Non-Volatile Memory Through Customized Key-Value Stores

Leonardo Mármol [1] Jorge Guerra [2] Marcos K. Aguilera [2]

[1]Florida International University

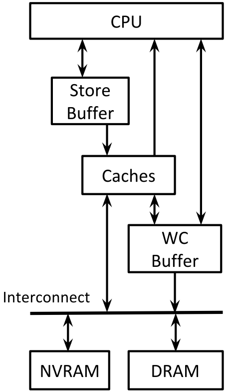
[2]VMware

Characteristics of NVM

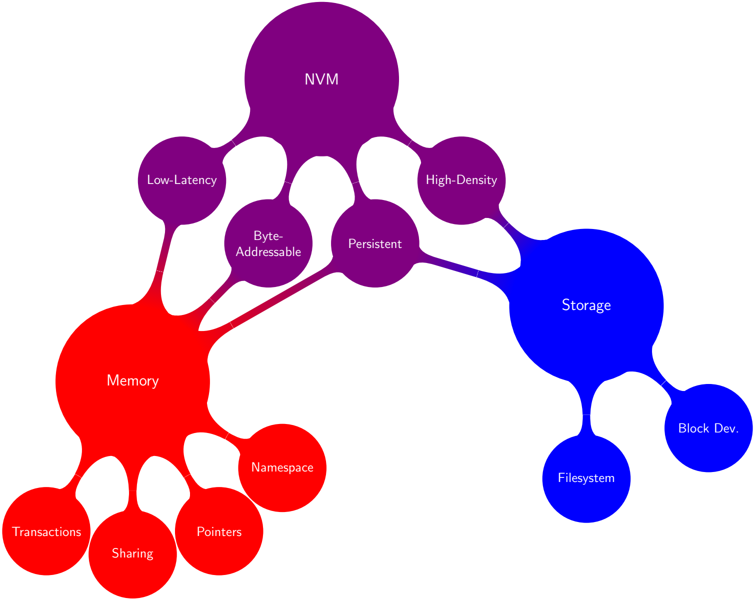
- Non-volatile
  - Memory survives power cycles
  - No need to restore from slow disks or flash
- High density
- Low latency
- Fine granularity updates
  - Operates on individual words
  - Access through `load` and `store` instructions

NVM Challenges

- Non-persistent caching
- Out-of-order flushes
  - write-back caches
- Torn writes
  - Updates bigger than 8 bytes are not atomic
- Complex interfaces
  - flushing cache lines, using memory fences, etc.



Approaches to use NVM

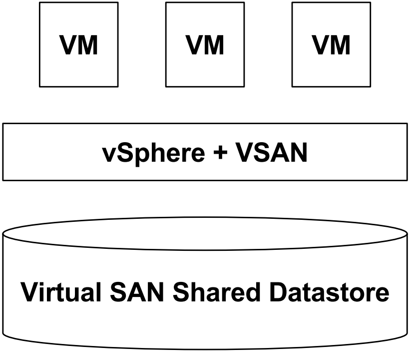


Application Specific Solution

We argue for consuming NVM through a transactional key-value store.

- Flexible
- Simple
- Performant

Case Study: VMware® Virtual San



METRADB: Specialized KV Store for VSAN

- Organizes objects in Containers
- Provides a flat namespace for Containers
- Provides transactional update containers
  - Only one active transaction per container
  - Transactions do not expand to multiple containers
- Provides KV-Store like interface

METRADB API

Operation	Description
$open(name, flags)$	open/create container, get handle
$remove(name)$	remove container
$close(h)$	close a handle
$put(h, k, buf, len)$	put key-value pair
$get(h, k, buf, len)$	get key-value pair
$delete(h, k)$	delete key-value pair
$commit(h)$	commit transaction
$abort(h)$	abort transaction

Transactions: How to do them?

