Killer Applications for Mapped NVM

Garret Swart
Why Map your NVM?

- Storage APIs on NVM are great
  - Provide faster implementations of familiar APIs
- But IO controllers often:
  - Have device driver and communication overheads
  - Don’t exploit processor cache and compute resources
- Mapping NVM into application memory has benefits:
  - Instruction level latency: LOAD/STORE
  - Programmer has control
- What are the killer apps for mapped NVM? How do they benefit?
- What issues do we need to solve to go production?
Outline

• Killer Applications for NVM
  1. Write Ahead Logs
  2. Persistent data caches
  3. In-memory databases

• Issues to address:
  1. HA and Durability
  2. Failure detection and reporting
  3. Avoiding Data corruption
Killer App 1: Write Ahead Logs

• What is it?
  • Atomic and durable: Record changes to many data items in an atomic log record written to stable storage
  • Low latency: Lazily update data items after log write
  • Cost: Must reapply log records after a failure

• Why Store in NVM?
  • Lower Latency Log Write ➔
    • Higher transaction rates with less parallelism
    • Locks held for a shorter time ➔ Reduced contention
  • Higher bandwidth ➔
    • Higher transaction rates with fewer storage devices
Killer App 2: Persistent Data Caches

• What is it?
  • In memory cache of:
    • Clean data items read from primary store
    • Dirty data items updated in a log record and waiting to be written to primary store
  • Cost: Must refill cache after a failure: Recovery is complete only when server is ready to operate at speed

• Why Store in NVM?
  • Reduce recovery time after a failure
    • Don’t need to read data from primary store
    • Don’t need to reapply many log records to recover dirty data items
  • Replication not needed for durability, only for HA
Killer App 3: In Memory Databases

- What is it?
  - Database with an in-memory primary data store
  - Store data items in processor optimized data formats
  - More efficient “pointer-like” data item references
  - Still needs write ahead log and checkpoint files for recovery

- Why store in NVM?
  - Reduce or eliminate checkpoints
  - Still need write-ahead log to handle:
    - Atomicity: Instruction atomicity too small for applications
    - Durability: In-flight NVM writes may not be durable
Issue 1: HA and Durability

- No dual ported DIMMs
  - Each DIMM is tied to a single failure domain
    - Lose the processor → Lose the DIMM
    - Replicate data to a processor in another failure domain
    - Deal with the increased write latency

- Replication abstraction level
  - STORE instruction too small a unit for atomic consistency
  - Replicate at the application level: “Transactionl”

- Need fast application level communication:
  - RDMA into remote NVM: Good for bulk data movement
  - Instruction latency messaging: Good for control
  - Fast communication mitigates cost of replication
Issue 2: Failure Detection and Reporting

- Everything fails sometime, want to limit scope of failure and allow NVM applications to be either:
  - More resilient with the complexity and loss of performance that using msync() entail
  - Simple and fast with the loss of resilience this may entail
- Flush-on-Shutdown:
  - Write all completed processor stores to NVM at shutdown
    - But the flush might fail unexpectedly
  - Check for FOS failure at next open:
    - The file might have lost an update to any location
  - NVM File Systems should not rely on FOS for meta-data:
    - Use explicit msync(), more expensive but more resilient
Issue 2: Failure Detection and Reporting

- **NVM DIMM write failure**
  - Transparent if write can be remapped within DIMM
  - Otherwise file system should mounted readonly so volume and file system can be rebuilt
  - Stall application during rebuild?
    - Nope, that complicates availability: Better to send signal on next write: It can die or handle as it chooses

- **NVM DIMM read failure**
  - Transparent if data can be reconstructed using ECC and remapped within DIMM
  - Otherwise unmap (cache line? VM page? file?) and signal
  - File system should be resilient to read failures on its key data structures by using internal replication
Issue 3: Avoiding Data corruption caused by wild writes

- Memory mapped files are more susceptible to wild writes than explicit read/write operations
- Avoid by:
  - Using managed runtimes like: Java, .Net, JavaScript
  - Persistent Data Structures that preserve previous versions of themselves
    - Only write into newly allocated objects
  - Constructing sparse address spaces and delaying reuse of old addresses:
    - Old heap pointers and random values less likely to be valid addresses
Issue 3: Avoiding Data corruption through logical replication

• Less susceptible to replicated bugs than physical replication

• Different replicas can run different implementations of the same service
  • Or different versions of the same system

• Bugs in mature systems tend to be complex and state related rather than request related
  • Even replicas running the same software are likely to have different state because of different low level operation serializations
  • Running the same request on two replicas can have different results ➔ Less chance of replicated bugs
Issue 3: Avoiding Data corruption through multi-level logging

- Logging at the request level is needed for application semantics but implementation logging can improve resiliency.
- A request level operation may be made up of many internal operations that allocate and release internal resources.
- Write ahead logging at an internal resource level
  - Allows the internal state of the resource to be recovered
  - Can use a smaller ordering scope → Less synchronization
  - May not need to be retained as long or replicated as widely
Conclusions

• As new technology arrives, new kinds of applications become possible

• BUT …
  • New performance bottlenecks emerge
  • Failure probabilities and consequences change

• And as engineers we respond with new implementation techniques
  • In HW, in OS, in Libraries, in Applications

• Ensuring full employment