Efficient Durability for Lock-based Code

Swapnil Haria, Pratyush Mahapatra, Mark D. Hill, Michael M. Swift
University of Wisconsin-Madison
Motivation

Most proposed persistent memory (PM) systems assume transactional memory (TM)

Most parallel and concurrent programs use locks

Let’s make PM systems support locks WELL by building on
• HP Atlas
• Wisconsin HOPS
Outline

Making lock-based code failure-atomic

Problem 1: Synchronous Flushing
   Solution: Unmodified HOPS

Problem 2: Synchronization History
   Solution: HOPS+

Evaluation
Outline

Making lock-based code failure-atomic

Problem 1: Synchronous Flushing  
Solution: Unmodified HOPS

Problem 2: Synchronization History  
Solution: HOPS+

Evaluation
Lock-based code and PM

Lock L
- S1: DATA = 1
- FLUSH S1
- S2: FLAG = 1
- FLUSH S2
- Unlock L

VOLATILE CACHE
- DATA = 0
- FLAG = 0

PERSISTENT MEMORY
- DATA = 0
- FLAG = 0
Lock-based code and PM

Lock L
S1: DATA = 1
FLUSH S1
S2: FLAG = 1
FLUSH S2
Unlock L

CRASH!
Lock-based code and PM

Lock L
S1: DATA = 1
FLUSH S1
S2: FLAG = 1
FLUSH S2
Unlock L
Failure Atomic Lock-based Code

Log all lock-acquire/release operations

Log all PM stores

Recovery: undo incomplete critical sections

Thread 1
Lock L
(Log-ACQ L); FLUSH
Log (DATA); FLUSH
S1: DATA = 1;
FLUSH S1
Log (FLAG); FLUSH
S2: FLAG = 1
FLUSH S2
(Log-Rel L); FLUSH
Unlock L
Problem 1: Synchronous Flushing

Thread 1
Lock L
(Log-ACQ L); FLUSH
Log (DATA); FLUSH
S1: DATA = 1;
FLUSH S1
Log (FLAG); FLUSH
S2: FLAG = 1
FLUSH S2
(Log-Rel L); FLUSH
Unlock L
Remember HOPS?

CPU

Private L1

Persist Buffer Front End

Shared LLC

Persist Buffer Rear End

Persist Buffer Back End

Volatile

Persistent

Loads + Stores

Private L1

Loads + Stores

DRAM Controller

PM Controller
OFENCE: Ordering Fence

• Orders stores preceding OFENCE before later stores
DFENCE: Durability Fence

• Makes the stores preceding DFENCE durable
# Eliminating synchronous flushes

Fast OFENCEs replace slow flushes

Only one DFENCE at unlock

HOPS works well with durable lock-based code!

<table>
<thead>
<tr>
<th>Original</th>
<th>ATLAS (HOPS)</th>
<th>ATLAS (HOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock L</strong></td>
<td>Lock L</td>
<td>Lock L</td>
</tr>
<tr>
<td></td>
<td>Log-Acq(L)</td>
<td>Log-Acq(L)</td>
</tr>
<tr>
<td></td>
<td>CLFLUSH Log</td>
<td>OFENCE</td>
</tr>
<tr>
<td><strong>Unlock L</strong></td>
<td>SFENCE</td>
<td>OFENCE</td>
</tr>
<tr>
<td></td>
<td>Log-Rel(L)</td>
<td>Log-Rel(L)</td>
</tr>
<tr>
<td></td>
<td>CLFLUSH Log</td>
<td>DFENCE</td>
</tr>
<tr>
<td></td>
<td>Unlock L</td>
<td>Unlock L</td>
</tr>
<tr>
<td><strong>ST A, val</strong></td>
<td>Undo-Log(A)</td>
<td>Undo-Log(A)</td>
</tr>
<tr>
<td></td>
<td>CLFLUSH Log</td>
<td>OFENCE</td>
</tr>
<tr>
<td></td>
<td>ST A, val</td>
<td>ST A, val</td>
</tr>
<tr>
<td></td>
<td>CLWB A</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Making lock-based code failure-atomic

