Towards Application Driven Storage

Optimizing RocksDB for Open-Channel SSDs

Javier González <javier@cnexlabs.com>

LinuxCon Europe 2015

Contributors: Matias Bjørling and Florin Petriuc
Application Driven Storage: What is it?

- RocksDB
- Oracle
- Ceph
- OpenStack

- App-specific opt.
- Standard Libraries
- Metadata Mgmt.
- FS-specific logic
- Page cache
- Block I/O interface
Application Driven Storage: What is it?

- **Application Driven Storage**
  - Avoid multiple (redundant) translation layers
  - Leverage optimization opportunities
  - Minimize overhead when manipulating persistent data
  - Make better decisions regarding latency, resource utilization, and data movement (compared to best-effort techniques today)

> Motivation: Give the tools to the applications that know how to manage their own storage
Application Driven Storage Today

  - Remove the OS kernel entirely from normal application execution

- **Samsung multi stream**
  - Let the SSD know from where “I/O streams” emerge to make better decisions

- **Fusion I/O**
  - Dedicated I/O stack to support a specific type of hardware

- **Open-Channel SSDs**
  - Expose SSD characteristics to the host and give it full control over its storage
Traditional Solid State Drive (SSD)

- Flash complexity is abstracted away from the host by an embedded Flash Translation Layer (FTL)
  - Maps logical addresses (LBAs) to physical addresses (PPAs)
  - Deals with flash constraints (next slide)
  - Has enabled adoption by making SSDs compliant with the existing I/O stack

High throughput + Low latency
Parallelism + Controller
Flash memory 101

• Flash constrains:
  - Write at a page granularity
  • Page state: Valid, invalid, erased
  - Write sequentially in a block
  - Write always to an erased page
    • Page becomes valid
  - Updates are written to a new page
    • Old pages become invalid - need for GC
  - Read at a page granularity (seq./random reads)
  - Erase at a block granularity (all pages in block)
    • Garbage collection (GC):
      • Move valid pages to new block
      • Erase valid and invalid pages -> erased state
Open-Channel SSDs: Overview

• Open Channel SSDs share control responsibilities with the Host in order to implement and maintain features that typical SSDs implemented strictly in the SSD device firmware

- Host-based FTL manages:
  - Data placement
  - I/O scheduling
  - Over-provisioning
  - Garbage collection
  - Wear-leveling

• Host needs to know:
  - SSD features & responsibilities
  - SSD geometry
    - NAND media idiosyncrasies
    - Die geometry (blocks & pages)
    - Channels, timings, etc.
    - Bad blocks & ECC

Physical flash exposed to the host (Read, Write, Erase)

Host manages physical flash

Application Driven Storage
Open-Channel SSDs: LightNVM

- **Targets**
  - Expose physical media to user-space

- **Block Managers**
  - Manage physical SSD characteristics
  - Evens out wear-leveling across all flash

- **Open-Channel SSD**
  - Responsibility
  - Offload engines
LightNVM’s DFlash: Application FTL

DFlash is the LightNVM target supporting application FTLs

```
struct vblock {
    uint64_t id;
    uint64_t owner_id;
    uint64_t nppas;
    uint64_t ppa_bitmap;
    sector_t bpa;
    uint32_t vlun_id;
    uint8_t flags;
};
```

Data placement
I/O scheduling
Over-provisioning
Garbage collection
Wear-leveling

Normal I/O
blockN->bppa * PAGE_SIZE

```
struct nvm_tgt_type tt_dflash = {
    ...
    .make_rq = df_make_rq,
    .end_io = df_end_io,
    ...
};
```
Open-Channel SSDs: Challenges

1. Which classes of applications would benefit most from being able to manage physical flash?
   - Modify storage backend (i.e., no posix)
   - Probably no file system, page cache, block I/O interface, etc.

2. Which changes do we need to make on these applications?
   - Make them work on Open-Channel SSDs
   - Optimize them to take advantage of directly using physical flash (e.g., data structures, file abstractions, algorithms).

3. Which interfaces would (i) make the transition simpler, and (ii) simultaneously cover different classes of applications?

