Minimally Ordered Durable Datastructures for Persistent Memory

Swapnil Haria, Mark D. Hill, Michael M. Swift

ASPLOS 2020
Persistent memory enables recoverable applications

Analysis on Intel Optane memory reveals:
• ~73% of runtime is overhead: mostly flushing data to PM
• Overlapping flushes reduces flush costs by 75%

Minimally Ordered Durable (MOD) Datastructures:
• C++ datastructures: easy to use & good performance
• Increases flush overlap with techniques from functional datastructures
• ~40% speedup compared to PMDK-STM
• Code at https://zenodo.org/record/3563186
Outline

BACKGROUND

MOD DATASTRUCTURES

EVALUATION
Persistent Memory is Here!

User-space access to non-volatile memory

Enables recoverable applications with durable in-memory data
Programming Challenges

1. Durability
2. Failure Atomicity
Background: Software Transactional Memory

FENCE: LOG DURABLE!

FLUSH LOG

FLUSH DATA
System Crash

Use LOG to clean up the mess

Background: Software Transactional Memory
PMDK-STM performance on Optane
PMDK-STM performance on Optane

~73% of STM runtime is overhead, mostly from flushing
PMDK-STM performance on Optane

~73% of STM runtime is overhead, mostly from flushing
Flush (CLWB) Overheads on Optane

Flushing overhead falls with overlap (following Amdahl’s Law)
Flush (CLWB) Overheads on Optane

PM workloads have up to 23 flushes and 11 fences per transaction!

Flushing overhead falls with overlap (following Amdahl’s Law)
Goal: Minimize Ordering!

Reduce FENCEs (ordering), even if extra computation required

How to provide failure-atomicity with minimal ordering?

**Shadow Paging:** Out-of-place updates instead of overwriting data
shadow = array // Create shadow copy
shadow[index1] = X
shadow[index2] = Y
FLUSH (shadow)
FENCE
// Application uses shadow subsequently
array = shadow
Purely Functional data structures are immutable

Implemented as efficient trees: Hash Array Mapped trie, RRBTree

Copying overheads reduced by structural sharing
Minimally Ordered Durable Datastructures

Recoverable datastructures adapted from existing functional ones

Durability: PM allocator + Flushes

Failure-Atomicity: Fences + out-of-place updates

Leverage 20+ years of work from functional programming community

Read/Write APIs that hides flushes, fences, out-of-place updates
Atomic Update of Single Datastructure

Update(arrayPtr, index, value) // Atomic, Durable w/ 1 FENCE
Atomic Update of Single Datastructure

\[
\text{shadowArray} = \text{arrayPtr->Update(updateParams)} \\
\text{FENCE} \\
\text{arrayPtr} = &\text{shadowArray}
\]
Advanced MOD usages

Multiple Atomic Updates to One Datastructure (in the paper)

Atomically Updating Multiple Datastructures
3: Updating Multiple Datastructures

ds1PtrShadow = ds1Ptr->Update1(updateParams1)
ds2PtrShadow = ds2Ptr->Update2(updateParams2)
...

BEGIN TX {
    ds1Ptr = ds1PtrShadow
    ds2Ptr = ds2PtrShadow
    ...
}
END TX

All Flushes Overlapped

CONCET (ds1Ptr, ds1PtrShadow,
Begin-TX{
    ds2Ptr, ds2PtrShadow, ...)
    ds1Ptr = ds1PtrShadow
    ds2Ptr = ds2PtrShadow
    ...
} End-TX

More ordering points but short transaction
3: Updating Multiple Datastructures

\[
ds1\text{PtrShadow} = ds1\text{Ptr} -> \text{Update1}(\text{updateParams1})
\]
\[
ds2\text{PtrShadow} = ds2\text{Ptr} -> \text{Update2}(\text{updateParams2})
\]

... 

\text{FENCE}

\text{Begin-TX} \{ 
\hspace{2em} \text{ds1Ptr} = \text{ds1PtrShadow} \\
\hspace{2em} \text{ds2Ptr} = \text{ds2PtrShadow} \\
\hspace{2em} \ldots
\}

\text{End-TX}

\text{Paper describes alternate method w/o transactions that handles many such cases}
Evaluation Methodology

Used C++ library of functional datastructures: https://github.com/arximboldi/immer

Used off-the-shelf persistent memory allocator: https://github.com/hyrise/nvm_malloc.git

MOD library released at: https://zenodo.org/record/3563186

Compared against Intel PMDK v1.5 (hybrid undo-redo logging): https://github.com/pmem/pmdk
Performance Comparison on Optane
MOD offers ~43% speedup for pointer-based datastructures (map, set, stack, queue)
MOD degrades update performance of vectors by 120%
MOD speeds up our 3 recoverable applications by ~38%
Summary

Persistent memory enables recoverable applications

Analysis on Intel Optane memory reveals:
• ~73% of runtime is overhead: mostly flushing
• Flushing overhead reduces by 75% with flush overlap

Minimally Ordered Durable Datastructures:
• C++ datastructures: easy to use & good performance
• Increases flush overlap with techniques from functional datastructures
• ~40% speedup compared to PMDK-STM
• Code at https://zenodo.org/record/3563186