Programming Non-Volatile Memory
Joint Work

Nachshon Cohen, post doc

David Aksun, PhD student
Core memory was persistent
NVM Background
Non-Volatile Memory (NVM)
NVM Characteristics

Low Power

Only consumes power during memory access

Retains data without power

Direct byte-granularity data access

Intel OPTANE DC Persistent Memory
Use Case

In-memory DB
How **Not** to Program NVM

```c
Transfer(Acct from, Acct to, Money amount) {
    from->balance -= amount;
    to->balance += amount;
}
```
Transfer(Acct from, Acct to, Money amount) {
  from->balance -= amount; clflush(&from->balance);
  to->balance += amount; clflush(&to->balance);
  sfence();
}
Programming NVM, version 2

```c
Transfer(Acct from, Acct to, Money amount) {
    atomic {
        from->balance -= amount; clflush(&from->balance);
        to->balance += amount; clflush(&to->balance);
        sfence();
    }
}
```
NVM Usage

OS Process

Transient Memory

Persistent Heap

DRAM

Processor
1. Code accesses NVM with load and store instructions

2. NVM must record a consistent memory state before termination, planned or unexpected

3. Ensure NVM state is consistent in the environment in which execution restarts
When is NVM “Consistent”? 

- Restarted program produces same output as some execution of program without premature termination
- In general, error recovery is difficult if memory state does not satisfy program invariants
- Programs transition between invariant-satisfying states, but states in between are inconsistent (and impossible to recover)
- NVM should not be left in one of these inconsistent states
Similar to Concurrency

Key difference: execution stops and only part of state is preserved. Some parallelization optimizations (such as privatization) can cause errors.
Durable Transactions

Transfer(Acct from, Acct to, Money amount) {
  atomic {
    from->balance -= amount;
    to->balance += amount;
  }  // clflush(&from->balance); clflush(&to->balance); sfence();
}
Software Transactional Memory

**Program**

**NVM Heap**

**Log**

**Undo log** – save contents of overwritten memory locations, so they can be restored if transaction fails

**Redo log** – save updated values, so they can be applied to memory if transaction succeeds
End of background
Outline

• Efficient Logging
• Checkpointing with InCLL
• NVM Recovery
Efficient Logging

NVM, Caches, and Logging

Loc: 0xcfd100
Val: 0x1

Loc: 0xcfd104
Val: 0x1

Loc: 0xcfd108
Val: 0x1

NVM

Head

Unordered Write-Back

Cache

Head
NVM, Caches, and Logging

NVM

Loc: 0xcfd100
Val: 0x1

Loc: 0xcfd104
Val: 0x1

Loc: 0xcfd108
Val: 0x1

Cache

Loc: 0xcfd100
Val: 0x1

Loc: 0xcfd104
Val: 0x1

Loc: 0xcfd108
Val: 0x1

Head

70-200 ns latency x2

Cache Line Flush

James Larus
NVM Consistency Model

• Assumption 1: If store instruction $S_1$ becomes visible to other threads before instruction $S_2$, then the value written by $S_1$ reaches the cache before the value written by $S_2$

• Assumption 2: A cache line is transferred from the cache to the NVM atomically

• **Persistent ordering**
  
  \[
  \begin{align*}
  S_1 &<_{hb} cflush(a(S_1)) <_{hb} sflush(a(S_2)) <_{hb} S_2 \\
  S_1 &<_{hb} S_2
  \end{align*}
  \]
  (explicit flush)

  \[
  \begin{align*}
  S_1 &<_{hb} S_2 \land c(S_1) = c(S_2) \\
  S_1 &<_{hb} S_2
  \end{align*}
  \]
  (granularity)
Validity Bit

Assume: log entry fits in cache line (64 bytes) and at least one bit (of 512) is unused.

Log entry is valid if bit is set (unset)

Requires only 1 cache flush
Where Does Validity Bit Fit?

• Almost always can find validity bit
  • x64 address have 15 unused bit at top and typically 2-3 unused bits at bottom

• Randomization
  • Initialize log memory with 64-bit random value
  • Write log entry in order
  • If last word is not random value, then entry is valid (with high probability)

• Flexible validity bit
  • Find first bit different between old cache line and new value
  • If bit exists, use it as validity bit (otherwise, doesn’t matter)
  • Store bit position in external table

• More details and examples in paper
Performance

Validity Bit: **STPS** (Single-Trip Persistent Set)
Two cache flushes: **TwoRounds**

YCSB write heavy benchmark (50% writes)
Outline

• Efficient Logging
• Checkpointing with InCLL
• NVM Recovery
Checkpointing used in high-performance computing (HPC)

Problems:
- Long pause while copying heap
- Recovery time proportional to checkpoint interval
Fine-Grain Checkpointing of Masstree

• Masstree is cache-efficient combination of trie and B+ tree
  • Mao, Kohler, Morris. *Cache craftiness for fast multicore key-value storage.* EuroSys ’12, 2012
• Used in Silo in-memory DB, key-value stores, etc.

