Minimally Ordered Durable Datastructures for Persistent Memory

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



ASPLOS 2020

Executive Summary

Persistent memory enables recoverable applications

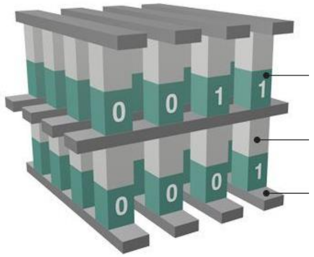
Analysis on Intel Optane memory reveals:

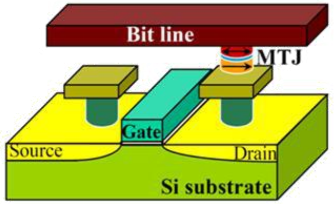
- ~73% of runtime is overhead: mostly flushing data to PM
- Overlapping flushes reduces flush costs by 75%

Minimally Ordered Durable (MOD) Datastructures:

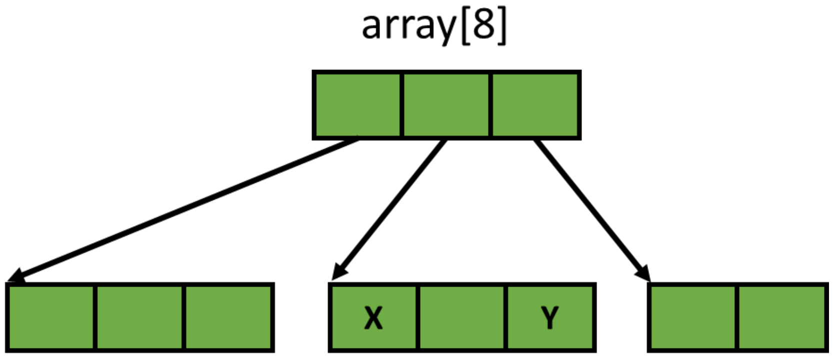
- C++ datastructures: easy to use & good performance
- Increases flush overlap with techniques from functional datastructures
- ~40% speedup compared to PMDK-STM
- Code at https://zenodo.org/record/3563186

Outline

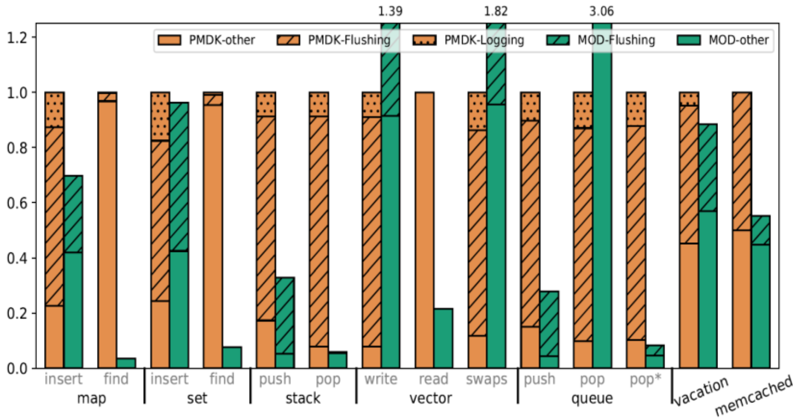




BACKGROUND

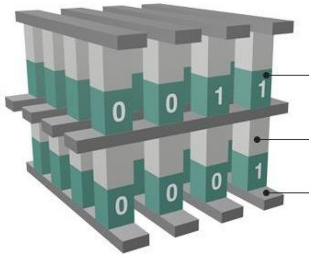


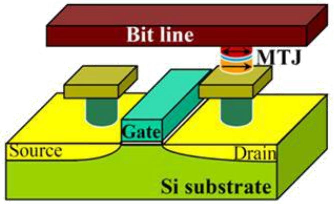
MOD DATASTRUCTURES



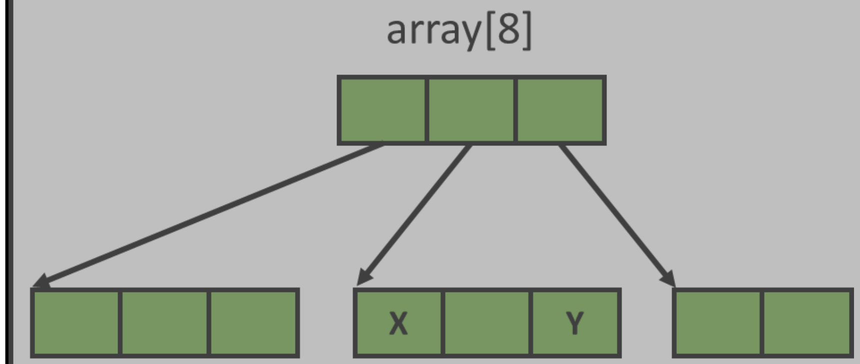
EVALUATION

Outline

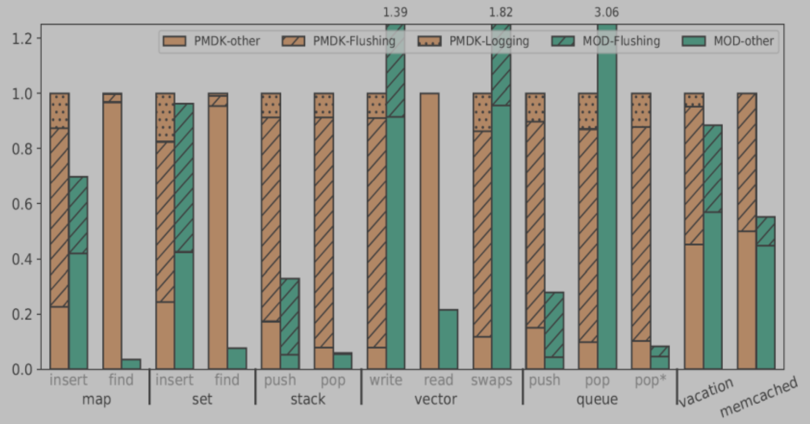




BACKGROUND



MOD DATASTRUCTURES

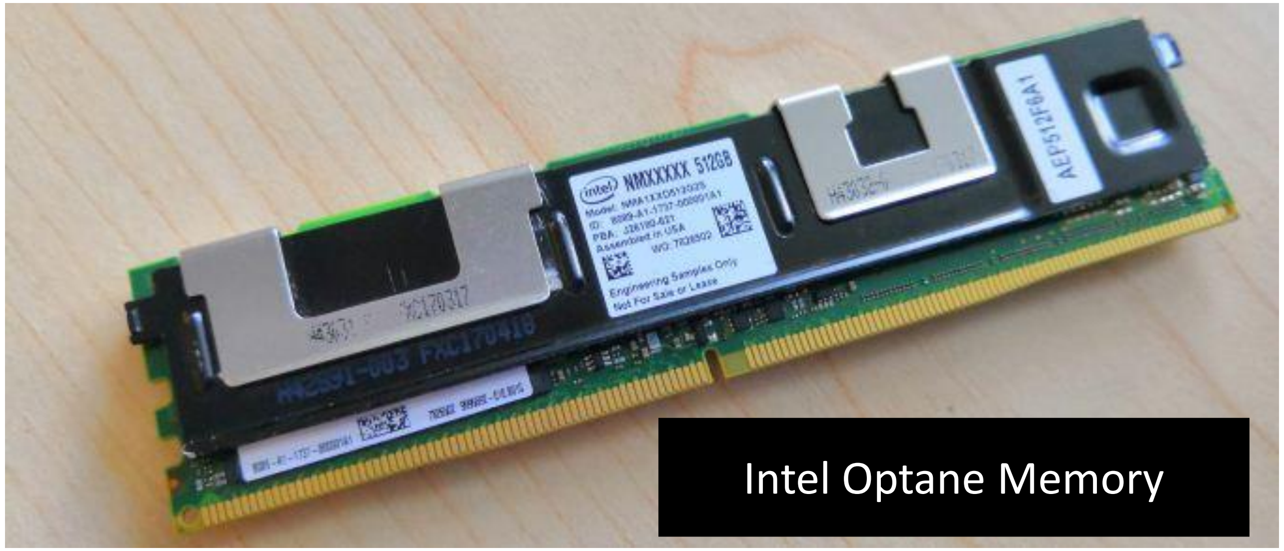


EVALUATION

Persistent Memory is Here!

