Application-Driven Flash Translation Layers on Open-Channel SSDs

Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, Yiren Ronnie Huang

NVMW'16

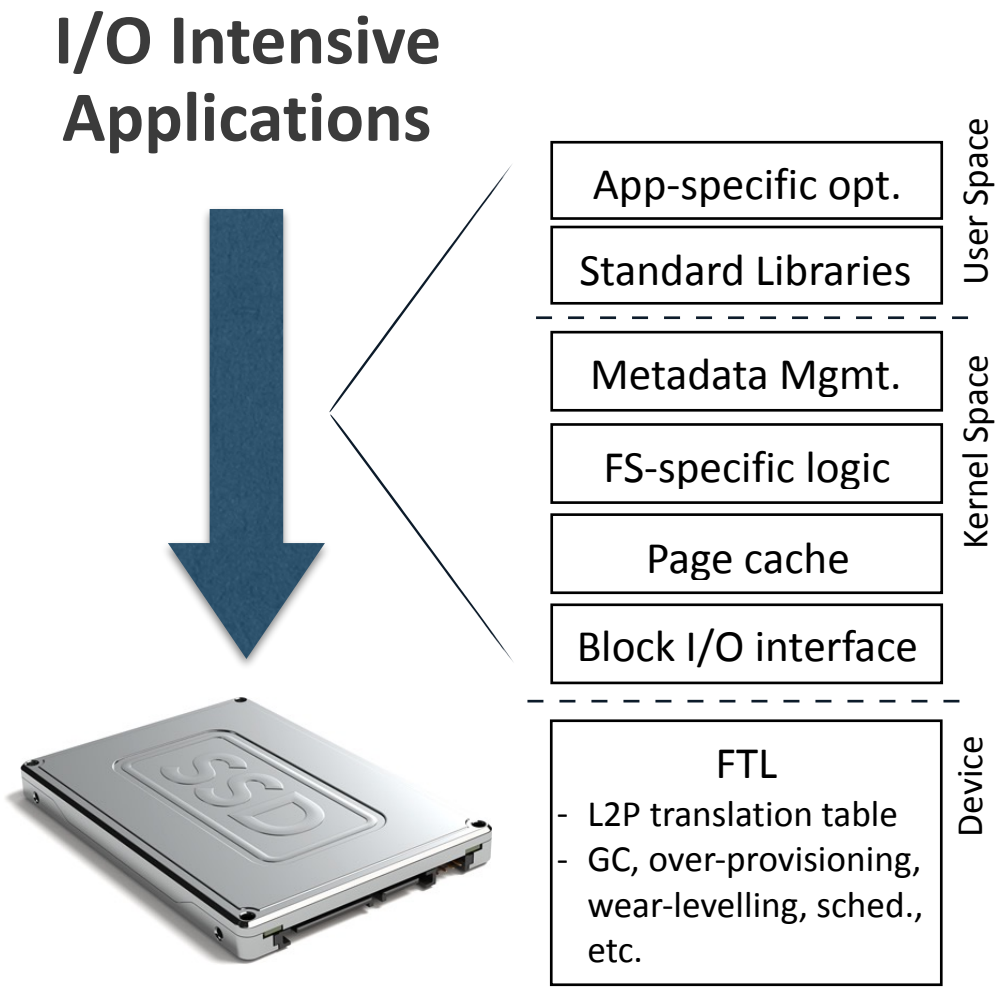
CNEXLABS

Solid State Drives (SSDs)