• Make Masstree persistent by storing data structure in NVM

Epochs

- Checkpoint with 64 ms epoch
  - Masstree uses this interval for allocating/reclaiming nodes
- Execute `wbincvd` instruction to flush entire cache (to NVM)
  - 430 – 550μs (< 1%)
- Failure during epoch causes execution to restart after previous epoch

- Need to undo changes written during a failed epoch
  - In cache-line log (InCLL)
  - Separate undo log for complicated, infrequent cases

Masstree Leaf Node

14 keys, values
(1 fewer than standard)
Insert (With Empty Value Slot)

If no empty slot, then split node
Use undo log for node splitting
Delete
Update

Encode index of modified value into unused bits in Val pointer.
Sequences of Operations

Epoch in which permutation was checkpointed
- Only copy permutation when Epoch differs (once per epoch)
- Allows lazy restoration

Mixed sequences of insert delete require redo logging because InCLL can only hold one value

Also encode epoch in unused bits
Performance

Durable Data Structures

Million operations per second

- Masstree
- DurableMasstree
- Log-Free

0 2 4 6 8 10 12

YCSB: a_uni, b_uni, c_uni, e_uni, a_zipf, b_zipf, c_zipf, e_zipf

11% 6.3% 5.7% 0.5% 8% 4.4% 5.9% 0%
Performance (Added NVM Latency)

InCLL overhead vs. added NVM latency

Latency added to write back + fence (nano-seconds)

Durability overhead (higher is slower)

Uniform
Zipfian

James Larus
Outline

• Efficient Logging
• Checkpointing with InCLL
• NVM Recovery
NVM Recovery

Persistent objects point to methods
• Code may be loaded at different address because of ASLR, debugger, profiler, code changes, ...

Persistent store may be mapped to different virtual address

Persistent data must be consistent in recovered process, as well as being consistent when original process fails

Potential Inconsistencies

- Transient objects pointed to from NVM
  - TCP sockets, locks, thread IDs, ...
- Pointers between persistent objects if NVM mapped to different location
- Pointers to code and read-only data if text segment mapped differently
Previous Solutions

• Ignore problem
  • If NVM maps to different location, quit...
    • Oops, there goes your data!
  • If text maps differently, continue...
    • Oops, there goes your data!
  • If you use an old lock, fail...
    • Oops, there goes your data!

• Use offsets between persistent objects instead of addresses
  • Memory access becomes more expensive
  • Extensive code changes
NVMReconstruction C++ Extension

```cpp
struct kp_vt_struct {
    kp_kvstore *parent; // back-pointer to parent kvstore
    transient pthread_mutex_t *lock; // lock for this version table
    ...

    reconstructor(kp_vt_struct* o) {
        assert(kp_mutex_create("(*new_vt)->lock", &(o->lock)) == 0);
    }
}

void main() {
    kp_vt_struct *new_vt = pnew kp_vt_struct;
    ...
    pdelete new_vt;
}
```
Reconstructing an Object

- Every persistent object has a type header
  - LLVM extension
- Zero transient fields
- Relocate pointers
  - Inter NVM
  - To code and read-only data
- Run reconstructor function

Similar to relocation in garbage collector, except...
Failure During Relocation

### Initial Execution

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr: 1000</th>
<th>Points to</th>
<th>Base Addr: 1100</th>
<th>Points to</th>
<th>Base Addr: 1200</th>
<th>Points to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
</tbody>
</table>

### Execution #2

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr: 1000</th>
<th>Points to</th>
<th>Base Addr: 1100</th>
<th>Points to</th>
<th>Base Addr: 1200</th>
<th>Points to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
</tbody>
</table>

### Execution #3

<table>
<thead>
<tr>
<th>Addr</th>
<th>Base Addr: 1000</th>
<th>Points to</th>
<th>Base Addr: 1100</th>
<th>Points to</th>
<th>Base Addr: 1200</th>
<th>Points to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td></td>
<td>1100</td>
<td></td>
<td>1200</td>
<td></td>
</tr>
</tbody>
</table>

### Reconstruction Info

<table>
<thead>
<tr>
<th>Addr of A</th>
<th>Δ</th>
<th>Page exec. #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1330</td>
<td>+200</td>
<td>1</td>
</tr>
<tr>
<td>1430</td>
<td>+100</td>
<td>2</td>
</tr>
<tr>
<td>1530</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Lazy Reconstruction
• Use VM page protection to detect first access to object
• Reconstruct all objects on page
Performance Overhead

Key-value store implemented in Atlas. 1GB NVM heap. YCSB write-heavy workload (50% writes).
Conclusion

- NVM is persistent, directly accessible main memory
  - Well suited for very large in-memory data structures (DB, graphs, etc.)

- Programs must be aware of “NVMness” to maintain consistency

- Caches in existing memory systems make consistency expensive
  - Key insight: memory consistently transfers entire cache line to NVM

- How much can we pack into a cache line?
  - More than you think

- But, don’t forget recovery!