Problem 1: Synchronous Flushing
   Solution: Unmodified HOPS

Problem 2: Synchronization History
   Solution: HOPS+

Evaluation
Complications from Nesting

Thread 1
- Lock L
  - Lock L2
    - S1: DATA = 1
  - Unlock L2
- --- other work ---
- Unlock L

Thread 2
- Lock L2
  - IF DATA == 1:
    - S2: FLAG == 1
  - Unlock L2

Thread 2 - Log
- DATA = 0
- FLAG = 0
- ACQ L
- ACQ L2
- S1
- REL L2
- ROLLBACK
- ACQ L2
- S2
- REL L2
- DISCARD

PM

Thread 1 - Log
- CRASH

- Log

Thread 2
- DATA = 1
- FLAG = 1
- DATA = 0
- FLAG = 0
- CRASH
- DISCARD
- ROLLBACK
Complications from Nesting

Thread 1
- Lock L
  - Lock L2
    - S1: DATA = 1
    - Unlock L2
  - <CRASH>
  - Unlock L

Thread 2
- Lock L2
  - IF DATA == 1:
    - S2: FLAG == 1
  - Unlock L2

Per-thread logs not sufficient for recovery!
Ordering Critical Sections

Revert every critical section (CS) dependent on incomplete CS

Track order in which critical sections executed
ATLAS: Synchronization History

Maintain order of synchronization operations

Thread 1 - Log
ACQ L → ACQ L2 → S1 → REL L2

Thread 2 - Log
ACQ L2 → S2 → REL L2
Problem 2: Synchronization History

Maintain order of synchronization operations

Additional work in every critical section (CS)

May revert completed critical section!!

Thread 2

Lock L2

Lookup sync history
Update sync history
IF DATA == 1:
   S2: FLAG == 1
Unlock L2
Send Network Packet
HOPS+

Only CS which see updates from incomplete CS have to be reverted

Preclude CS from seeing updates from incomplete CS

CS cannot see updates from outermost CS – mutual exclusion

Only isolate updates from nested CS till outermost CS completes
HOPS+: Closed Nesting

Isolate updates from nested CS via coherence mechanisms

Thread 1
Lock L
Lock L2
S1: DATA = 1
Unlock L2
--- other work ---
Unlock L

--- other work ---

Thread 2
Lock L2
IF DATA == 1:
S2:FLAG == 1
Unlock L2

Thread 1
Lock L
Lock L2
S1: DATA = 1
Unlock L2
--- other work ---
Unlock L

Thread 2
Lock L2
IF DATA == 1:
S2:FLAG == 1
Unlock L2

Unlock L2
HOPS++: Delayed Durability?

Updates from dependent CS buffered till outermost CS commits

Thread 1
Lock L
  Lock L2
  S1: DATA = 1
  Unlock L2

--- other work ---

Unlock L

Thread 2
Lock L2
IF DATA == 1:
  S2: FLAG == 1
Unlock L2

Thread 1
Lock L
  Lock L2
  S1: DATA = 1
  Unlock L2

--- other work ---

Unlock L

Thread 2
Lock L2
IF DATA == 1:
  S2: FLAG == 1
Unlock L2

Made Durable After
Outline

Making lock-based code failure-atomic

Problem 1: Synchronous Flushing
Solution: Unmodified HOPS

Problem 2: Synchronization History
Solution: HOPS+

Evaluation
Methodology

Extended for PM
✓ NUMA system with separate PM node
Methodology

Extended for PM
✓ NUMA system with separate PM node

✓ New memory controller for PM
• Asymmetric Read/Write Characteristics
• Configurable bandwidth and latency
• Optional Persistent Write Queue
Performance Evaluation

![Bar chart showing normalized speedup for different workloads and implementations.](chart.png)
Conclusions

Unmodified HOPS works well with durable lock-based code

HOPS+ eliminates sync history and preserves completed CS

HOPS+ may hurt concurrency by serializing CS
Thanks!

Questions?