User-space access to non-volatile memory

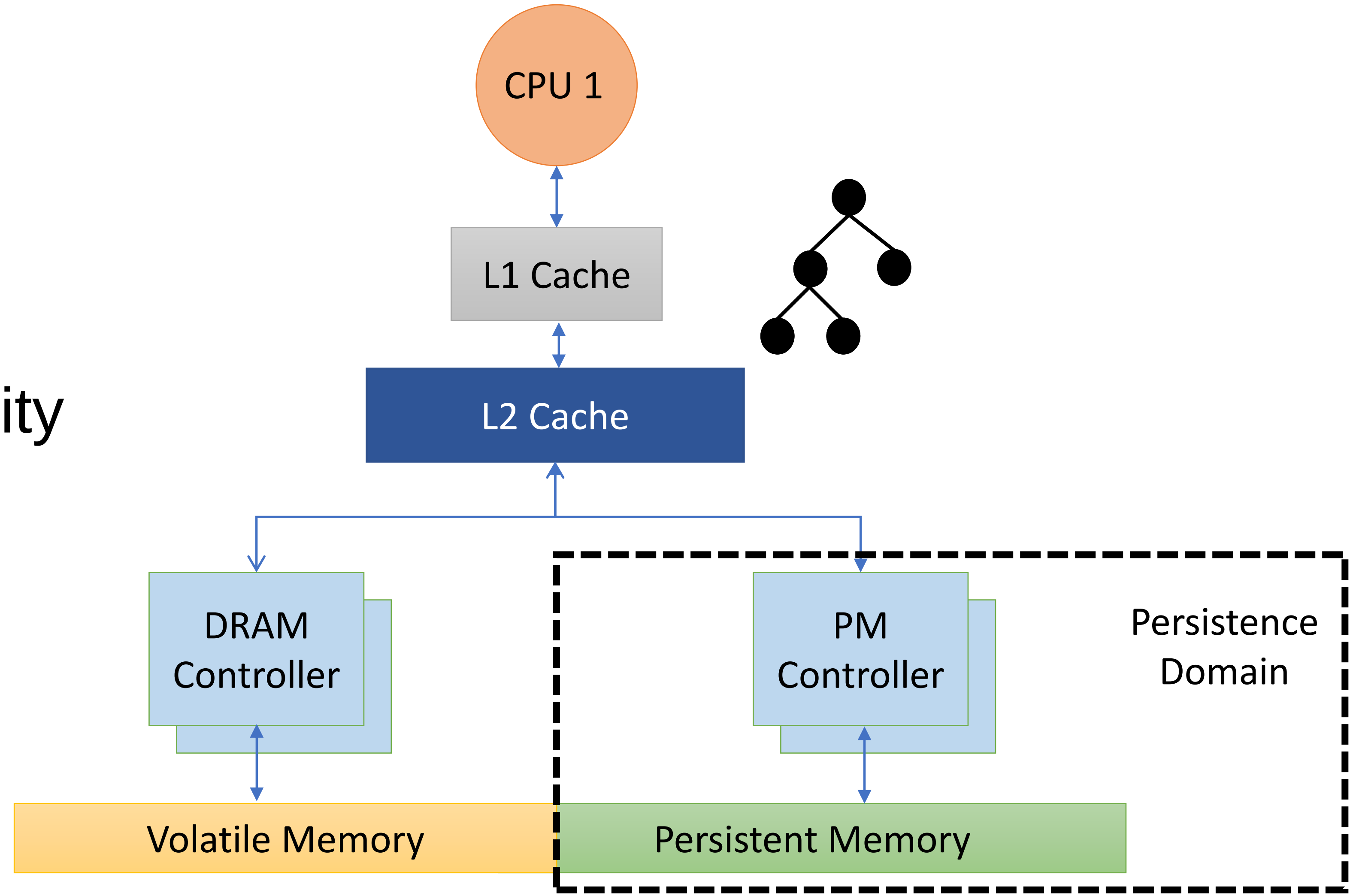
Enables recoverable applications with durable in-memory data



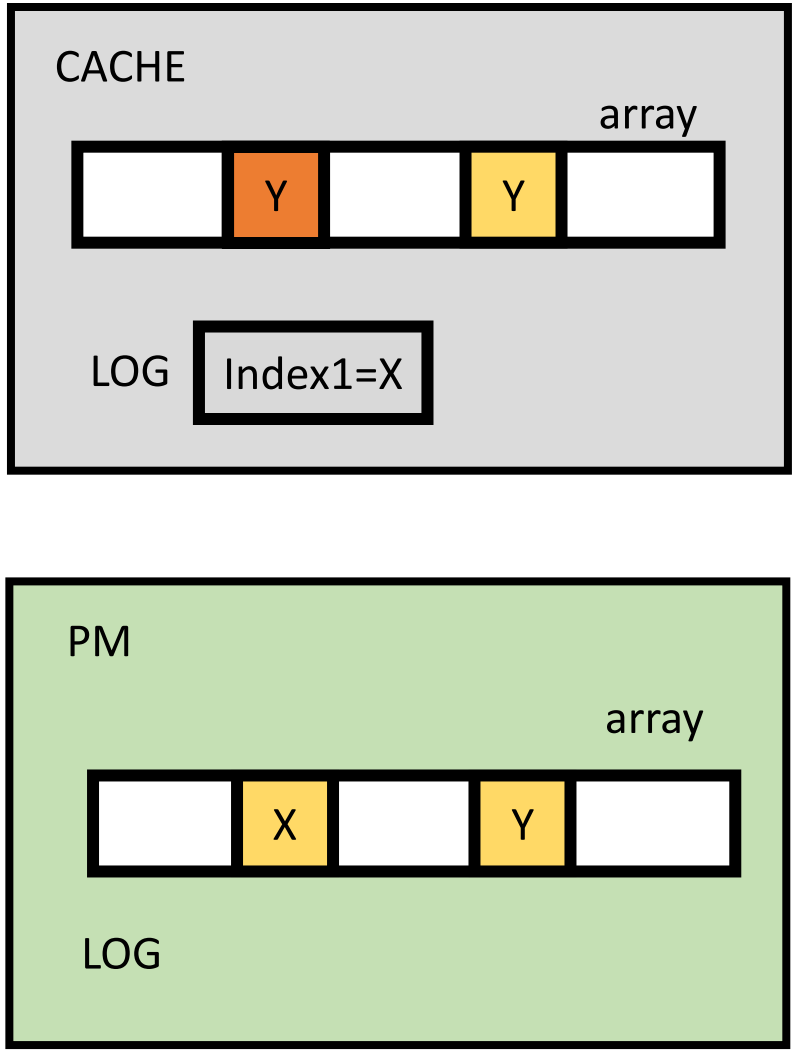
Intel Optane Memory

Programming Challenges

1. Durability
2. Failure Atomicity



Background: Software Transactional Memory



FLUSH LOG

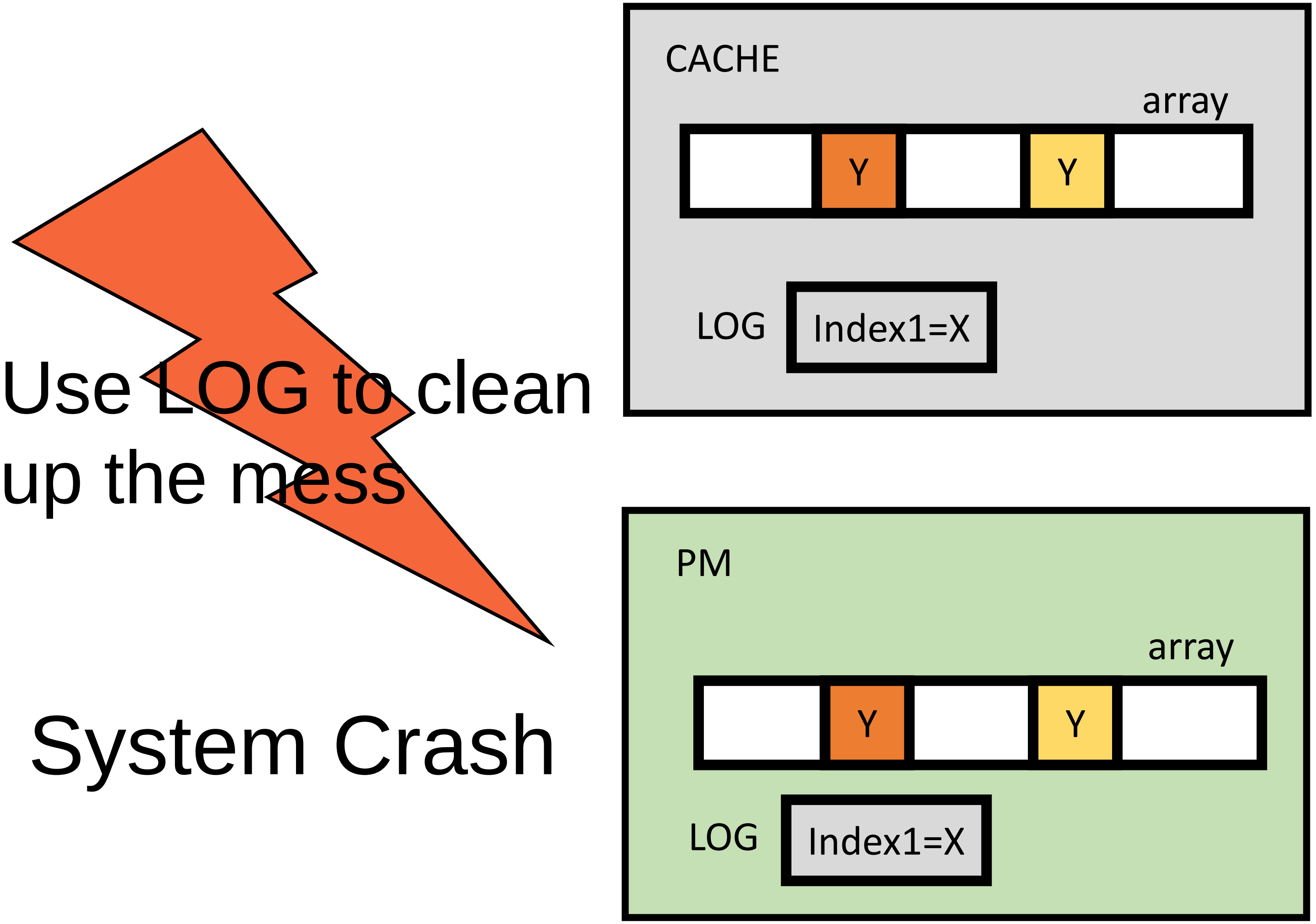
FENCE: LOG DURABLE!