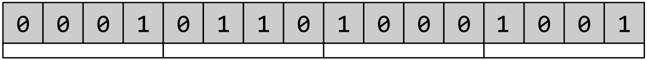
Undo Logging

- Update in-place
- Adds latency to critical path
- No easy way to batch and flush (poor cache locality)
- Data can be read from its original location
- Easy to implement

Redo Logging

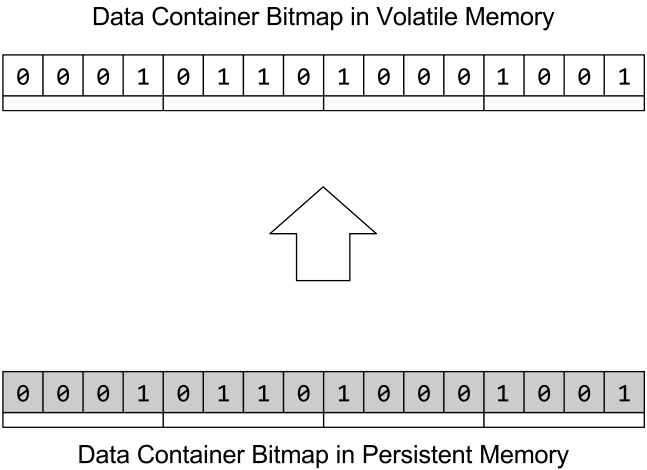
- Updates are buffered and applied at commit
- Batch flushes and sync (better cache locality)
- No latency added to the critical path
- Data may need to be read from the log
- Implementation is more complicated

