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

NVM
- Low-Latency
- High-Density
- Byte-Addressable
- Persistent

Storage
- Block Dev.
- Filesystem

Memory
- Namespace
- Transactions
- Pointers
- Sharing

L. Már mol, J. Guerra, M. K. Aguilera (FIU and VMware)
Application Specific Solution

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

- Flexible
- Simple
- Performant
Case Study: VMware® Virtual San

![Diagram of VMware Virtual SAN with VMs and vSphere + VSAN]

Virtual SAN Shared Datastore
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>open(name, flags)</code></td>
<td>open/create container, get handle</td>
</tr>
<tr>
<td><code>remove(name)</code></td>
<td>remove container</td>
</tr>
<tr>
<td><code>close(h)</code></td>
<td>close a handle</td>
</tr>
<tr>
<td><code>put(h, k, buf, len)</code></td>
<td>put key-value pair</td>
</tr>
<tr>
<td><code>get(h, k, buf, len)</code></td>
<td>get key-value pair</td>
</tr>
<tr>
<td><code>delete(h, k)</code></td>
<td>delete key-value pair</td>
</tr>
<tr>
<td><code>commit(h)</code></td>
<td>commit transaction</td>
</tr>
<tr>
<td><code>abort(h)</code></td>
<td>abort transaction</td>
</tr>
</tbody>
</table>
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

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Data Container Bitmap in Persistent Memory
Shadow Bitmaps: Handling Allocations

Data Container Bitmap in Volatile Memory

0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1

Data Container Bitmap in Persistent Memory

0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1
Shadow Bitmaps: Handling Allocations

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

Data Container Bitmap in Volatile Memory

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

Data Container Bitmap in Persistent Memory
Shadow Bitmaps: Handling Allocations

Data Container Bitmap in Volatile Memory

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

Data Container Bitmap in Persistent Memory

\[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Shadow Bitmaps: Handling Allocations

Data Container Bitmap in Volatile Memory

| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

Data Container Bitmap in Persistent Memory

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
Shadow Bitmaps: Handling Allocations

Data Container Bitmap in Volatile Memory

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

Data Container Bitmap in Persistent Memory

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Shadow Bitmaps: Handling Allocations

Data Container Bitmap in Volatile Memory

```
1 0 0 1 0 1 1 0 0 0 1 0 1 0 0 1
```

Data Container Bitmap in Persistent Memory

```
1 0 0 1 0 1 1 0 0 0 1 0 1 0 0 1
```
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

- **Get**
  - metradb: 0.1
  - ctree: 1.50
  - btree: 1.44
  - rbtree: 1.45
  - htbl_atomic: 1.44
  - htbl_tx: 1.45

- **Put**
  - metradb: 33.6
  - ctree: 59.8
  - btree: 58.4
  - rbtree: 59.8
  - htbl_atomic: 58.4
  - htbl_tx: 59.8

Lower is better
Comparison with NVML

Lower is better

- **Get**
  - metradb: 1.45 µs
  - ctree: 1.50 µs
  - btree: 1.44 µs

- **Put**
  - metradb: 33.6 µs

- **Delete**
  - metradb: 59.8 µs
  - ctree: 58.4 µs

L. Mármol, J. Guerra, M. K. Aguilera (FIU and VMware)
Comparison with NVML

Get: 1.45 1.50 1.44

Put: 33.6

Delete: 59.8 58.4

Lower is better
Comparison with NVML

![Comparison with NVML](image)

L. Már mol, J. Guerra, M. K. Aguilera (FIU and VMware)
Comparison with NVML

![Comparison with NVML](image-url)

- **Get Latency**:
  - metradb: 1.45 µs
  - ctree: 1.50 µs
  - btree: 1.44 µs
  - rbtree: 1.44 µs
  - htbl_atomic: 1.44 µs
  - htbl_tx: 1.44 µs

- **Put Latency**: 33.6 µs

- **Delete Latency**: 59.8 µs

Lower is better

12-50x improvement

L. Mármol, J. Guerra, M. K. Aguilera (FIU and VMware)
Comparison with NVML

![Bar chart showing comparison between different KV-stores and NVML. The x-axis represents various KV-stores: metradb, ctree, btree, rbtree, htol_atomic, htbl_tx. The y-axis represents avg latency (µs). The chart highlights that NVML has 2.2–10x lower latency compared to the other KV-stores.]

Lower is better.
Throughput Scalability of METRADB

Higher is better

Number of cores

L. Mármol, J. Guerra, M. K. Aguilera (FIU and VMware)

NVM Through Customized KV-Stores
Throughput Scalability of METRADB

![Graph showing throughput scalability](image)

- Ideal
- Get
- Put
- Delete

Higher is better

Number of cores

Number of Containers / Threads

Relative Throughput (Ops)
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>