FLUSH DATA

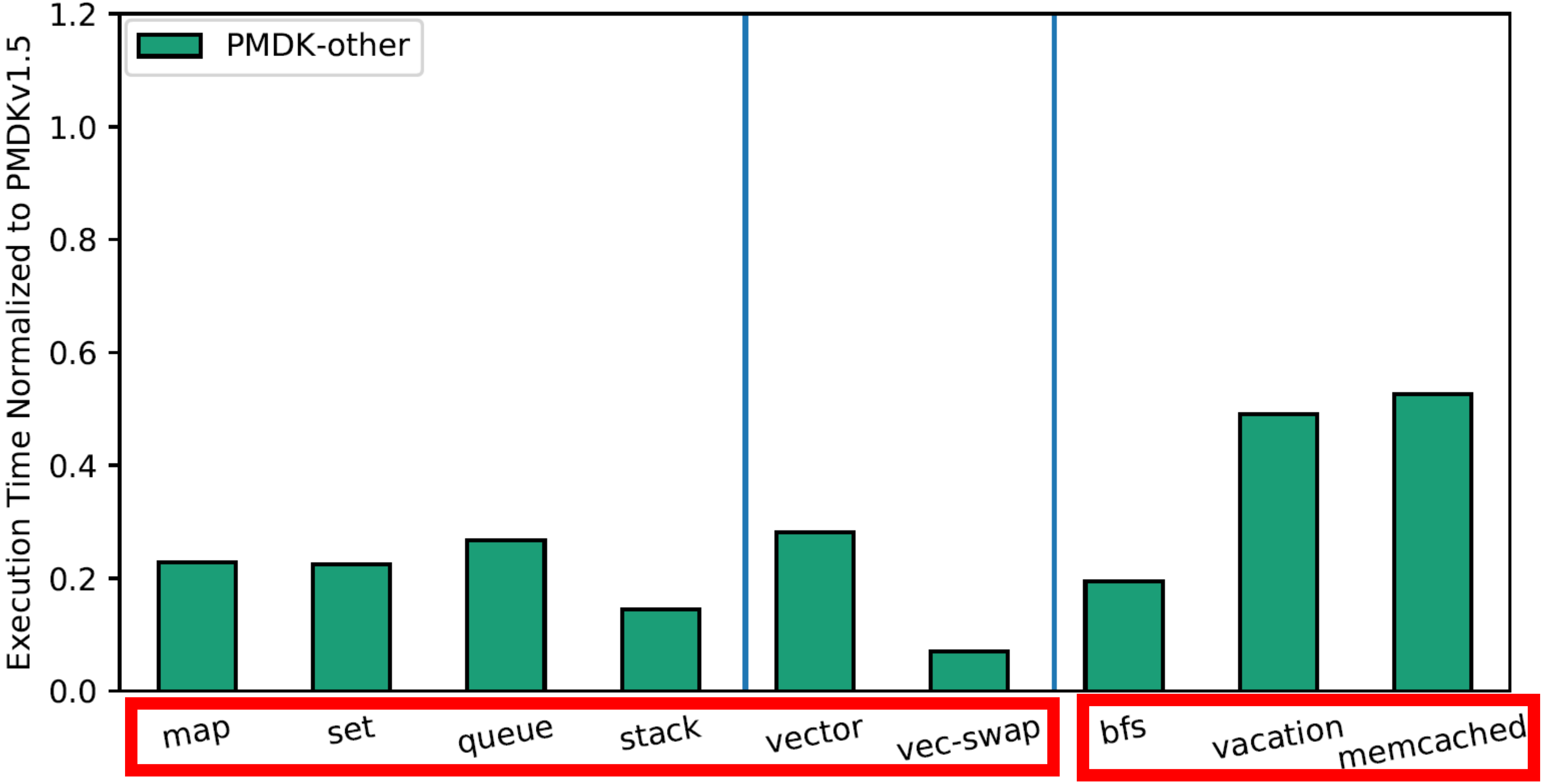
Background: Software Transactional Memory

