Persistency Programming 101
Why and What of memory persistency

Aasheesh Kolli*  Steven Pelley†  Ali Saidi$
Peter M. Chen*  Thomas F. Wenisch*$

* University of Michigan
† Snowflake Computing
$ARM
Executive summary

• NVMs → in-memory, recoverable data structs
• Require ordering of NVM writes (persists)
• Consistency models do not order persists
• Memory persistency [ISCA ‘14]
  – Consistency models for NVM
• This talk:
  – Motivate memory persistency
  – Summarize persistency models
Background - Memory consistency

- Contract b/w hardware and software on store visibility (implementation independent)
- May be strict (SC) or relaxed (RMO)
- Does not apply to persists

Need persistency models to order persists
Future system abstraction

Memory events (stores/loads)

Core0  Core1

1  2  3  4  5  6

Volatile memory order (consistency model)

Persist memory order (persistency model)

No Failures

Failures

Recovery
Future system abstraction

Memory events (stores/loads)

Core0  Core1

1 2

Core0  Core1

1 2

Persistency model governs store visibility in the presence of failures

Volatile memory order (consistency model)

Persist memory order (persistency model)

No Failures

Failures

Persistency model governs store visibility in the presence of failures
Types of persistency models

Memory events (stores/loads)

Consistency

Strict persistency

Relaxed persistency
Strict vs relaxed persistency

• Strict persistency
  – Consistency model = persistency model
  – Expensive (cost of persist > cost of store)

• Relaxed persistency
  – Separate consistency and persistency models
  – Reduces persist memory order constraints
  – Might require more robust recovery software
  – Failures are infrequent $\rightarrow$ recovery overhead OK

Relaxed persistency models $\rightarrow$ ↑ performance
How do persists affect performance?

- Persist forms a directed acyclic graph (DAG)
  - Critical path limits overall performance

1: persist Qe.data[0] = x
2: persist Qe.data[1] = y
3: persist Qe.valid = true

Models should allow expression of minimal constraints
Strict persistency

• Consistency model = persistency model
• Relaxed consistency $\rightarrow$ persist concurrency
Strict persistency with SC

1. lock (volatile mutex)
2. persist Qe.data[0] = x
3. persist Qe.data[1] = y
4. persist Qe.valid = true
5. unlock

- Program order $\rightarrow$ store order $\rightarrow$ persist order
- No annotations required
- Suboptimal persist critical paths
Strict persistency with RMO

1. lock (volatile mutex)
2. barrier
3. persist Qe.data[0] = x
4. persist Qe.data[1] = y
5. barrier
6. persist Qe.valid = true
7. barrier
8. unlock

- Barriers for **synchronization** and recovery
- Annotation burden on the programmer
- Affects volatile perf
Strict persistency with RMO

1. lock (volatile mutex)
2. barrier
3. persist Qe.data[0] = x
4. persist Qe.data[1] = y
5. barrier
6. persist Qe.valid = true
7. barrier
8. unlock

- Barriers for synchronization and recovery
- Annotation burden on the programmer
- Affects volatile perf
Relaxed persistence models

- Decouple consistency and persistency models
- Expose additional persist concurrency
  - Important for conservative consistency models
- Will use SC as underlying consistency model
**Epoch persistency**

1. lock (volatile mutex)
2. persist Qe.data[0] = x
3. persist Qe.data[1] = y
4. **persist barrier**
5. persist Qe.valid = true
6. unlock

- **Persist barriers** divide program into epochs
  - Inspired from BPFS [SOSP ‘09]
  - Persists within epoch concurrent
  - Persists from successive epochs ordered
  - Store visibility still dictated by program order
Epoch persistency drawback

**Ideal**
- persist A
- persist B
- persist C

**DAG-1**
- persist A
- persist barrier
- persist B
- persist C

**DAG-2**
- persist A
- persist C
- persist barrier
- persist B
- persist C
- persist A
- persist barrier
- persist B

Cannot express minimal constraints
Strand persistency

• Divide thread’s execution into strands
  – A strand can be abstracted as a logical thread
• Persists on different strands are concurrent
• New memory event `newStrand`
  – Persist barriers order persists within a strand

Can enforce only the necessary constraints
Ordering persists across threads

- Conflicting accesses establish persist order
  - Two accesses to same address, at least one store
  - Persist order must match volatile order
  - Could be to volatile/non-volatile addresses
  - **Strong persist atomicity**
Strong persist atomicity example

Thread - 1
- lock X (volatile mutex)
- persist barrier
- persist A
- persist barrier
- unlock X

Thread - 2
- lock X
- persist barrier
- persist B
- persist barrier
- unlock X

Persist order

Conflicting accesses

Time
Conclusions

• Strict persistency
  – Unifies consistency and persistency model

• Epoch persistency
  – Persists within epoch concurrent, epochs ordered

• Strand persistency
  – Allows enforcing only minimal constraints

• Strong persist atomicity
  – Allows ordering persists across threads/strands
Thank You!

Questions?