NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store

Leonardo Mármol‡*, Swaminathan Sundararaman†, Nisha Talagala†, Raju Rangaswami‡

† SanDisk ‡ Florida International University *work done at SanDisk
Introducing KV Stores

• Preferred data management solution for Internet services

• Provide simple interface
  – get, put, delete

• Provide weaker consistency model
  – Compared to RDMS
  – Tradeoff between performance and consistency
Limitations of existing solutions*

• Use log-structured/out-of-place updates
  – Better performance on hard-disk and older SSD’s
• Require compaction/garbage collection
• Creates auxiliary write amplification
  – Performance penalty
  – Reduces the life of NAND flash

*Existing Key-Value Store optimized for FLASH storage
How bad is the situation?

- Auxiliary write amplification varying from 2.5x to 43x
- “Don’t Stack your Log on my Log” by Yang et al. (INFLOW’14)
Existing KV Store Designs

App

KV Store

- Logging
- Data Mapping
- Compaction / GC

FTL
Existing KV Store Designs

App

KV Store
- Logging
- Data Mapping
- Compaction / GC

FTL
- Standard R/W Interface
- Logging
- Data Mapping
- Compaction / GC
- Others Subsystems
Existing KV Store Designs

**KV Store**
- Logging
- Data Mapping
- Compaction / GC

**FTL**
- Standard R/W Interface
  - Logging
  - Data Mapping
- Compaction / GC
- Others Subsystems
FTL Cooperative KV Store Design

- **App**
- **KV Store**
  - Hashing Scheme
- **FTL**
  - Extended Interface
  - Logging
  - Sparse Data Mapping
  - Compaction / GC
  - Others Subsystems
Design
NVMKV System Architecture

```
<table>
<thead>
<tr>
<th>User Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVMKV API</td>
</tr>
<tr>
<td>NVMKV Library</td>
</tr>
<tr>
<td>FTL Primitives</td>
</tr>
<tr>
<td>FTL</td>
</tr>
<tr>
<td>Flash</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
</tr>
<tr>
<td>Put</td>
</tr>
<tr>
<td>Delete</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

| Direct Access to Sparse Block Device |
| Read / Write Trim + Exists Atomic_write Iterate ... |
```

Apps $i$ and $j$ communicate with the NVMKV library, which in turn interacts with FTL primitives. FTL manages the Flash device, providing direct access to sparse block devices.
NVMKV API

• Management
  – Open / Create
  – Close / Destroy

• Basic operations
  – Get, Put, Delete

• Batch operations
  – Batch-Get, Batch-Put
  – Batch-Delete
  – Delete-all

• A full description of the API can be found in the paper

  “Beyond Block I/O: Rethinking Traditional Storage Primitives” by Ouyang et al.
I/O on a Sparse Block Device

- Exists(LBA) does not perform I/O
- New LBA(s) are computed using polynomial probing
- The size of the virtual slots can be configured
Multiple NVMKV On Single Device

- Pool: Set of unique key-value pairs that shared the same pool id.
  - Part of hashing scheme
    - LBA = Hash(key, pool_id)
  - No in-memory metadata
  - No extra I/O operations
  - Now operations required pool_id, in addition to key & value
Inserting KV-Pairs into Pools

LBA = Hash(key, pool_id)
Exists(LBA)
Write(<key, val>, LBA)

• Only one <key, val> per virtual slot
• Offset within the virtual slot determines the pool_id
• Offsets in virtual address does not creates wasted space in flash
Improving Performance by Caching

• **Read-only Cache**: Improves read performance.

• **Collision Cache**: Prevents multiple I/Os when cache collision occurs.

• For more details, please consult the paper.
Evaluation Results
NVMKV Performance

- NVMKV outperform LevelDB even when LevelDB uses async writes
- NVMKV writes are always synchronous (no buffering)
• LevelDB benefits from both its own cache and the file system cache (warmed)
• NVMKV perform competitively up to 32 pools even without any caching
YCSB Benchmark

![Graph showing throughput vs. read cache size for different benchmarks.]

- **Read Only**: NVMKV (64MB) vs. NVMKV (0MB) vs. LevelDB.
  - The graph shows a clear advantage for NVMKV (64MB) over NVMKV (0MB) and LevelDB.
  - The throughput for NVMKV (64MB) stabilizes at a higher level compared to the other two.

- **Update Heavy (50/50)**: NVMKV (64MB) vs. NVMKV (0MB) vs. LevelDB.
  - Similar to the Read Only case, NVMKV (64MB) shows a higher and more stable throughput.

The graphs illustrate the performance of NVMKV with and without memory, as well as LevelDB, under both read-only and update-heavy workloads.
Summary

• We propose a FTL cooperative design that allows for:
  – Simple KV-Store design and implementation
  – Constant amount of metadata
  – Multiple KV-Stores sharing the same FLASH
  – High performance / parallelism
  – Atomicity and durability of KV operations
  – Low write amplification
    • Reduced write amplification by up to 29x compared to LevelDB
Thank you!

opennvm.github.io