Use LOG to clean up the mess

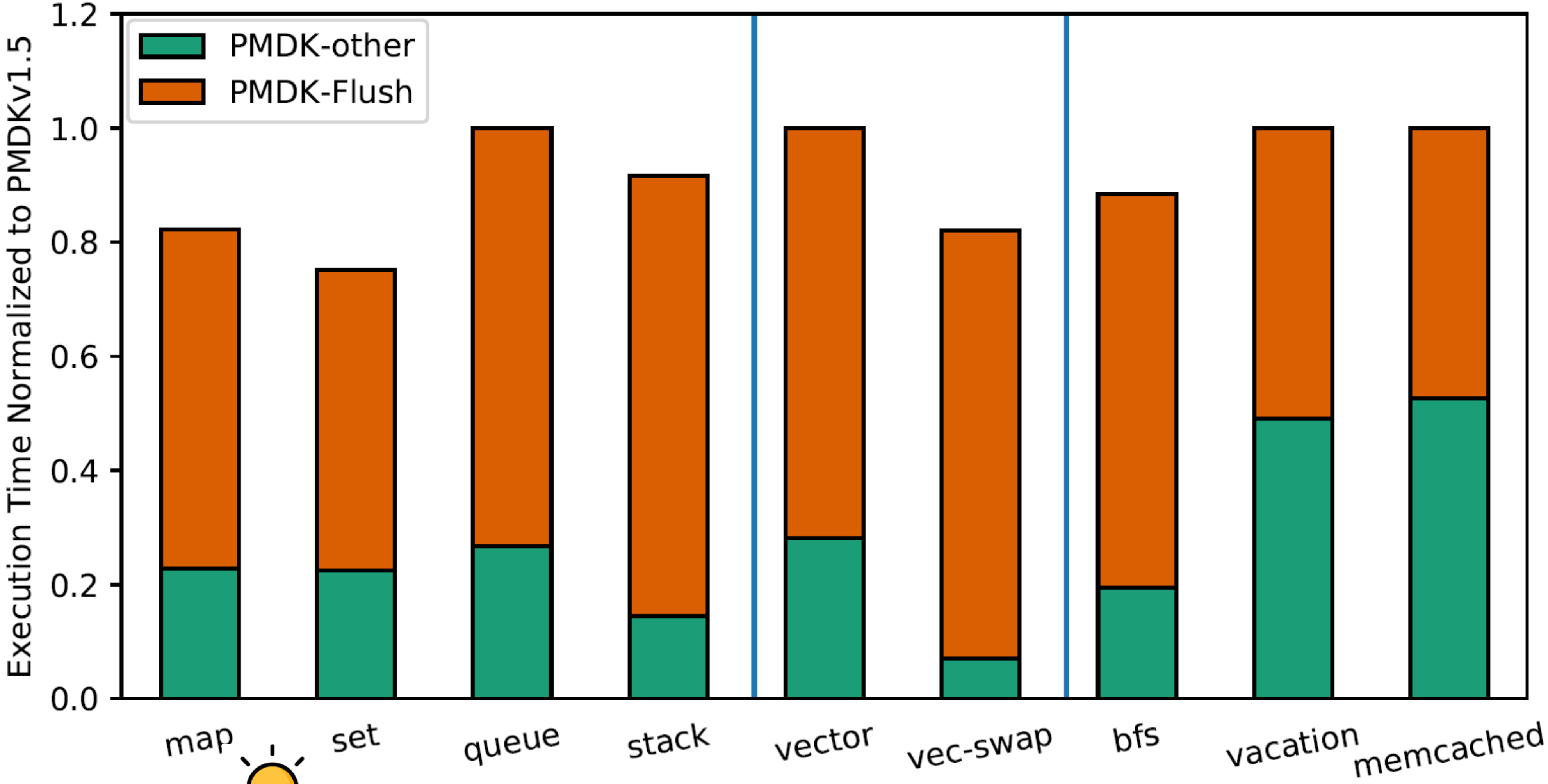
System Crash



PMDK-STM performance on Optane

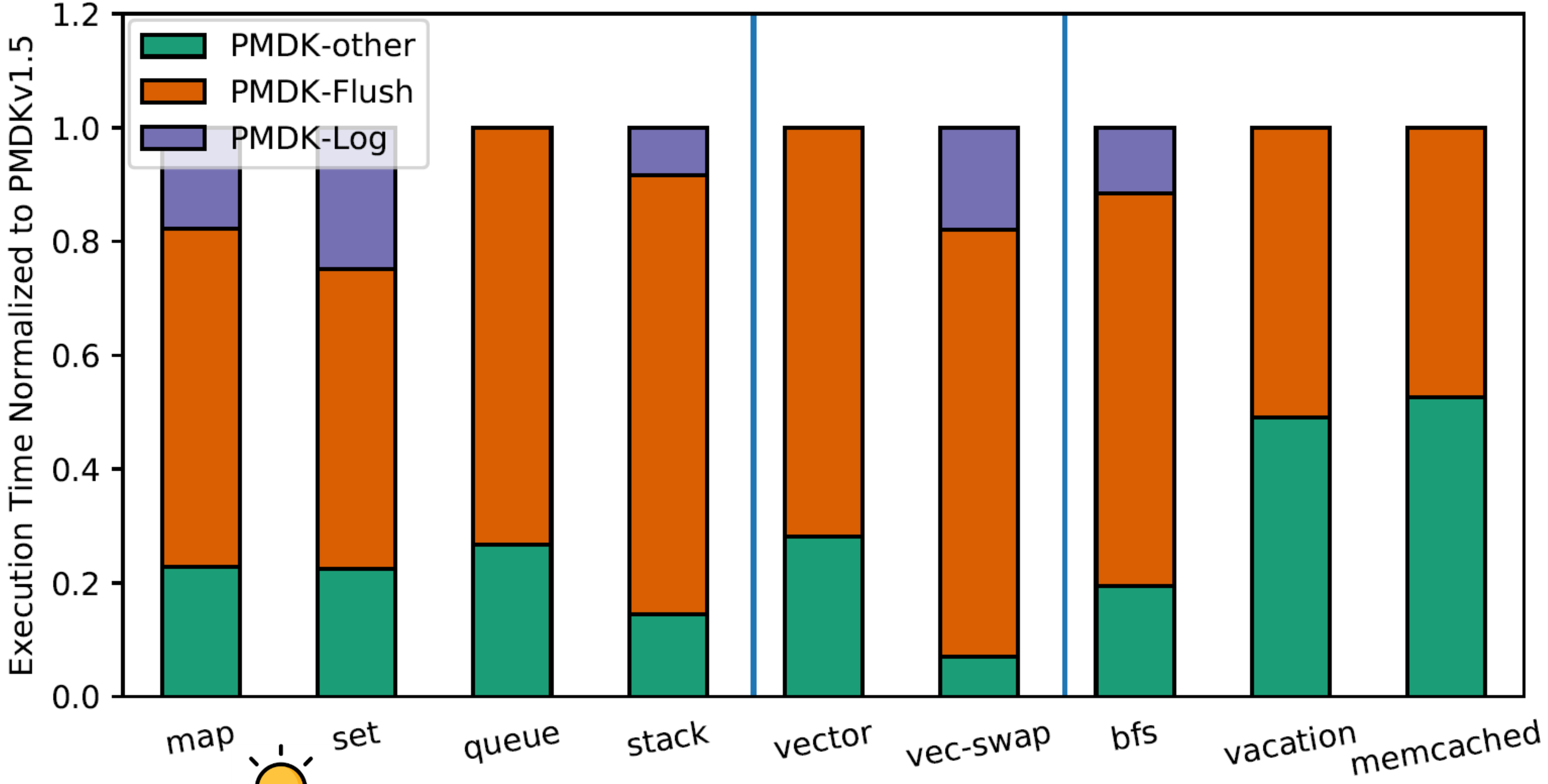


PMDK-STM performance on Optane



~73% of STM runtime is overhead, mostly from flushing

PMDK-STM performance on Optane

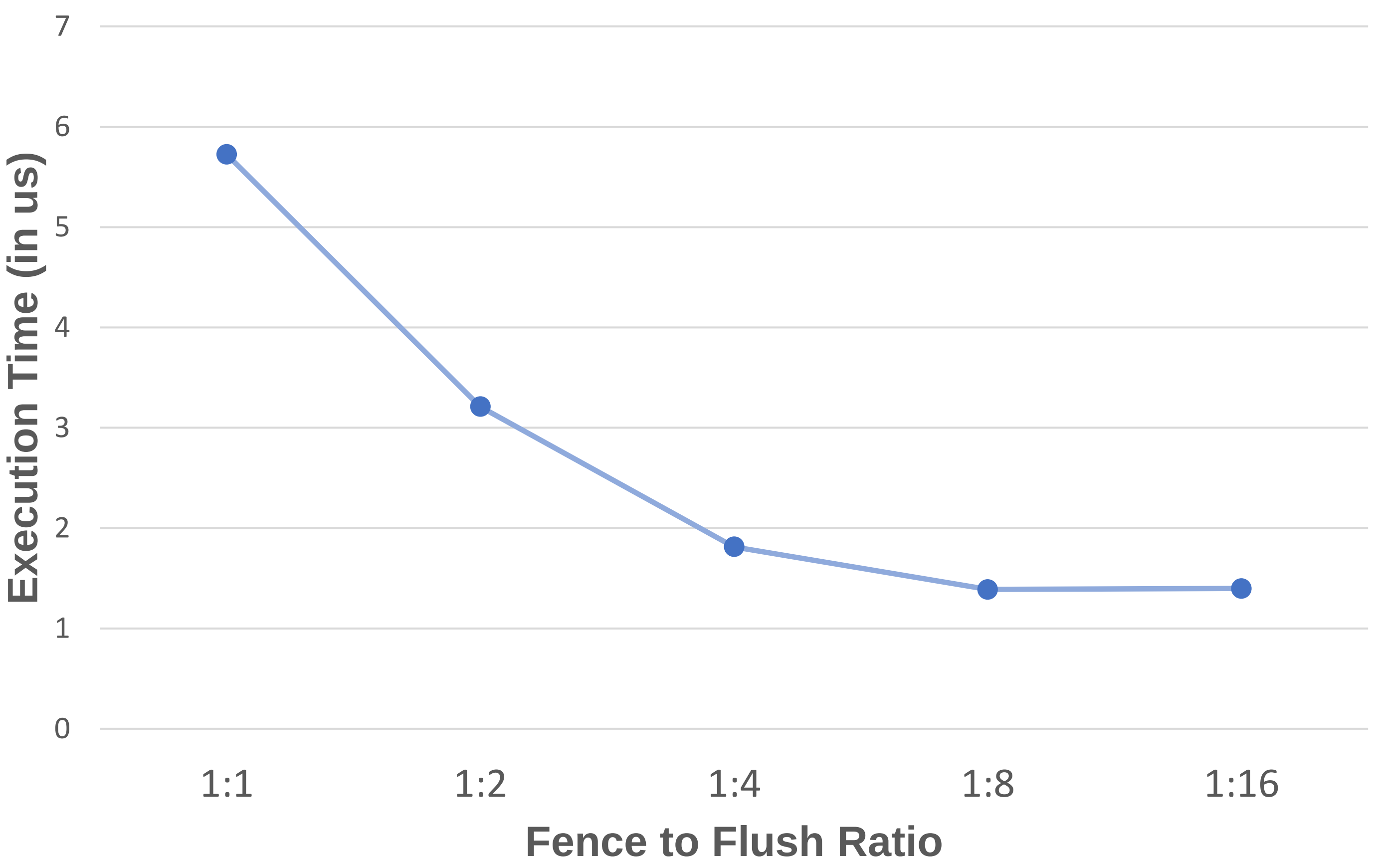


~73% of STM runtime is overhead, mostly from flushing

Flush (CLWB) Overheads on Optane

FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE

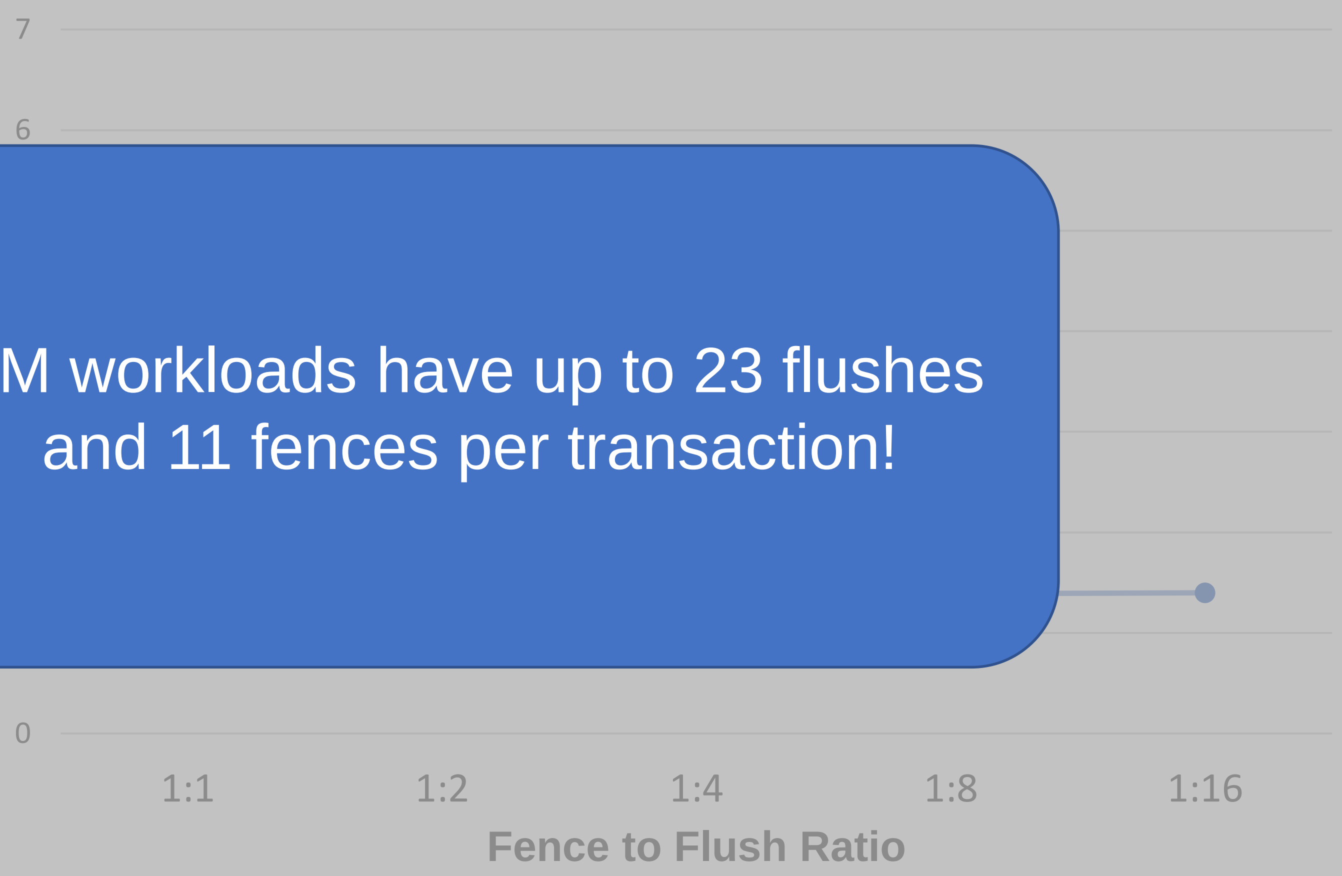
FLUSH
FENCE
FLUSH
FENCE
...



Flushing overhead falls with overlap (following Amdahl's Law)

Flush (CLWB) Overheads on Optane

