RocksDB on Open-Channel SSDs

Javier González <javier@cnexlabs.com>
RocksDB Annual Meetup'15 - Facebook
Solid State Drives (SSDs)

- High throughput + Low latency
- Parallelism + Controller
Embedded Flash Translation Layers (FTLs)

- Have enabled wide adoption by exposing a block I/O interface to the application
- However, they are a roadblock for I/O intensive workloads
  - Hardwire design decisions about data placement, over-provisioning, scheduling, garbage collection, and wear leveling -> Assumptions on application workload
  - Introduces redundancies, missed optimizations, and underutilization of resources
  - Predictable latencies cannot be guaranteed – 99 percentiles
  - Introduces unavoidable write-amplification on the device side (GC)

I/O Applications use expensive data structures and employ host resources to align their I/O patterns to (unreachable) flash constraints (e.g., LSM tree)
Open-Channel SSDs share control responsibilities with the Host in order to implement and maintain features that typical SSDs implement strictly in the device firmware.
Why Open-Channel SSDs

- Optimize high-performance I/O applications for fast Flash memories
  - Control data placement - use physical flash blocks directly
  - Remove device garbage collection (GC) - reuse LSM compaction on the host
    - Achieve predictable latency - no 99 percentiles (measured in seconds)
    - Avoid write-amplification introduced by the FTL - control device endurance with host software
    - Improve the steady state of the device (and start from it)
  - Minimize over-provisioning (normal SSDs employ 10-30% over-provisioning for performance)
RocksDB: Using Open-Channel SSDs

struct vblock {
    uint64_t id;
    uint64_t owner_id;
    uint64_t npas;
    uint64_t ppa_bitmap;
    sector_t ppa;
    uint32_t vlun_id;
    uint8_t flags;
};

struct nvm_tgt_type tt_dflash = {
    .make_rq = df_make_rq,
    .end_io = df_end_io,
};

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

Application

FTL

Provisioning buffer

Block device

Provisioning interface

Normal I/O

libaio

posixaio

...
RocksDB: Using Open-Channel SSDs

• Sstables
  - **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:** Upstream to vanilla RocksDB
RocksDB: Data Placement

- 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 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)
RocksDB: Crash Recovery

- 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)

- A “file” can be reconstructed from individual blocks (P2, P4, P5 P6)
- 1. Metadata for the blocks forming a file is stored in MANIFEST
- 2. On recovery, LightNVM provides an application with all its valid blocks
- 3. Each block stores enough metadata to reconstruct a file
Work Upstream
Architectural Overview:

**RocksDB LSM**
- LSM is the FTL
  - Tree nodes (files) control data placement on physical flash
  - Sstables, WAL, and MANIFEST on Open-Channel SSD - rest in FS
  - Garbage collection takes place during LSM compaction

**LightNVM manages flash**
- liblightnvm takes care of flash block provisioning; Env DFlash works directly with PPAs
- Media manager handles wear-levelling (get/put block)
Architecture: Optimizing RocksDB

- RocksDB is in full control:
  - Do not mix R/W
  - Different VLUN per IO path
  - Different VLUN types
  - Enabling I/O scheduling
  - Block pool in DFlash (prefetching)
FPGA Prototype Platform before ASIC:
- PCIe G3x4 or PCI G2x8
- 4x10GE NVMoE
- 40 bit DDR3
- 16 CH NAND

* Not real performance results - FPGA prototype, not ASIC
**Evaluation: RocksDB on CNEX WestLake FPGA SDK**

<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 (2 LUNS)</th>
<th>READS (2 LUNS)</th>
<th>WRITES (64 LUNS)</th>
<th>READS (64 LUNS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RocksDB DFLASH</strong></td>
<td>10000 keys</td>
<td>23MB/s</td>
<td>40MB/s</td>
<td>35MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>100000 keys</td>
<td>26MB/s</td>
<td>40MB/s</td>
<td>33MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>1000000 keys</td>
<td>25MB/s</td>
<td>40MB/s</td>
<td>32MB/s</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Raw DFLASH (with fio)</strong></td>
<td></td>
<td>32MB/s</td>
<td>64MB/s</td>
<td>64MB/s</td>
<td>128MB/s</td>
<td>920MB/s</td>
<td>1,3GB/s</td>
</tr>
</tbody>
</table>
Status & Ongoing work

• Status:
  - LightNVM has been accepted in the Linux Kernel
  - Working prototype of RocksDB with DFlash storage backend on LightNVM
  - Implemented liblightnvm append-only support + IO interface (first iteration)

• Ongoing:
  - Performance testing and tuning on Westlake ASIC
  - Move RocksDB DFlash storage backend logic to liblightnvm
  - Improve I/O submission
    • Submit I/Os directly to the media manager
    • Support libaio to enable async I/O through the traditional stack
  - Exploit device parallelism within RocksDB’s LSM
Invvm – Open-Channel SSDs administration
https://github.com/OpenChannelSSD/lightnvm-adm

liblightnvm – LightNVM application support
https://github.com/OpenChannelSSD/liblightnvm

Test tools
https://github.com/OpenChannelSSD/lightnvm-hw

QEMU NVMe with Open-Channel SSD Support
https://github.com/OpenChannelSSD/qemu-nvme
RocksDB on Open-Channel SSDs

Open-Channel SSD Project: https://github.com/OpenChannelSSD
LightNVM: https://github.com/OpenChannelSSD/linux
liblightnvm: https://github.com/OpenChannelSSD/liblightnvm
RocksDB: https://github.com/OpenChannelSSD/rocksdb
Interface Specification: http://goo.gl/BYTjLI
Documentation: http://openchannelssd.readthedocs.org

Javier González <javier@cnexlabs.com>
RocksDB Annual Meetup'15 - Facebook