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

Leonardo Mármol‡, Swaminathan Sundararaman†, Nisha Talagala†, Raju Rangaswami‡, Sushma Devendrappa*, Bharath Ramsundar*, Sriram Ganesan*

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

• Preferred data management solution for Internet services
• Provide simple interface
  – get, put, delete
• Provide weaker consistency model
  – compared to RDMS
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
Auxiliary Write Amplification

- App
  - w
  - KV Store
    - Write application data
    - Update metadata
    - Data compaction / GC
  - aW
    - ...
How bad is the situation?

- Auxiliary write amplification varying from 2.5x to 43x
Existing KV Store Designs

App

KV Store
- Logging
- Data Mapping
- Compaction / GC

FTL

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

FTL

App

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
New Approach

• Cooperative design with FTL
  – Minimize auxiliary write amplification
  – Maximize application level performance
  – Leverage FTL for atomicity and durability
    • Using the extended interface, updates are done atomically
  – Use constant amount of metadata
    • Independent of the number of KV pairs
  – Provide close to raw device performance
  – Provide greater I/O parallelism
    • Implementation with fewer locks
Classes of Key-Value Stores

- **Disk Optimized**
  - BerkeleyDB
  - HashCache
  - MemcacheDB
  - MengoDB

- **SSD Optimized**
  - LevelDB
  - RockDB
  - SILT
  - FlashStore
  - FAWN-KV
  - Skimpy-Stash

- **FTL Optimized**
  - NVMKV
Design
Sparse Address Mapping

1. NVMKVM
   - get (key_x)
   - LBA_i = hash (key_i)
   - put (key_y, value)

2. FTL Sparse Address
   - fixed sized slot
   - Addr1
   - Addr2
   - Addr3

3. NAND Physical Address
   - key_x, value
   - key_y, value
   - key_z, value
## Extended FTL Interface

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exists</td>
<td>Query if an address is populated.</td>
</tr>
<tr>
<td>Atomic-Write</td>
<td>Write to an address range atomically.</td>
</tr>
<tr>
<td>Atomic-Trim</td>
<td>Delete an address range atomically.</td>
</tr>
<tr>
<td>Iterate</td>
<td>Return all populated addresses.</td>
</tr>
</tbody>
</table>

Many applications will benefit from having this extended interface available to them!
Hashing and Collision

```
Put(key, val)

i = 0

Failed

Yes

i > MAX

No

LBA = hash(key, i)

i += 1

Exists(LBA)

Yes

No

Atomic-Write(LBA, ... )
```
Hashing and Collision (Details)

• Keys are hashed into 48-bit addresses
  – Key Bit Range (KBR): maximum number of keys
  – Value Bit Range (VBR): maximum kv-pair size
  – KBR + VBR = 48 bits

• Collision are handle with Polynomial probing
  – Try 8 different locations before failing

• Case Study: {Device Size: 1TB, KBR: 36, VBR: 12}
  – 64 billion kv-pairs of 2MB of maximum size
  – Probability of put failing is $1/2^{40}$
Evaluation Results
Microbenchmark Results

- NVMKV scales better than LevelDB as the number of threads increases
- NVMKV outperforms LevelDB even at low thread counts and without FS cache
- NVMKV provides durability and atomicity of put operations
NVMKV outperforms LevelDB in every workload using 1/4 of the cache size.
LevelDB uses both its own cache and the file system cache.
Unlike LevelDB, NVMKV perform every update atomically and synchronous.
LevelDB Comparison using YCSB

- NVMKV outperforms LevelDB in both sync and async configurations
- NVMK provides a more stable and predictable performance
Auxiliary Write Amplification

- Input Data size: 1GB
- Input Data Size: 4GB

![Diagram showing write amplification comparison between different data sizes and operations.]
Conclusions
Summary

• We propose a FTL cooperative design that allows for:
  – Simple KV-Store design and implementation
  – Constant amount of metadata
  – High performance / parallelism
  – Atomicity and durability of KV operations
  – Low write amplification
Thank you!

opennvm.github.io