⇒ New paradigm that we need to explore in the whole I/O stack
RocksDB: Overview

- Embedded Key-Value persistent store
- Based on Log-Structured Merge Tree
- Optimized for fast storage
- Server workloads
- Fork from LevelDB
- Open Source: https://github.com/facebook/rocksdb

- RocksDB is not:
  - Not distributed
  - No failover
  - Not highly available

RocksDB Reference: The Story of RocksDB, Dhruba Borthakur and Haobo Xu (link)
RocksDB: Overview
Problem: RocksDB Storage Backend

- **RocksDB LSM**
  - Storage backend decoupled from LSM
    - `WritableFile()`: Sequential writes -> Only way to write to secondary storage
    - `SequentialFile()` -> Sequential reads. Used primarily for sstable user data and recovery
    - `RandomAccessFile()` -> Random reads. Used primarily for metadata (e.g., CRC checks)

- **User Data**
- **DB Log**
- **Metadata**
  - `Current`
  - `LOCK`
  - `IDENTITY`
  - `MANIFEST`: File metadata
  - `LOCK`: `use_existing_db`
  - `CURRENT`: Superblock
  - `Info Log`: Log & Debugging
RocksDB: LSM using Physical Flash

- Objective: Fully optimize RocksDB for Flash memories
  - Control data placement:
    - User data in sstables is close in the physical media (same block, adjacent blocks)
    - Same for WAL and MANIFEST
  - Exploit Parallelism:
    - Define virtual blocks based on file write patterns in the storage backend
    - Get blocks from different LUNs based on RocksDB’s LSM write patterns
  - Schedule GC and minimize over-provisioning
    - Use LSM sstable merging strategies to minimize (and ideally remove) the need for GC and over-provisioning on the SSD
  - Control I/O scheduling
    - Prioritize I/Os based on the LSM persistent needs (e.g., L0 and WAL have higher priority than levels used for compacted data to maximize persistency in case of power loss)

➡ Implement an FTL optimized for RocksDB, which can be reused for similar applications (e.g., LevelDB, Cassandra, MongoDB)
RocksDB + DFlash: Challenges

• Sstables (persistent memtables)
  - **P1:** Fit block sizes in L0 and further level (merges + compactions)
    - No need for GC on SSD side - RocksDB merging as GC (less write and space amplification)
  - **P2:** Keep block metadata to reconstruct sstable in case of host crash

• WAL (Write-Ahead Log) and MANIFEST
  - **P3:** Fit block sizes (same as in sstables)
  - **P4:** Keep block metadata to reconstruct the log in case of host crash

• Other Metadata
  - **P5:** Keep superblock metadata and allow to recover the database
  - **P6:** Keep other metadata to account for flash constrains (e.g., partial pages, bad pages, bad blocks)

• Process
  - **P7:** Follow RocksDB architecture - upstreamable solution
P1, P3: Match flash block size

- WAL and MANIFEST are reused in future instances until replaced
  - **P3:** Ensure that WAL and MANIFEST replace size fills up most of last block

- Sstable sizes follow a heuristic - MemTable::ShouldFlushNow()

**P1:**
- kArenaBlockSize = sizeof(block)
- Conservative heuristic in terms of overallocation
  - Few lost pages is better than allocating a new block
- Flash block size becomes a “static” DB tuning parameter that is used to optimize “dynamic” ones

➡ Optimize RocksDB bottom up (from storage backend to LSM)
**P2, P4, P6: Block Metadata**

- Blocks can be checked for integrity
- New DB instance can append; padding is maintained in OOB (P6)
- Closing a block updates bad page & bad block information (P6)

```c
struct vblock_init_meta {
    char filename[100];  // RocksDB file GID
    uint64_t owner_id;   // Application owning the block
    size_t pos;          // relative position in block
};

struct vpage_meta {
    size_t valid_bytes;  // Valid bytes from offset 0
    uint8_t flags;       // State of the page
};

struct vblock_close_meta {
    size_t written_bytes; // Payload size
    size_t ppa_bitmap;    // Updated valid page bitmap
    size_t crc;           // CRC of the whole block
    unsigned long next_id; // Next block ID (0 if last)
    uint8_t flags;        // Vblock flags
};
```
A DFlash file can be reconstructed from individual blocks (P2, P4)

