Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory

Virendra J. Marathe, Margo Seltzer, Steve Byan, Tim Harris
Penumbra Research Group

Oracle Labs
Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
Outline

1. Persistent Memory and the System Stack
2. Making Memcached Persistent: lessons learned
3. Takeaways: this is *harder* than you think!
Persistent Memory (pmem): What is it?

• Looks like DRAM
  – Byte addressable
  – DIMMs across the memory bus
  – Cacheable
  – Low latency: 1000X lower than NAND flash (3D XPoint), projected to approach and eventually eclipse DRAM performance (e.g. STT-MRAM, Carbon-NanoTubes, etc.)

• Acts like Storage: State persists across restart events

• Dense: projections better than DRAM (~10X, 3D Xpoint)

• Cost expected to be between DRAM and NAND flash
Integrating Persistent Memory in the Software Stack

**Diagram Description:**

- **Application**
  - Standard File API

- **File System**
  - NVDIMM Driver

- **Kernel Space**
  - PM Aware File System
  - MMU Mappings

- **User Space**
  - Standard File API
  - `mmap()`

**Persistent Memory DIMM**
Use cases for persistent memory

• Faster block storage device
  – No changes to most of the software stack; gains are limited

• Bigger, slower volatile memory
  – New tier in the memory hierarchy

• Leverage byte addressability and fast persistence (most disruptive change; our focus)
  – Make persistence fast via byte addressable data representation and access
  – Make application restart/warmup fast
Use cases for persistent memory

• Faster block storage device
  – No changes to most of the software stack; gains are limited

• Bigger, slower volatile memory
  – New tier in the memory hierarchy

• Leverage byte addressability and fast persistence (most disruptive change; our focus)
  – Make persistence fast via byte addressable data representation and access
  – Make application restart/warmup fast

Memcached fits here
Use cases for persistent memory

• Faster block storage device
  – No changes to most of the software stack; gains are limited

• Bigger, slower volatile memory
  – New tier in the memory hierarchy

• Leverage byte addressability and fast persistence *(most disruptive change; our focus)*
  – Make persistence fast via byte addressable data representation and access
  – Make application restart/warmup fast

Memcached fits here...... large caches, warmup can take hours
Outline

1. Persistent Memory and the System Stack
2. Making Memcached Persistent: Lessons Learned
   - Early decision: Don’t change Memcached’s high level architecture
3. Takeaways: this is harder than you think!
Lessons Learned (on memcached v1.4.25)
This is harder than you might think!

1. Persistence of data structures can be contagious
2. Failure-atomic transactions can be crucial
3. Persistent and nonpersistent objects interact in unexpected ways
4. Critical sections can pose significant problems
...
...
...
(10 lessons in the paper)
Lesson I: Mods can be contagious
Lesson I: Mods can be contagious

Central Hash Table

LRU Cache
Lesson I: Mods can be contagious

Central Hash Table

LRU Cache

Slab Allocator
Lesson I: Mods can be contagious

Central Hash Table

Client Request Mgmt State Machine

LRU Cache

Slab Allocator
Lesson I: Mods can be contagious
Lesson I: Mods can be contagious
Lesson I: Mods can be contagious

• Several (tightly coupled) modules
  – Central hash table (can grow, doesn’t shrink)
  – LRU cache
  – Client session management state machine
  – Own allocator
  – Helper threads: LRU crawler, hash table resizer, allocator slab manager

• Data structures intertwined

• Needed to make changes all over the application
Lesson II: Failure-Atomic Transactions can be crucial

- Complex code paths; e.g. Put(K,V)
  - Change hash table
  - Allocate new <K,V> pair; touches slab allocator
  - Update LRU cache
Lesson II: Failure-Atomic Transactions can be crucial

• Complex code paths; e.g. Put(K,V)
  – Change hash table
  – Allocate new <K,V> pair; touches slab allocator
  – Update LRU cache
Lesson II: Failure-Atomic Transactions can be crucial

• Complex code paths; e.g. Put(K,V)
  – Change hash table
  – Allocate new <K,V> pair; touches slab allocator
  – Update LRU cache
Lesson II: Failure-Atomic Transactions can be crucial

• Complex code paths; e.g. Put(K,V)
  – Change hash table
  – Allocate new <K,V> pair; touches slab allocator
  – Update LRU cache

*All updates need to be mutually consistent across failures*
Lesson II: Failure-Atomic Transactions can be crucial

- Complex code paths; e.g. Put(K,V)
  - Change hash table
  - Allocate new <K,V> pair; touches slab allocator
  - Update LRU cache

*Failure-Atomic Transaction for Put(K,V)*
Lesson II: Failure-Atomic Transactions can be crucial

• Ended up using in-house transactional runtime
• Original 15K LOC, we ended up adding 7K LOC
• Most of the added LOCs come from hand instrumenting transactional loads/stores to persistent memory
• *Language support will help a lot!*
Lesson III: Persistent and nonpersistent objects interact in unexpected ways
Lesson III: Persistent and nonpersistent objects interact in unexpected ways
Lesson III: Persistent and nonpersistent objects interact in unexpected ways
Lesson III: Persistent and nonpersistent objects interact in unexpected ways

• Persistent and nonpersistent objects can have dependencies

• In Memcached
  – References from nonpersistent client-session objects to key-value pairs tracked using reference counts
  – Deleted objects with non-zero reference counts could be lost on failure
  – Solution: Add a persistent “pending reclamations” list, and use it during recovery to avoid memory leaks.
Lesson IV: Critical Section Headaches

• Locking: Let the application use its locks within transactions
  – But lock release should be deferred to transaction completion (the runtime implicitly releases the acquired locks after the transaction completes)
  – Can inflate critical sections by a lot – introduces deadlocks, scalability problems
  – Each LRU list has a lock
  – Memory allocation/free: memcached has its own allocator which uses its own (single) lock

• Solutions: Make locks reentrant; postpone execution of some critical sections to end of transaction (e.g. hash table size update); approximate LRU

• LRU cache, allocator lock still scalability problems
Memcached evaluation: YCSB workload generator
Experiments conducted on Intel’s Software Emulation Platform for Persistent Memory

Emulator Configuration
- Assume CLWB
- Latency: 300 nanosecs
- Pbarrier: 100 nanosecs
Outline

1. Persistent Memory and the System Stack
2. Making Memcached Persistent: lessons learned
3. Takeaways: this is harder than you think!
Takeaways
This can be *hard*, and the returns may not be worth it!

• **Fundamental issues**
  – Main decision to not change memcached’s architecture was problematic
  – Modules too tightly coupled, leading to extraneous complexity and overheads

• **So**
  – Either may need to radically rehash the architecture of such systems
  – Or, just build one from scratch
  – Or, trade instantaneous warmup for simplicity and better common case performance

• **New programming abstractions need to be explored/researched**

• **Story may be very different for other applications**
Integrated Cloud
Applications & Platform Services
Lesson I: Mods can be contagious

`Tightly Coupled Subsystems`

- Central Hash Table
- LRU Cache
- Slab Allocator
- Client Request Mgmt
- State Machine
- Background Maintenance Threads