Shadow Bitmaps: Handling Allocations

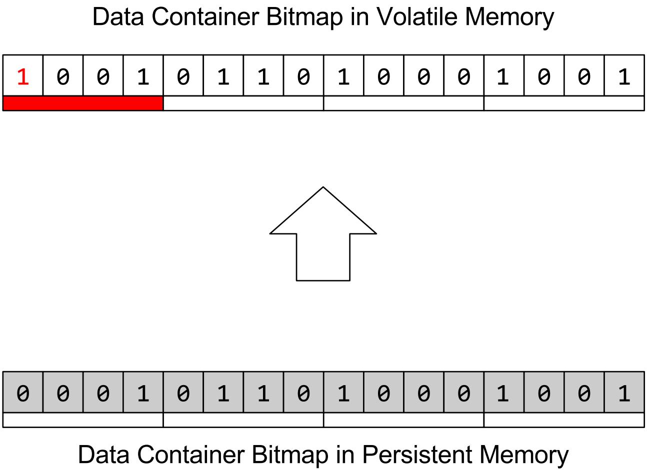


Data Container Bitmap in Persistent Memory

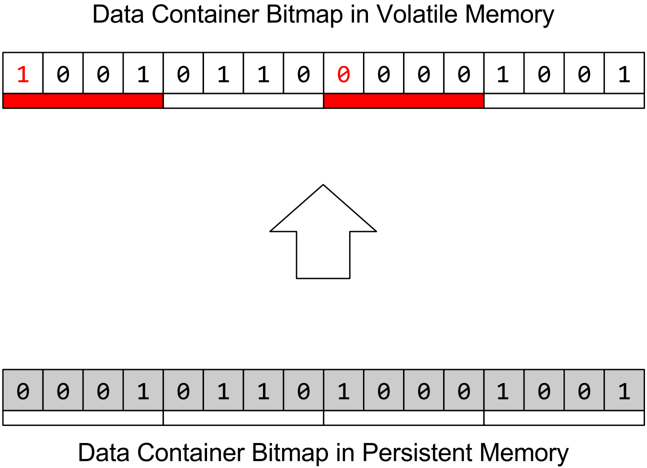
Shadow Bitmaps: Handling Allocations



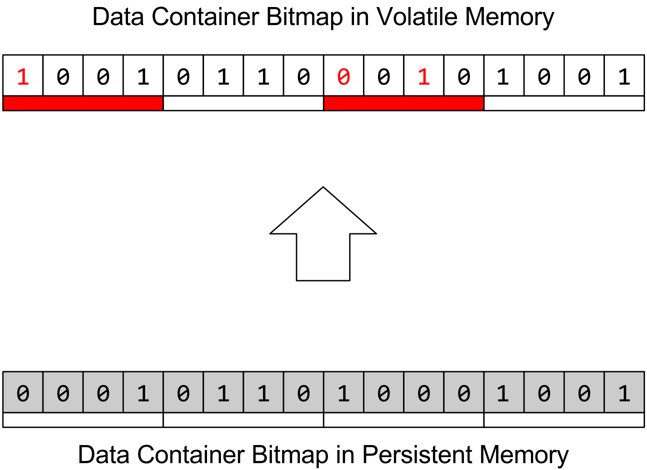
Shadow Bitmaps: Handling Allocations



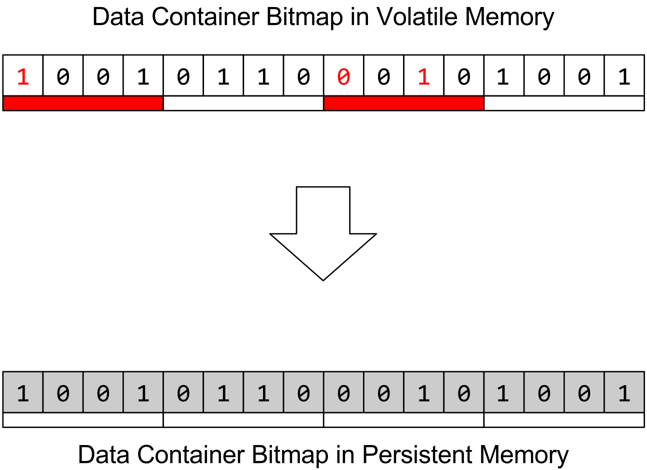
Shadow Bitmaps: Handling Allocations



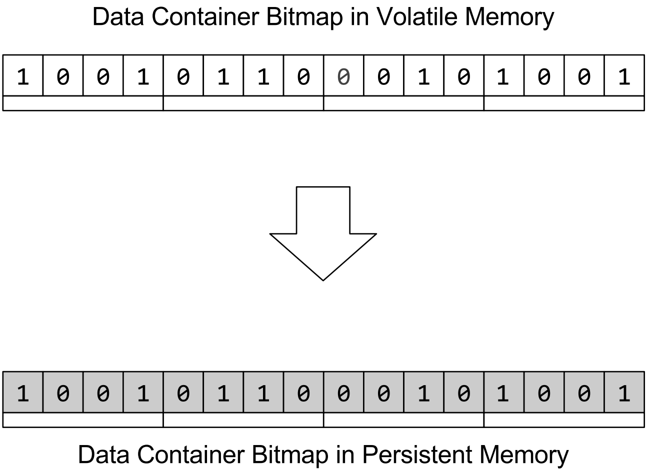
Shadow Bitmaps: Handling Allocations



Shadow Bitmaps: Handling Allocations



Shadow Bitmaps: Handling Allocations



Implementing Transactions

- Redo logging
- Out-of-place updates
- Shadow data structures
- Idempotent commits
  - Volatile metadata can be reconstructed from the logs
- Implicit start transaction
  - Move the state of the KV Store from one consistent state to the next

Indexing: Which data structure to use?

B+ Tree

- Higher latency for average operations
- Higher write amplification
- Predictable performance
- More difficult to implement
- Maintain key order

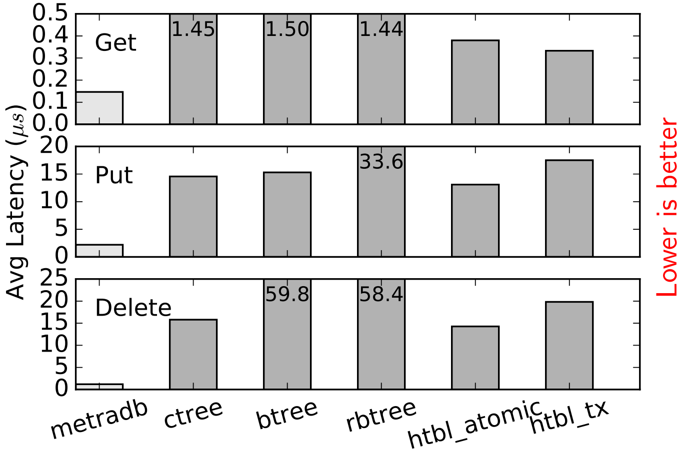
Hash Table

- Low latency for average operation
- Lower write amplification
- Less predictable performance
- Easy to implement
- Does not maintain key order

