency direct-access to storage

• New system interfaces to leverage SCM capabilities
  – Mnemosyne: Persistent memory
    • 35-1400% faster than alternative persistence strategies
  – Aerie: Memory-mapped file systems
    • 33-66% faster than kernel FS via customization