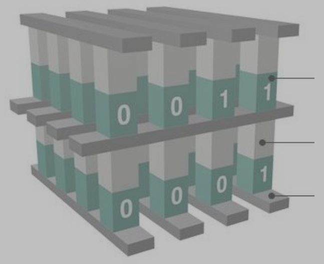
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE
FLUSH
FENCE

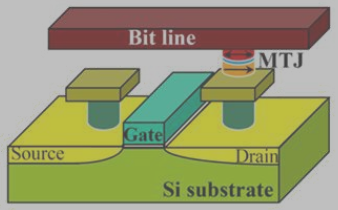


PM workloads have up to 23 flushes and 11 fences per transaction!

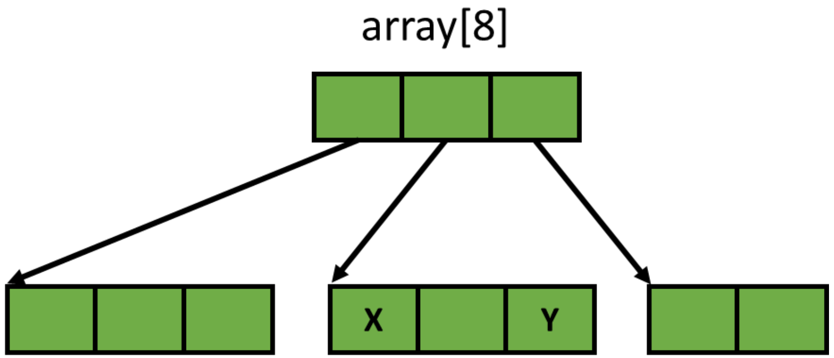
Flushing overhead falls with overlap (following Amdahl's Law)

Outline

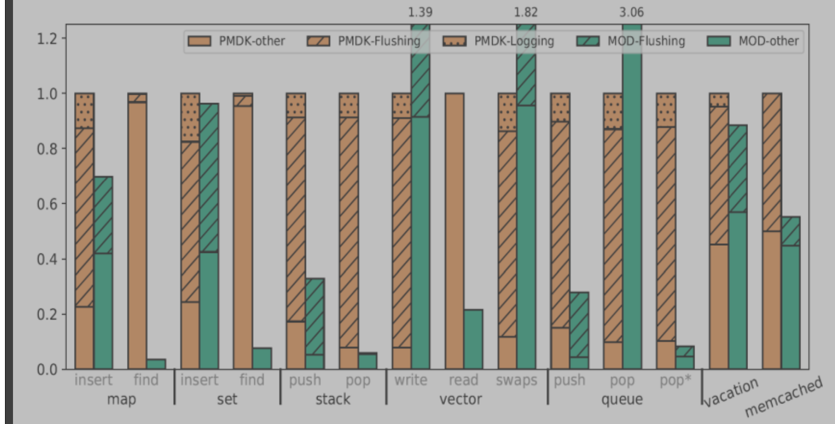




BACKGROUND



MOD DATASTRUCTURES



EVALUATION

Goal: Minimize Ordering!

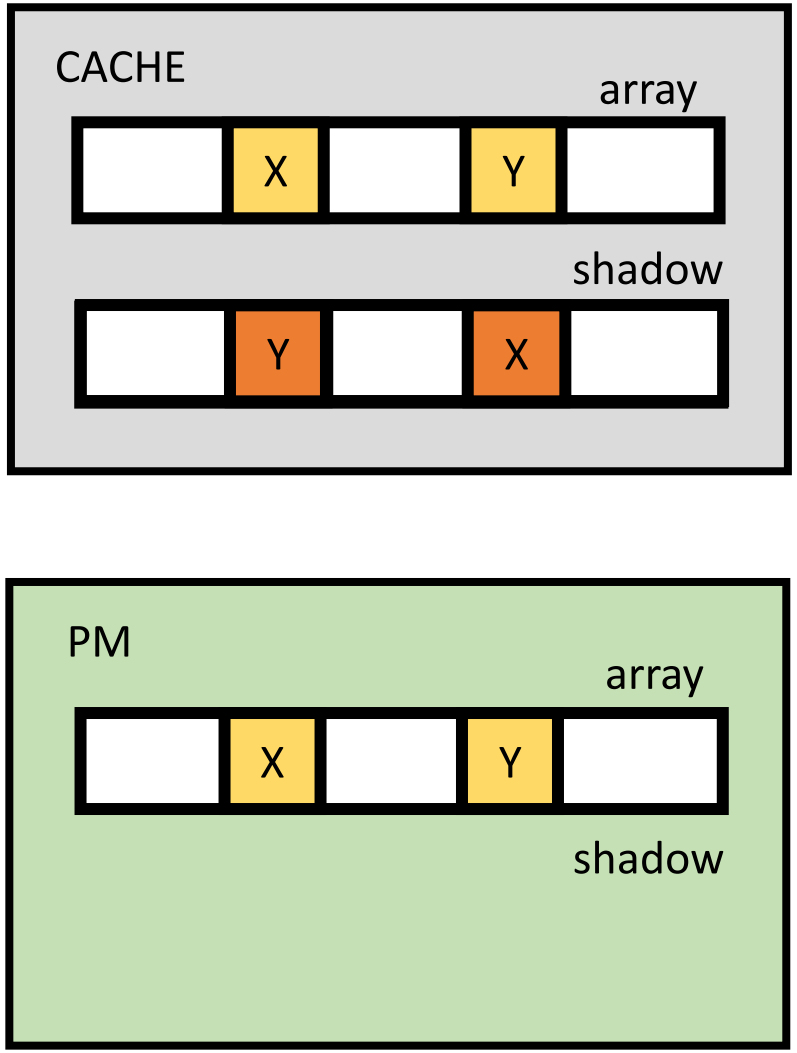
Reduce FENCEs (ordering), even if extra computation required

How to provide failure-atomicity with minimal ordering?