1. Metadata for the blocks forming a DFlash file is stored in MANIFEST
   - The last WAL is not guaranteed to reach the MANIFEST -> RECOVERY metadata for DFLASH
2. On recovery, LightNVM provides an application with all its valid blocks
3. Each block stores enough metadata to reconstruct a DFlash file
P5: Superblock

- CURRENT is used to store RocksDB “superblock”
  - Points to current MANIFEST, which is used to reconstruct the DB when creating a new instance. We append the block metadata that points to the blocks forming the current MANIFEST (P5)

Metadata Type:
- Log
- Current
- Metadata
- Sstable
- Private (Env)

Private (DFlash): vbblocks forming the DFlash File

Normal Recovery

Enough metadata to recover database in a new instance
P7: Work upstream
RocksDB + DFlash: Prototype (1/2)

• Optimize RocksDB for Flash storage
  - Implement a user space append-only FS that deals with flash constrains
    • Append-only: Updates are re-written and old data invalidated -> LSM understands this logic
    • Page cache implemented in user space; use Direct I/O
    • Only “sync” complete pages, and prefer closed blocks.
    • In case of write failure, write to a new block (or mark bad page and re-try)
  - Implement RocksDB’s file classes for DFlash:
    • WritableFile(): Sequential writes -> Only way to write to secondary storage
    • SequentialFile() -> Used primarily for sstable user data and recovery
    • RandomAccessFile() -> Used primarily for metadata (e.g., CRC checks)

• Use flash block as the central piece for storage optimizations
  - The Open-Channel SSD fabric is configured at first
    • Define block size - across luns and channels to exploit parallelism
    • Define different types of luns with different block features
  - RocksDB is configured with standard parameters (e.g., write buffer, cache)
    • DFlash backend tunes these parameters based on the type of lun and block
RocksDB + DFlash: Prototype (2/2)

• Use LSM merging strategies as perfect garbage collection (GC)
  - All blocks in a DFlash file are either active or inactive -> no need to GC in SSD
  - Reduce over-provisioning significantly (~5%)
  - Predictable latency -> SSD is in stable state from the beginning

• Reuse RocksDB concepts and abstractions as much as possible
  - Store private metadata in MANIFEST
  - Store superblob in CURRENT
  - Minimize the amount of “visible” metadata - use OOB, Root FS, etc.

• Separate persistent (meta)data between “fast” and “static”
  - Fast data is all user data (i.e., sstables) and the WAL
  - Fast metadata that follows user data rates (i.e., MANIFEST)
  - Static metadata is written once and seldom updated
    • CURRENT: Superblock for MANIFEST
    • LOCK and IDENTITY
Architecture: RocksDB with DFlash

LSM Logic
- Persist operations
- Environment Tuning
- Metadata, Close, Crash

DFlash File Classes
- DFWritableFile()
- DFRandomAccessFile()
- DFSequentialFile()

Env DFlash Optimizations
- Env Options (Flash charac. on init)

Env DFlash Private metadata

Open-Channel SSD (Fast Storage)
- User Data
  - sst
- DB Log
  - WAL
- Metadata
  - Manifest

Posix FS (Static Meta.)
- Metadata
  - CURRENT
  - IDENTITY
  - LOCK
  - LOG (Info)

DFlash Storage Backend

DFlash Storage Backend

User%Data
DB%Log
WAL
WAL
WAL
metadata
CURRENT
IDENTITY
LOCK
LOG
(Info)
Architecture: DFlash + LightNVM

RocksDB LSM

Env DFlash
- DFRandomAccessFile
- DFSquentialFile
- DFWritableFile
- DFlash File (blocks)

Env Options Optimizations

Init

I/O

get_block()
put_block()

Provisioning interface

Device Features

Block device

Sector Calculations

I/O Path

DFWritableFile

DFSequentialFile

DFRandomAccessFile

DFFlash File (blocks)