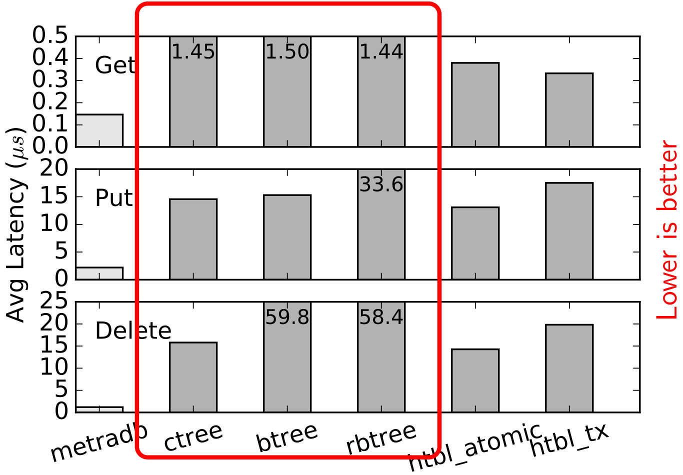
Experimental Setup

- METRADB is a user space library for GNU/Linux
- Linux Kernel v4.4
- 24 GB of RAM
- Intel XeonE5-2440 v2 1.90GHz CPU
  - 8 cores each with 2 hyper-threads
- NVM was simulated with memory mapped files
  - EXT4 with DAX support