Shadow Paging: Out-of-place updates instead of overwriting data

Background: Shadow Paging

shadow = array // Create shadow copy
shadow[index1] = X
shadow[index2] = Y
FLUSH (shadow)
FENCE
// Application uses shadow subsequently
array = shadow

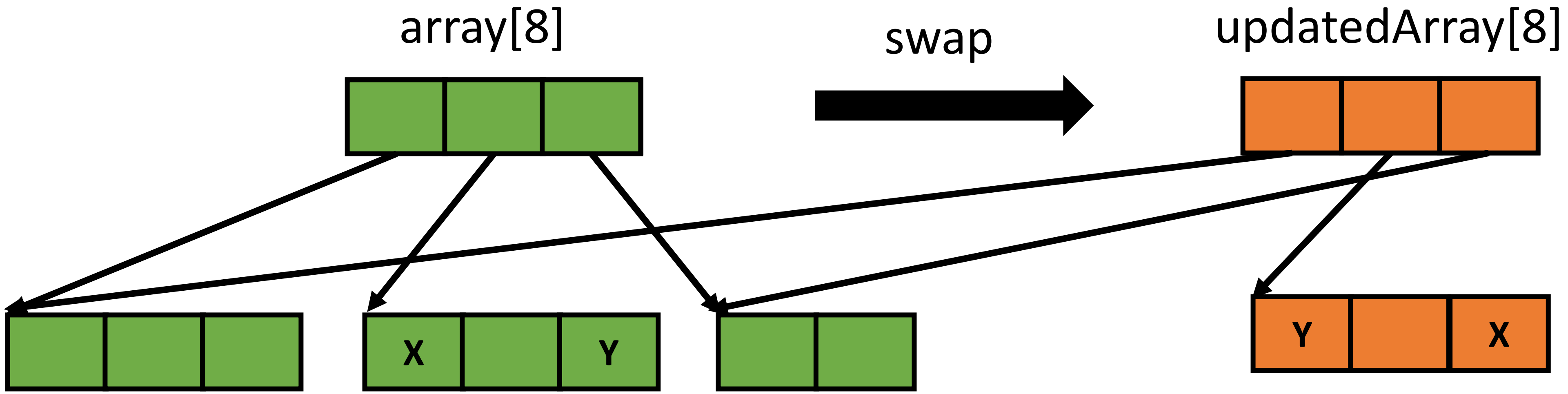


Cue Functional Datastructures!

Purely Functional datastructures 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
```

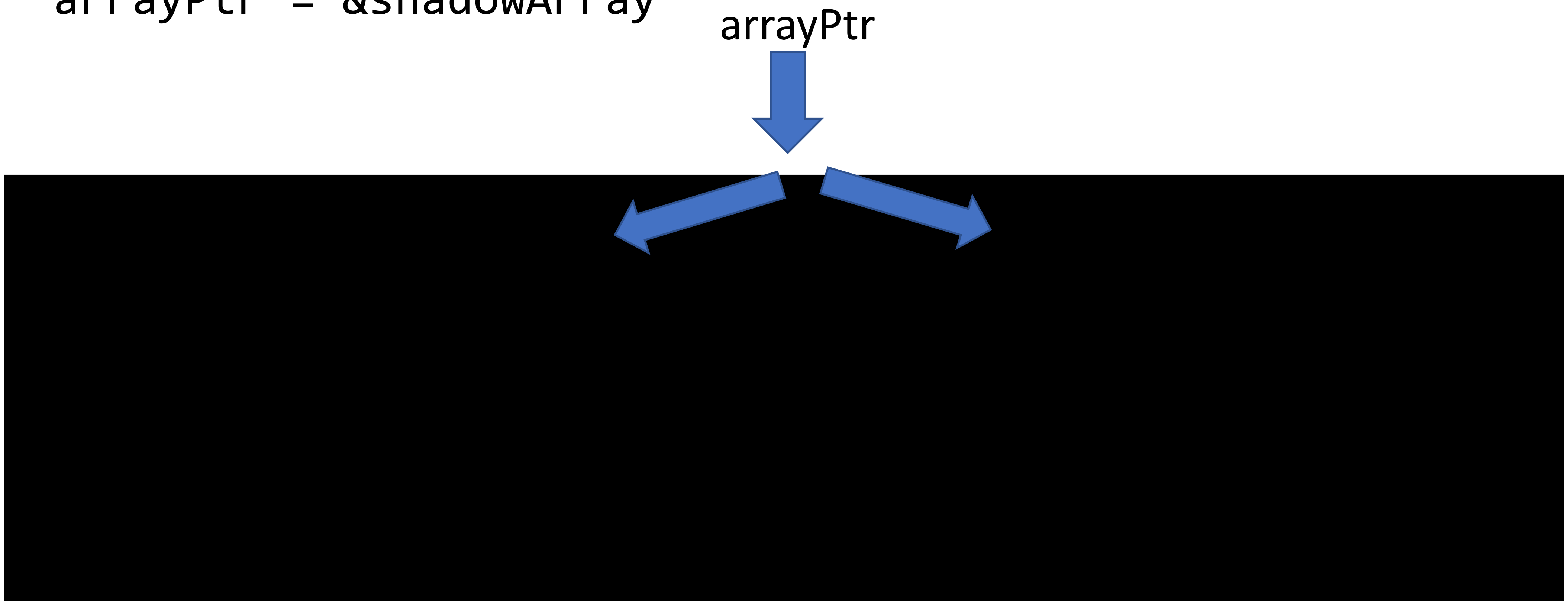
arrayPtr

[redacted]

Atomic Update of Single Datastructure

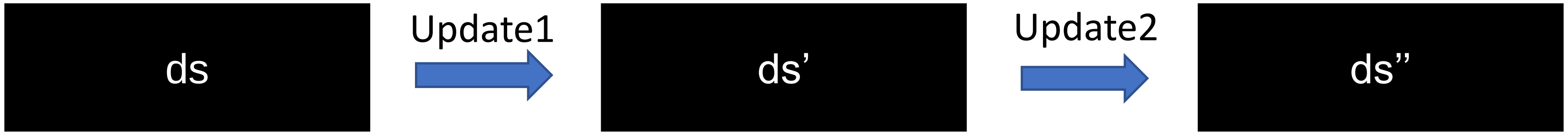
```
shadowArray = arrayPtr->Update(updateParams)
FENCE
arrayPtr = &shadowArray
```

Overlapped Flushes



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)
...
FENCE
Commit (ds1Ptr, ds1PtrShadow,
        ds2Ptr, ds2PtrShadow, ...)
Begin-TX {
          ds1Ptr = ds1PtrShadow
          ds2Ptr = ds2PtrShadow
          ...
} End-TX
```

All Flushes Overlapped

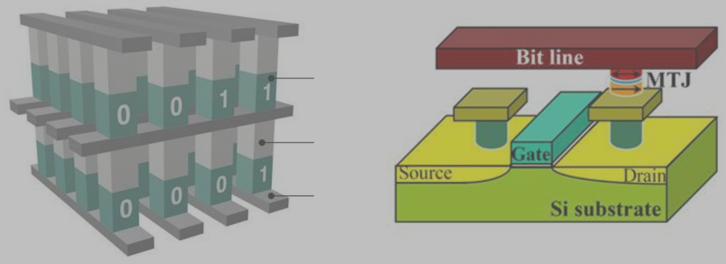
More ordering points but short transaction

3: Updating Multiple Datastructures

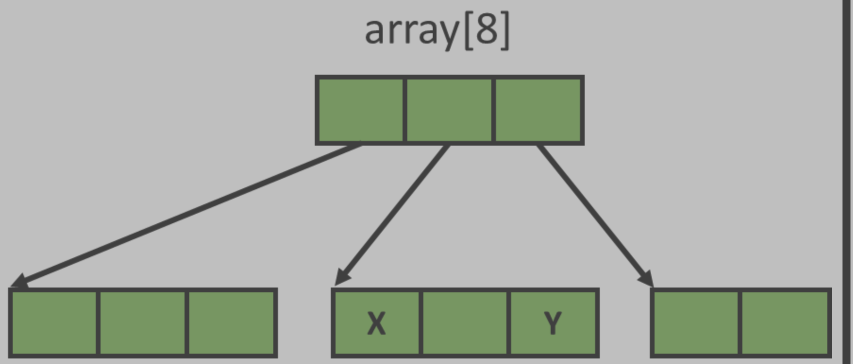
```
ds1PtrShadow = ds1Ptr->Update1(updateParams1)
ds2PtrShadow = ds2Ptr->Update2(updateParams2)
...
FENCE
Begin-TX {
          ds1P      trShadow
          ds2Pt      PtrShadow
          ...
} End-TX
```