DFSequen?alFile

DFRandomAccesslFile

DFWritableFile

DFFlash

BM

Free Blocks
Used Blocks
Bad Blocks

I/O Path

File System

Traditional SSD
- Code
- CURRENT IDENTITY
- LOCK

LOG (Info)

Open-Channel SSD
- Data Placement controlled by LSM

sst
WAL
Manifest

PxWritableFile

PxSequentialFile

PxRandomAccesslFile

PxPosix

CURRENT
LOCK

IDENTITY

LOG (Info)

Env Posix

PxWritableFile

PxSequentialFile

PxRandomAccesslFile

PXPosix

Current Space

User Space

Kernel Space

USER SPACE

KERNEL SPACE
Architecture: DFlash + LightNVM

- **LSM is the FTL**
  - DFlash target and RocksDB storage backend take care of provisioning flash blocks

- **Optimized critical I/O path**
  - Sstables, WAL, and MANIFEST are stored in the Open-Channel SSD, where we can provide QoS

- **Enable a RocksDB distributed architecture**
  - BM abstracts the storage fabric (e.g., NVM) and can potentially provide blocks from different drives -> single address space
## QEMU Evaluation: Writes

<table>
<thead>
<tr>
<th>RocksDB make release</th>
<th>ENTRY KEYS with 4 threads</th>
<th>DFLASH (1 LUN)</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000 keys</td>
<td>70MB/s</td>
<td>25MB/s</td>
<td></td>
</tr>
<tr>
<td>100000 keys</td>
<td>40MB/s</td>
<td>25MB/s</td>
<td></td>
</tr>
<tr>
<td>1000000 keys</td>
<td>25MB/s</td>
<td>20MB/s</td>
<td></td>
</tr>
</tbody>
</table>

- DFlash write page cache + Direct I/O + flash page aligned write buffer is better optimized than best effort techniques and top-down optimizations (RocksDB parameters). Write buffer required by RocksDB due to small WAL writes.

- If we manually tune buffer sizes with Posix, we obtain similar results. However, it requires lots of experimentation for each configuration.

Bare Metal: ~180MB/s

Page-aligned write buffer
## QEMU Evaluation: Reads

<table>
<thead>
<tr>
<th>RocksDB make release</th>
<th>ENTRY KEYS</th>
<th>DFLASH (1 LUN)</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>10000 keys</td>
<td>5MB/s</td>
<td>300MB/s</td>
</tr>
<tr>
<td></td>
<td>100000 keys</td>
<td>5MB/s</td>
<td>500MB/s</td>
</tr>
<tr>
<td></td>
<td>1000000 keys</td>
<td>5MB/s</td>
<td>570MB/s</td>
</tr>
</tbody>
</table>

- Without a DFLASH page cache we need to issue an I/O for each read!
  - Sequential 20 byte-reads in same page would issue different PAGE_SIZE I/Os
### QEMU Evaluation: “Fixing” Reads

<table>
<thead>
<tr>
<th>RocksDB make release</th>
<th>ENTRY KEYS</th>
<th>DFLASH (1 LUN)</th>
<th>DFLASH (1 LUN) (+ simple page cache)</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000 keys</td>
<td>5MB/s</td>
<td>160MB/s</td>
<td>300MB/s</td>
</tr>
<tr>
<td></td>
<td>100000 keys</td>
<td>5MB/s</td>
<td>280MB/s</td>
<td>500MB/s</td>
</tr>
<tr>
<td></td>
<td>1000000 keys</td>
<td>5MB/s</td>
<td>300MB/s</td>
<td>570MB/s</td>
</tr>
</tbody>
</table>

- Posix + buffered I/O using Linux’s page cache is still better, but we have confirmed our hypothesis.