Comparison with NVML

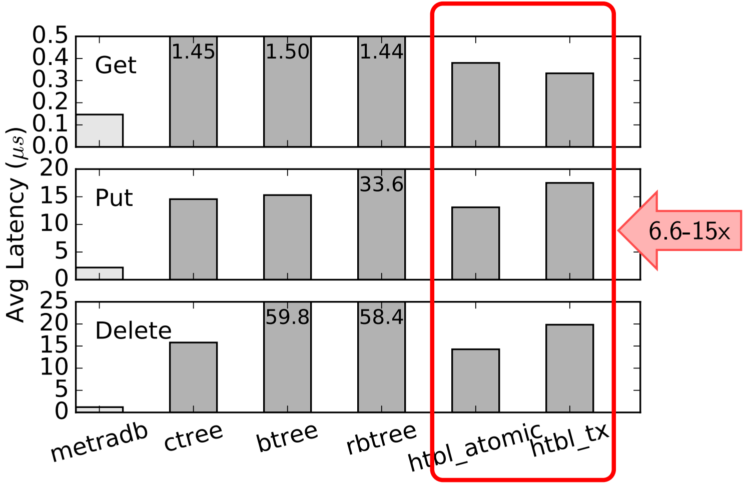


Comparison with NVML

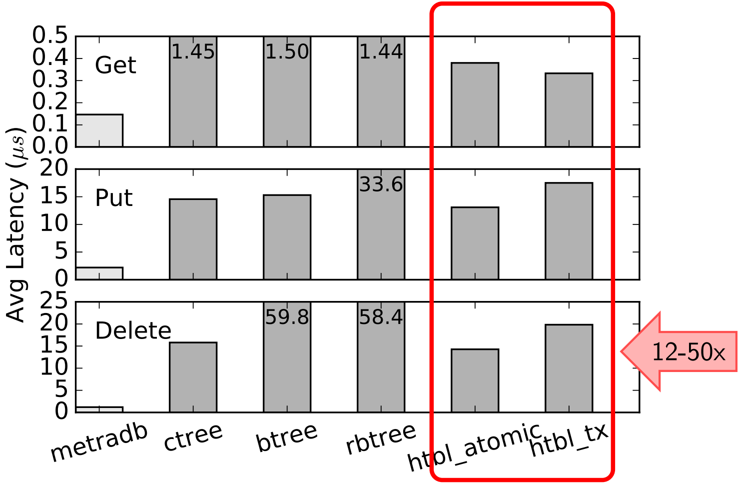


Comparison with NVML

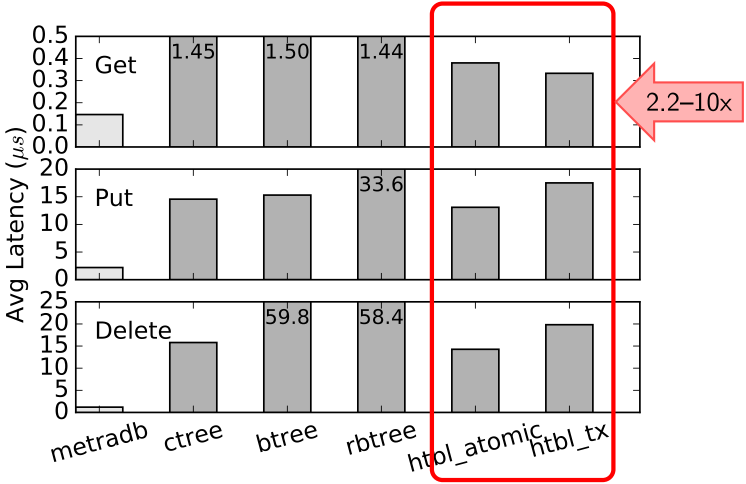
Comparison with NVML



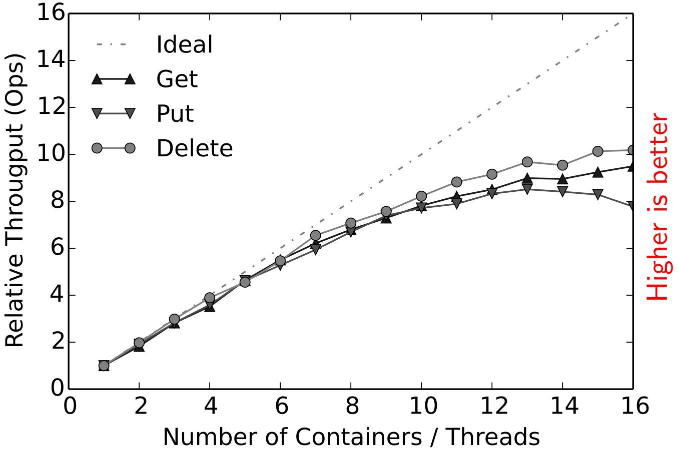
Comparison with NVML



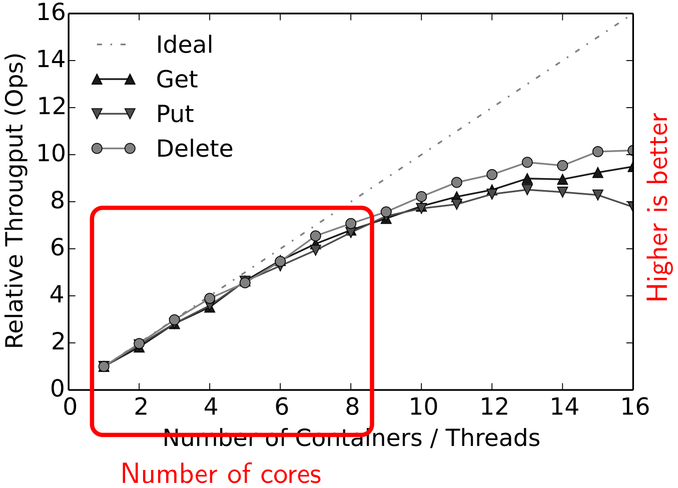
Comparison with NVML



Throughput Scalability of METRADB



Throughput Scalability of METRADB



Summary

- We propose application to consume NVM through a middle layer
  - For our application a key-value interface was sufficient
- This approach allows simplicity, easy adoptions of different NVM technologies, and fast development
  - About 2.3K LOC
- Because our solution was tailored to our application, we achieved higher performance than more general solutions

Thank you!

Leonardo Mármol <marmol@cs.fiu.edu>