Memory Persistency

Thomas F. Wenisch
University of Michigan


Dali, Salvador. The Persistence of Memory. 1931.
Promise of persistent memory (PM)

Performance + Density + Non-volatility

Byte-addressable, load-store interface to storage
Recovery can inspect the data-structures in PM to restore system to a consistent state.
Recovery needs ordered PM updates

Task: Fill node and add to linked list, safely

Recoverable

- fillNewNode()
- updateTailPtr()

In-memory data
Recovery needs ordered PM updates

Task: Fill node and add to linked list, safely

In-memory data

No ordering → No recovery
Memory persistency models

- Programmers can express write order to PM
- Hardware enforces write order to PM
- Similar to how memory consistency models ensure store visibility in multi-core systems
- Models needed at the ISA level and language level
  - Compiler & runtime map from language to ISA model
Our work on memory persistency

- High-Level Languages (HLL)
  - Language-Level Persistency [ISCA 2017]
    [SIGARCH/TCCA Outstanding Dissertation Award]
- Libraries & Runtime
  - High-Performance Transactions for Persistent Memories [ASPLOS 2014]
  - Persistency for Synchronization-Free Regions [PLDI 2018]
- Compiler
  - Memory Persistency [ISCA 2014]
    [IEEE MICRO Top Picks]
- Architecture (ISA)
- Microarchitecture
  - Delegated Persist Ordering [MICRO 2016] [Best Paper Nominee]
Terminology

• **Persist**
  – Act of making a store durable in PM

• **Persistent memory order (PMO)**
  – Memory events ordered by *persistency model*
  – Governs the *order in which stores persist*
Persistency model guarantees

• Ordering
  – How can programmers order persists?

• Failure-atomicity granularity
  – Which group of stores will persist atomically?
Ordering guarantee design space

Happens before:

- Volatile memory order
- Persistent memory order

Strict persistency: single memory order

Relaxed persistency: separate volatile and (new) persistent memory orders
Example ISA-level relaxed persistency model: Epoch persistency [Condit ‘09] [Pelley ‘14] [Joshi ‘15] ...

- FENCEs break **thread execution into epochs**
- Persists across epochs are ordered
  - No ordering within epoch
Failure-atomicity design space

Granularities of failure-atomicity

Individual persists

L1.lock();
L2.lock();
x = 100;
y = 100;
a = 100;
b = 100;
L2.unlock();
L1.unlock();

Sync. free regions

L1.lock();
L2.lock();
x = 100;
y = 100;
a = 100;
b = 100;
L2.unlock();
L1.unlock();

Outer critical sections

L1.lock();
L2.lock();
L2.unlock();
a = 100;
b = 100;
L2.unlock();
L1.unlock();

Programmability

Easier

Implementation

Costlier
Our recent language-level persistency work: Failure-atomic SFRs [PLDI'18]

- Persist SFRs in sequentially consistent order
- Allow hardware/compiler optimizations within SFR
- **Inter-thread** ordering
  - Synchronizing *acquire* and *release* ops.
- Failure-atomicity
  - Compiler-orchestrated *undo*-logging

Extends SC-for-DRF guarantee to post-failure recovery
Undo-logging for SFRs

L1.acq();

\( x = 100; \)

L1.rel();

Need to ensure that undo logs persist and commit in the order of execution of SFRs
Design 1: Coupled-SFR

Thread 1
L1.acq();
SFR1
x -= 100;
L1.rel();

Thread 2
L1.acq();

SFR2
x -= 200;
L1.rel();

+ Persistent state lags execution by at most one SFR
→ Simpler implementation, latest state at failure

- Need to flush updates at end the of each SFR
→ Performance cost
Design 2: Decoupled-SFR

• Coupled-SFR has simple design, but lower perf.
  – Persists and log commits on critical execution path 😞

• Key idea: Decouple persistent state from program exec.
  – Persist updates and commit logs in background
  – Create undo logs in order
  – Roll back updates in reverse order of creation on failure
Decoupled-SFR in action

Thread 1
L1.acq();
SFR1
L1.rel();

Thread 2
SFR2
x = 100;
L1.acq();
L1.rel();

Create logs in order during execution

Flush and commit performed in background

P1
C1

P2
C2

Thread 1
L1
M1
REL1

Thread 2
ACQ2
L2
M2

SFR1
SFR2
Performance evaluation

Atlas  Coupled-SFR  Decoupled-SFR  No-persistency

Better

Normalized exec. time

Failure-atomic SFRs outperform atomicity at outer-most critical sections by 66%
Summary

- **Memory persistency models** provide guarantees required for recoverable software on persistent memories
  - Like consistency models, needed at ISA and *language* level

- **Two key dimensions:**
  - Ordering
  - Failure atomicity granularity

- **Failure-atomic synchronization-free regions**
  - Persistent state moves from one sync. operation to the next
  - Extends clean SC semantics to post-failure recovery