Paper describes alternate method w/o transactions that handles many such cases

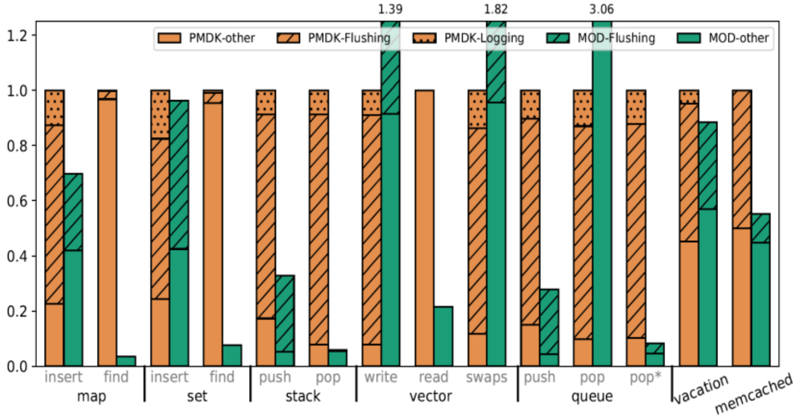
Outline



BACKGROUND



MOD DATASTRUCTURES



EVALUATION

Evaluation Methodology

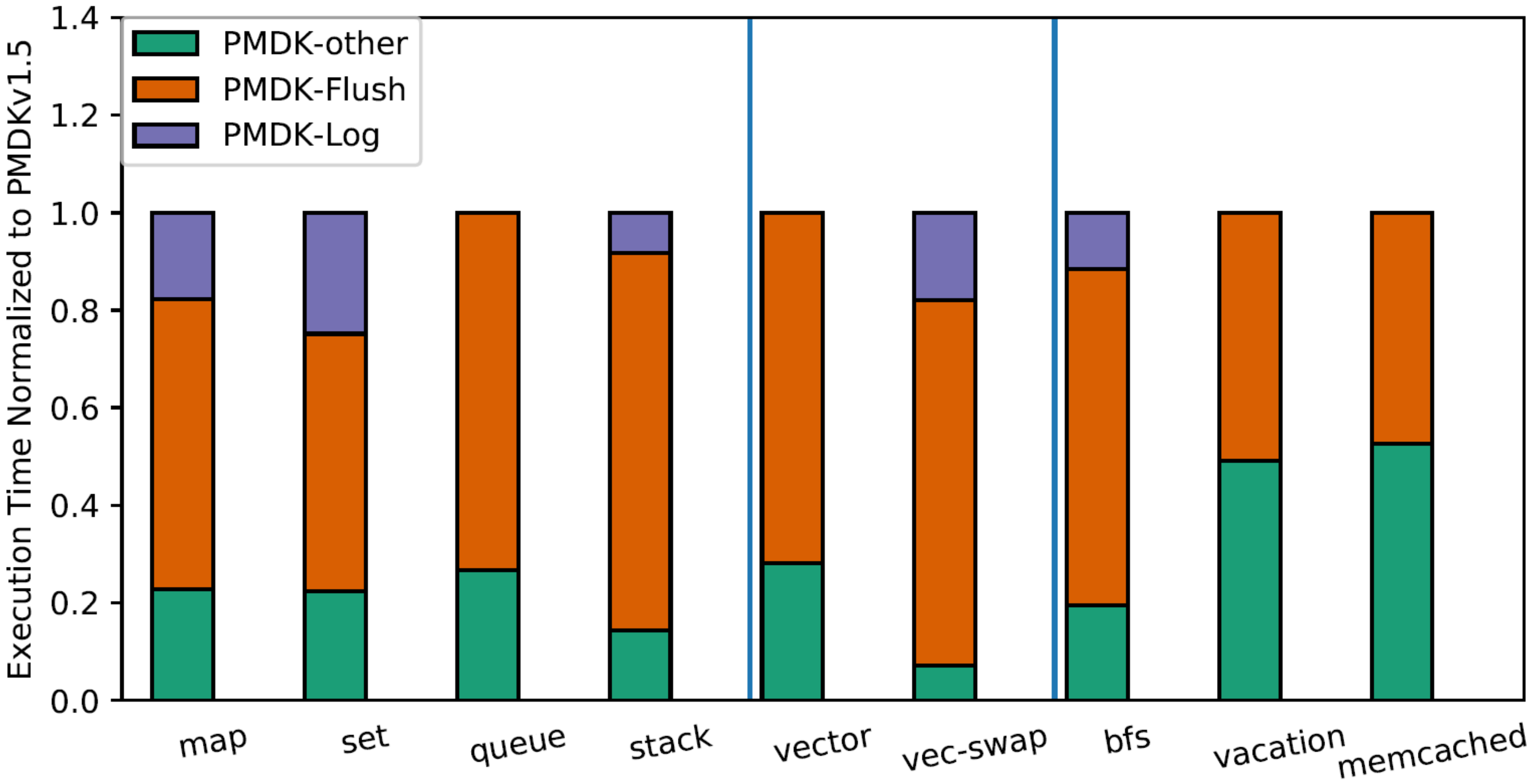
Used C++ library of functional datastructures:
https://github.com/arximboldi/immer

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

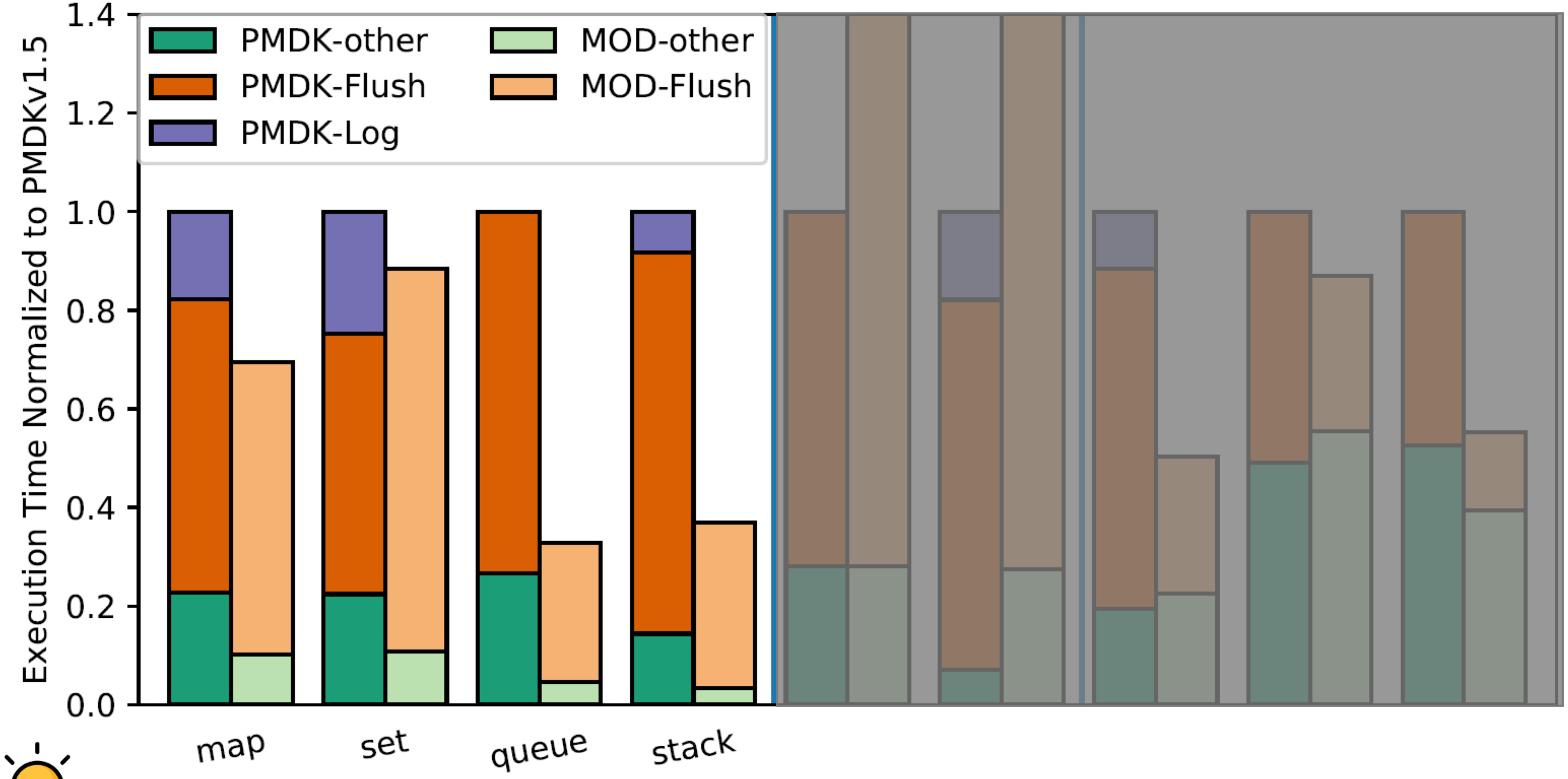
MOD library released at:
https://zenodo.org/record/3563186

