Minimally Ordered Durable Datastructures for PM

Swapnil Haria, Mark D. Hill, Michael M. Swift
University of Wisconsin-Madison
Promise of Persistent Memory

Care about Memory

NOT OUR FOCUS

Recoverable Applications

Care about Storage

NOT OUR FOCUS
Vision

Let programmers build recoverable applications with durable datastructures.

Hide the details of persistence within libraries of efficient datastructures (i.e., C++ STL).

Map    Set    Stack    Vector    Queue
Outline

Introduction

Analyzing durability overheads on Optane

MOD Datastructures

Evaluation
1. Durability
2. Consistency
3. (Failure) Atomicity

Programming Challenges
BEGIN-TX
value1 = array[index1]
value2 = array[index2]
LOG (index1, value1); FLUSH LOG; FENCE
array[index1] = value2
FLUSH (&array[index1])
LOG (index2, value2); FLUSH LOG; FENCE
array[index2] = value1
FLUSH (&array[index2])
FENCE
END-TX
Use LOG to clean up the mess

System Crash
Outline

Introduction

Analyzing durability overheads on Optane

MOD Datastructures

Evaluation
STM performance on Optane
Measuring Flush Overheads on Optane

![Graph showing execution time vs. fence to flush ratio]
Reduce FENCEs (ordering), even if extra computation required
Outline

Introduction

Analyzing durability overheads on Optane

MOD Datastructures

Evaluation
Goal: Minimize Ordering!

How to provide atomicity and consistency with minimal ordering? Databases, Filesystems: **Shadow Paging**

**Key Idea:** Avoid overwriting data!
Ordering in STM

BEGIN-TX
value1 = array[index1]
value2 = array[index2]
LOG (index1, value1); FLUSH LOG; FENCE
array[index1] = value2
FLUSH (&array[index1])
LOG (index2, Y); FLUSH LOG; FENCE
array[index2] = value1
FLUSH (&array[index2])
FENCE
END-TX
Intention: Minimize Ordering!

How to provide atomicity and consistency with minimal ordering?
Databases, Filesystems: Shadow Paging

**Key Idea**: Avoid overwriting data!

**Mechanism**: Out-of-place updates
value1 = array[index1]
value2 = array[index2]
shadow = array // Create shadow copy
shadow[index1] = value2
shadow[index2] = value1
FLUSH (shadow)
FENCE
// Application uses shadow subsequently
Déjà vu? Functional Datastructures!

*Purely Functional* datastructures are immutable

Implemented as efficient trees: Hash Array Mapped trie, RRBTree

Copying overheads reduced by *structural sharing*
Functional Shadowing

Functional Datastructures to reduce shadow paging overheads

Shadow paging to minimize ordering constraints
Recipe for building MOD datastructures

1. Find efficient functional datastructures in language of choice
2. Pick any persistent memory allocator
3. Allocate internal state on the persistent heap
4. Flush modified PM cachelines in all update operations
5. Profit?
MOD usecases

- One Update to One Datastructure (Atomic, Consistent, Durable)

- Multiple Updates to One Datastructure (Atomic, Consistent, Durable)

- Updating Multiple Datastructures (Atomic, Consistent, Durable)
1: One Update to One Datastructure

dsptrShadow = dsptr->Update(updateParams)
FENCE
Update(arrayPtr, Index, Value)
// Atomic, Durable, Consistent with 1 FENCE
 dsptr = dsptrShadow

All Flushes Overlapped
2: Multiple Updates to One Datastructure

dsPtrShadow1 = dsPtr->Update1(updateParams)
dsPtrShadow2 = dsPtrShadow1->Update2(updateParams)
Commit (dsPtr, dsPtrShadow1, dsPtrShadow2)
dsPtr = dsPtrShadow2

All Flushes Overlapped
3: Updating Multiple Datastructures

ds1Ptrshadow = ds1Ptr->Update1(updateParams1)
ds2Ptrshadow = ds2Ptr->Update2(updateParams2)

\texttt{Commit (ds1Ptr, ds1PtrShadow,}
\texttt{ds2Ptr, ds2PtrShadow)}

\texttt{Begin-TX \{}
\texttt{ds1Ptr = ds1PtrShadow}
\texttt{ds2Ptr = ds2PtrShadow}
\texttt{\}} \texttt{End-TX}

\textbf{All Flushes Overlapped}

\textbf{More ordering points but short transaction}
Open Questions

Concurrency
  Read-Copy-Update style concurrency could be interesting

Preventing MemoryLeaks
  Allocator can lose newly allocated data in case of crash

Further Optimizations
  Batching updates, in-place updates in shadows, etc.
Outline

Introduction

Analyzing durability overheads on Optane

MOD Datastructures

Evaluation
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

Compared against Intel PMDK v1.5 (Oct 2018)
https://github.com/pmem/pmdk
Performance Comparison on Optane

Execution Time Normalized to PMDK
Performance Comparison on Optane

![Bar chart showing execution time normalized to PMDK for various operations and scenarios.]
MOD: Summary

Library of recoverable data structures: map, set, vector, stack, queue

Minimizes number of ordering points in software

Uses functional shadowing instead of STM

Offers 60% performance improvements over current STM
Thanks!

Questions?