> **User-space page cache is a necessary optimization when the generic OS page cache is on the way.** Other databases use this technique (e.g., Oracle, MySQL)
QEMU Evaluation: Insights

• Posix backend and DFlash backend (with 1 lun) should achieve very similar throughput for reads/writes when using same page cache and write buffer optimizations

• But...
  - DFlash allows to optimize buffer and cache sizes based on Flash characteristics
  - DFlash knows which file class is calling - we can do prefetching for sequential reads (DFSequentialFile) at block granularity
  - DFlash designed to implement a Flash optimized page cache using Direct I/O

• If the Open-Channel SSD exposes several LUNs, we can exploit parallelism within DFlash and RocksDB’s LSM write/read patterns
  - How many luns and their characteristics are organized is controller specific
CNEX WestLake SDK: Overview

FPGA Prototype Platform before ASIC:
- PCIe G3x4 or PCI G2x8
- 4x10GE NVMe
- 40 bit DDR3
- 16 CH NAND

Read/Write Performance (MB/s)
## CNEX WestLake Evaluation

<table>
<thead>
<tr>
<th>RocksDB make release</th>
<th>ENTRY KEYS with 4 threads</th>
<th>WRITES (1 LUN)</th>
<th>READS (1 LUN)</th>
<th>WRITES (8 LUNS)</th>
<th>READS (8 LUNS)</th>
<th>WRITES (64 LUNS)</th>
<th>READS (64 LUNS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RocksDB DFLASH</td>
<td>10000 keys</td>
<td>21MB/s</td>
<td>40MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>100000 keys</td>
<td>21MB/s</td>
<td>40MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>1000000 keys</td>
<td>21MB/s</td>
<td>40MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Raw DFLASH (with fio)</td>
<td></td>
<td>32MB/s</td>
<td>64MB/s</td>
<td>190MB/s</td>
<td>180MB/s</td>
<td>920MB/s</td>
<td>1,3GB/s</td>
</tr>
</tbody>
</table>

- We focus on a single I/O stream for first prototype -> 1 LUN
CNEX WestLake Evaluation: Insights

- RocksDB checks for sstable integrity on writes (intermittent reads)
  - We pay the price of not having an optimized page cache also on writes
  - Reads and writes are mixed in one single lun

- Ongoing work: Exploit parallelism in RocksDB’s I/O patterns
  - Do not mix R/W
  - Different VLUN per path
  - Different VLUN types
  - Enabling I/O scheduling
  - Block pool in DFlash (prefetching)
• Also, in *any* Open-Channel SSD
  - DFlash will not get a performance hit when the SSD triggers GC - RocksDB does GC when merging sstables in LSM > L0
    • SSD steady state is improved (and reached from the beginning)
    • We achieve predictable latency
Status and ongoing work

• Status:
  - get_block/put_block interface (first iteration) through the DFlash target
  - RocksDB DFlash target that plugs into LightNVM. Source code to test in available (RocksDB, Kernel, and QEMU). Working on WestLake upstreaming

• Ongoing:
  - Implement functions to increase the synergy between the LSM and the storage backend (i.e., tune write buffer based on block size) -> Upstreaming
  - Support libaio to enable async I/O in DFlash storage backend
    • Need to deal with RocksDB design decisions (e.g., Get() assumes sync IO)
  - Exploit device parallelism within RockDB internal structures
  - Define different types of virtual luns and expose them to the application
  - Other optimizations: double buffering, aligned memory in LSM, etc.
  - Move RocksDB DFlash’s logic to liblightnvm -> append-only FS for Flash
Conclusions

• Application Driven Storage
  - Demo working on real hardware:
    • (RocksDB -> LightNVM -> WestLake-powered SSD)
  - QEMU support for testing and development
  - More Open-Channel SSDs coming soon.

• RocksDB
  - DFlash dedicated backend -> append-only FS optimized for Flash
  - Set the basis for moving to a distributed architecture while guaranteeing performance constrains (specially in terms of latency)
  - Intention to upstream the whole DFlash storage backend

• LightNVM
  - Framework to support Open-Channel SSDs in Linux
  - DFlash target to support application FTLs
Towards Application Driven Storage
Optimizing RocksDB on Open-Channel SSDs with LightNVM

- Open-Channel SSD Project: https://github.com/OpenChannelSSD
- LightNVM: https://github.com/OpenChannelSSD/linux
- RocksDB: https://github.com/OpenChannelSSD/rocksdb

Javier González <javier@cnexlabs.com>
LinuxCon Europe 2015