Compared against Intel PMDK v1.5 (hybrid undo-redo logging):
https://github.com/pmem/pmdk

Performance Comparison on Optane

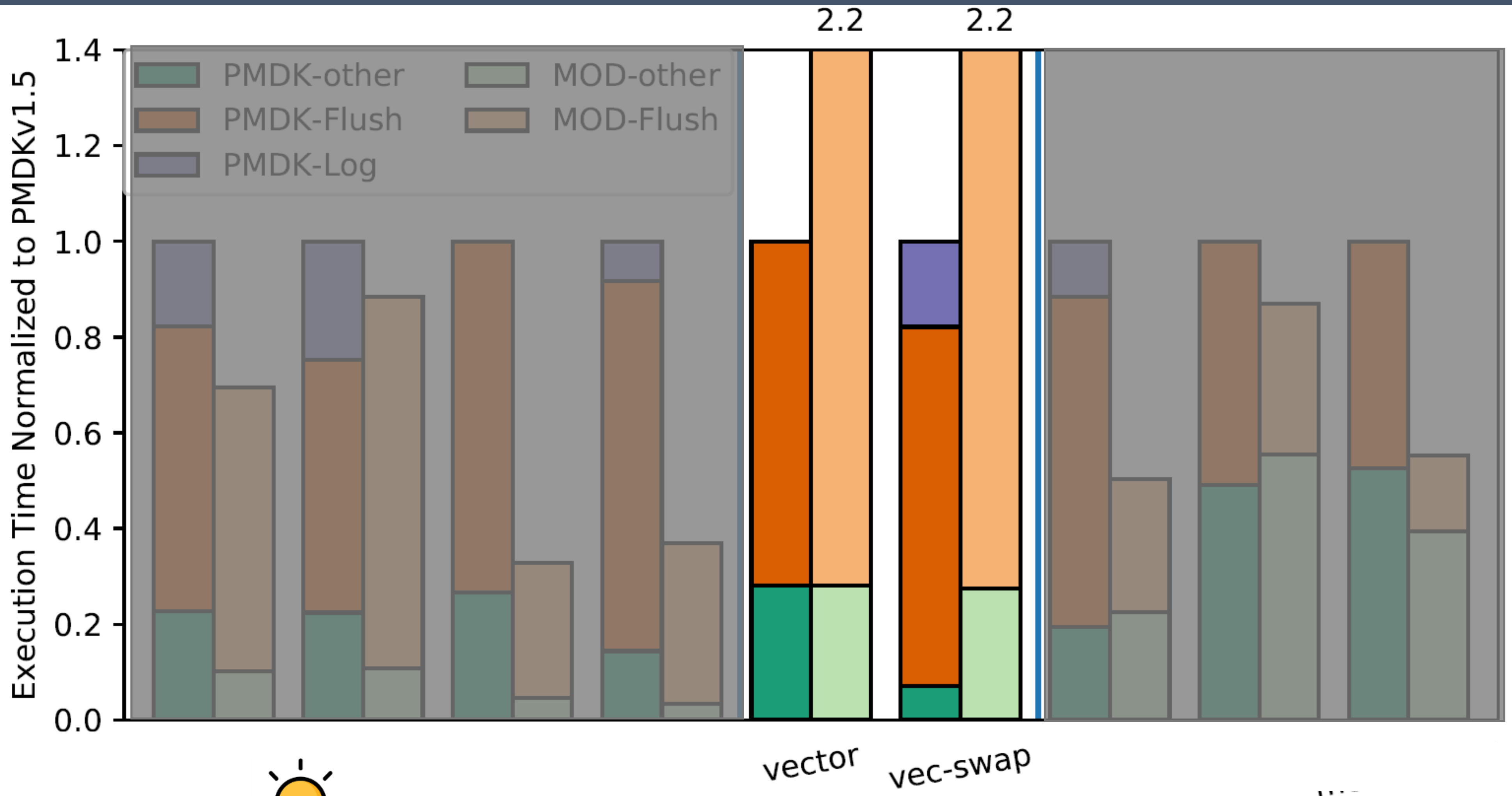


Performance Comparison on Optane



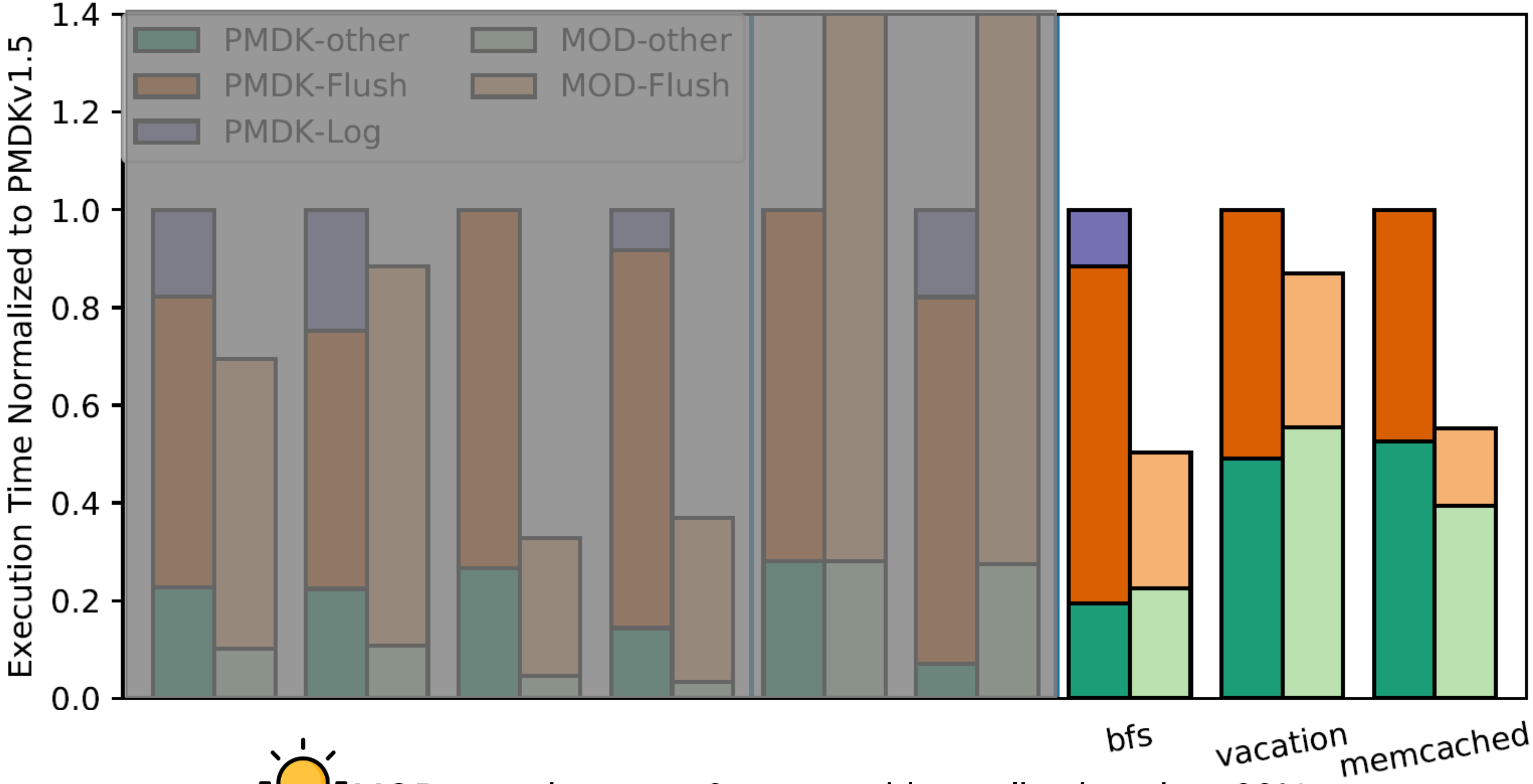
MOD offers ~43% speedup for pointer-based datastructures (map, set, stack, queue)

Performance Comparison on Optane



MOD degrades update performance of vectors by 120%

Performance Comparison on Optane



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