Persistent Memory Ordering

Michael Swift

Includes slides from Pelley [ISCA’14], Condit [SOSP’09], Volos [ASPLOS’11]
Volatile Memory Ordering

- Caches reorder writes between the CPU and memory.
- For DRAM, cache coherence protocols and memory barriers ensure that all CPUs have a consistent global view of the state of memory.

- It does not matter when or in what order data is actually written back to DRAM.
Why Persistence Changes Things

• With NVM in place of DRAM, though, the order in which write-backs occur is now important.

• Why?
  – May write commit record of a transaction before data is persistent
  – May write partial data

• Note: persistent memory ordering is
  – Not visible to executing code
  – Only visible following crash
CPU cache may reorder writes to NVM

- Breaks “crash-consistent” update protocols

```
STORE value = 0xC02
STORE valid = 1
```

```
<table>
<thead>
<tr>
<th>NVM</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xDEADBEEF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write-back Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0xC02</td>
</tr>
</tbody>
</table>
Simple solutions

- Disable caching
- Write through caching
- Flush entire cache at commit
Outline

• Mnemosyne
• BPFS / epochs
• Intel’s new instructions
Mnemosyne Ordering

- Primitives
- Transactions
Primitive operation: ordering writes

- **Why?**
  - Ensures ability to **commit** a change

- **How?**
  - Flush – **MOVNTQ/CLFLUSH**
  - Fence – **MFENCE**

```
STORE value = 0xC02
FLUSH (&value)
FENCE
STORE valid = 1
```
Durable memory transactions

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

• Compiler instruments atomic blocks

```c
patomic {
    B.next = C;
    C.prev = B;
}
```

• Runtime supports ACID transactions
  – Write-ahead redo logging
  – Log-based recovery after crash

```c
begin_transaction();
stm_store(&B.next, C);
stm_store(&C.prev, B);
commit_transaction();
```
Mnemosyne Ordering

- Single transaction per thread
- Data made persistent on commit by flushing log
- Data lazily evicted from cache with CLFUSH
  - In background by log cleaner (for low-load/low-latency systems)
  - Synchronously for high-load systems
- Synchronous commit with global transaction ID strictly orders persistent writes
BPFS Epochs Barriers

• An epoch is a **sequence of writes to persistent memory from the same thread**, delimited by a new form of memory barrier issued by software.

• An epoch that contains dirty data that is not yet reflected to BPRAM is an **in-flight epoch**;
  – an in-flight epoch commits when all of the dirty data written during that epoch is successfully written back to persistent storage.

• The key invariant is that when **a write is issued to persistent storage**, **all writes from all previous epochs must have already been committed to the persistent storage**, including any data cached in volatile buffers on the memory chips themselves.
Detailed Considerations

• Per-processor **epoch ID** tags writes
• Each cache line stores epoch ID at time of modification
  – Cache tracks **oldest in-flight epoch** for each processor
• On write-back to NVM, cache ensures line is from oldest epoch **or** evicts all earlier epochs
• Writes to tagged line flush that epoch (and older ones)
• Read/write data tagged by a **different** processor flushes **all** old epochs
  – Ensures read/write dependencies enforced across cores
Problem 1: Ordering

A problem has been detected and Windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.
Problem 2: Atomicity

... CoW Commit ...

L1 / L2

BPRAM
Enforcing Ordering and Atomicity

• Ordering
  – Solution: Epoch barriers to declare constraints
  – Faster than write-through
  – Important hardware primitive (cf. SCSI TCQ)

• Atomicity
  – Solution: Capacitor on DIMM
  – Simple and cheap!
Ordering and Atomicity

- CoW
- Barrier
- Commit

Ineligible for eviction!

MP works too (see paper)
BPFS Epoch Model

• Per-thread transactions
  – Ordering between transactions within a thread guaranteed

• Between threads
  – Must access **persistent** data from previous epoch to enforce ordering

• No durability guarantee
  – Data eventually ages out of cache
Intel’s HW Support

• CLFLUSHOPT
  – is defined to provide efficient cache flushing; unordered version of CLFLUSH

• CLWB
  – writes back modified data of a cacheline similar to CLFLUSHOPT, but avoids invalidating the line from the cache (and instead transitions the line to non-modified state).

• PCOMMIT
  – defined to commit write data queued in the memory subsystem to persistent memory.

• These operations are not ordered with respect to each other; need SFENCEs
Generalizing Persistence Order

• *Memory Persistency*
  – Work by Steven Pelley, Peter M. Chen Thomas F. Wenisch at Umich

• Recovery Observer
  – Sees writes to NVM, not caches

• Persist ordering
  – Order of writes to NVM, visible to observer
  – May be different than volatile ordering
Strand persistency example

A  B  C

::

A  B  C

::

A  Barrier  B  C

::

B must be ordered with A and/or C

Epoch

A  A

B  B

C  C

Strand

NewStrand

A

Barrier

A

Barrier

C

NewStrand

B

Strands remove unnecessary ordering constraints
A Few Open Questions

• What hardware mechanisms are needed for enforcing ordering?

• How can stores be ordered across cores?
  – Can a persistence operation span cores?
  – Should persistence ordering follow volatile memory ordering?
  – Example:
    • Write data from multiple cores simultaneously
    • Commit only when all data persistent

• How much ordering is necessary?
  – Do we need arbitrary dependence graphs?
Other issues

• What granularity of atomic writes is needed?
  – 64 bits or cache line?

• What is the programmer interface?
  – Library (e.g. K/V or object store)
  – Load/store?
  – Transactions?