High throughput + Low latency

Parallelism + Controller

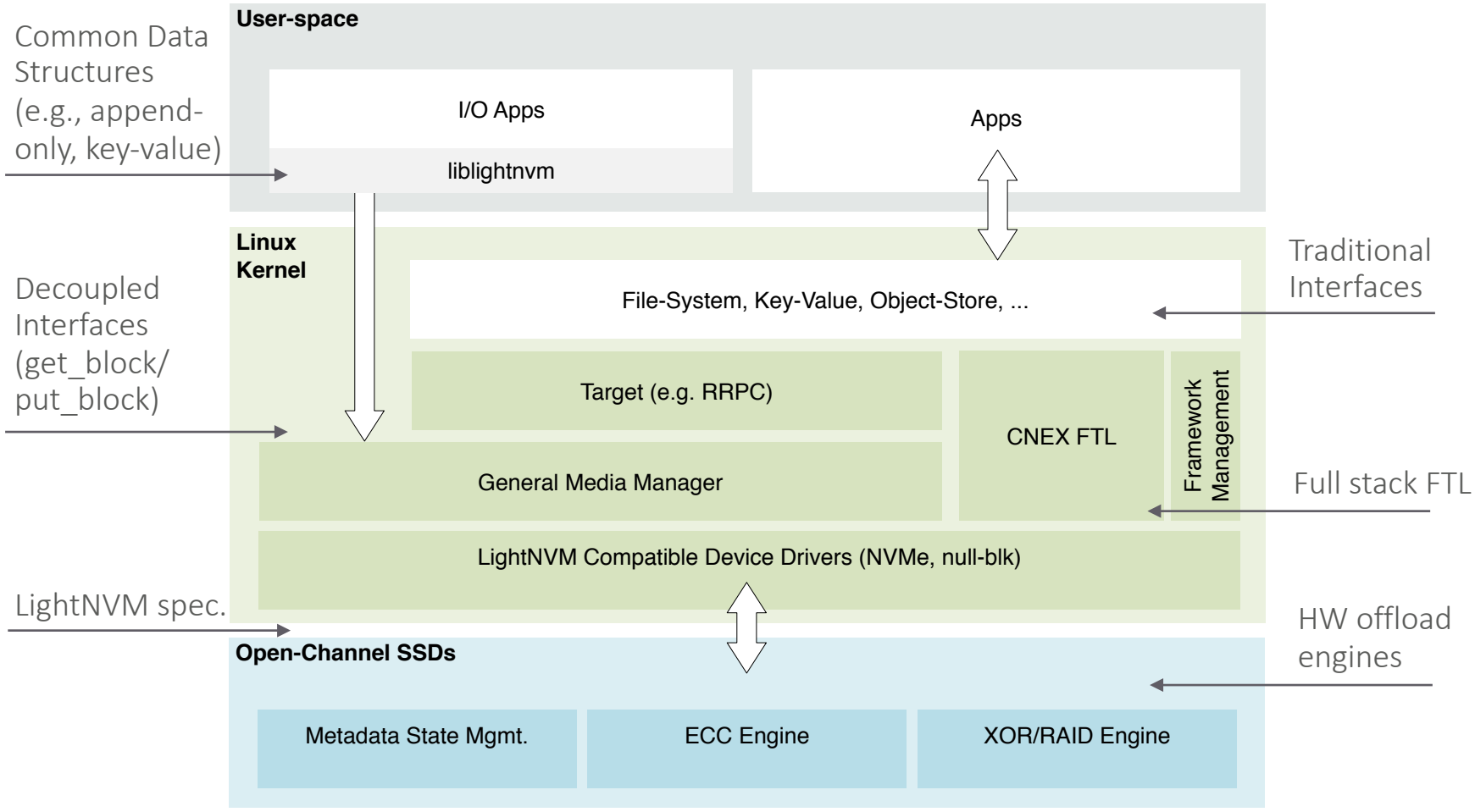
Embedded Flash Translation Layers (FTLs)



- 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 ressources to align their I/O patterns to (unreachable) flash constrains (e.g., LSM tree)

Open-Channel SSDs

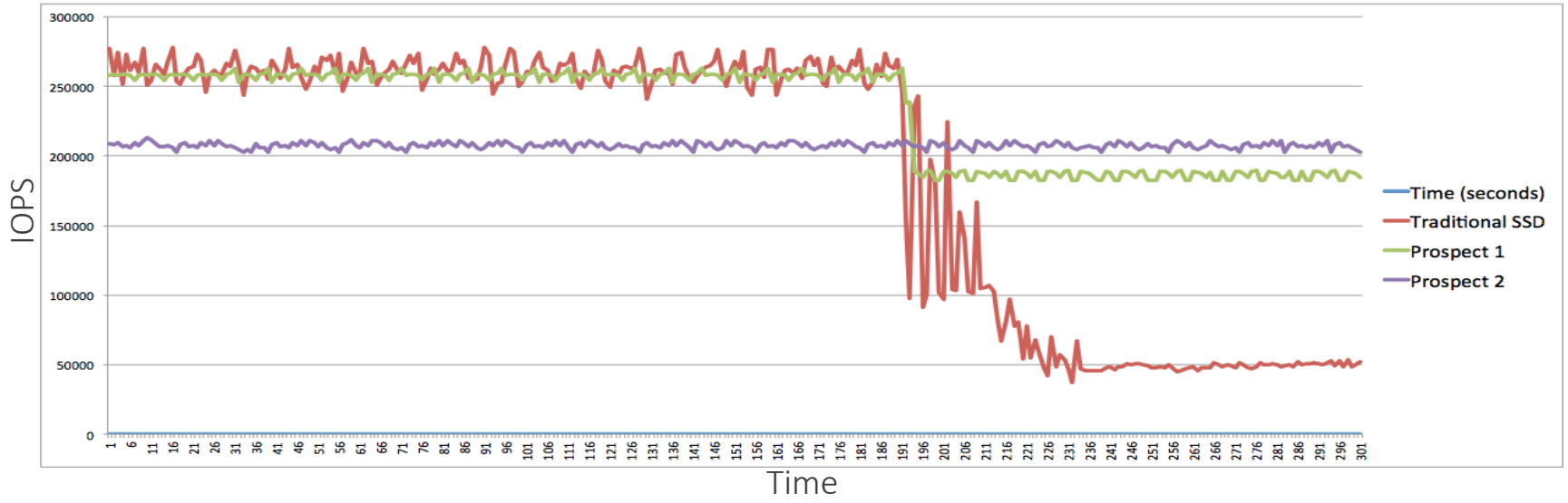


- Host Manages
  - Data Placement
  - Garbage Collection
  - I/O Scheduling
  - Over-provisioning
  - Wear-levelling
- Device Information
  - SSD offload engines & responsibilities
  - SSD geometry:
    - NAND media geometry
    - Channels, timings, etc.
    - Bad blocks
    - PPA Format

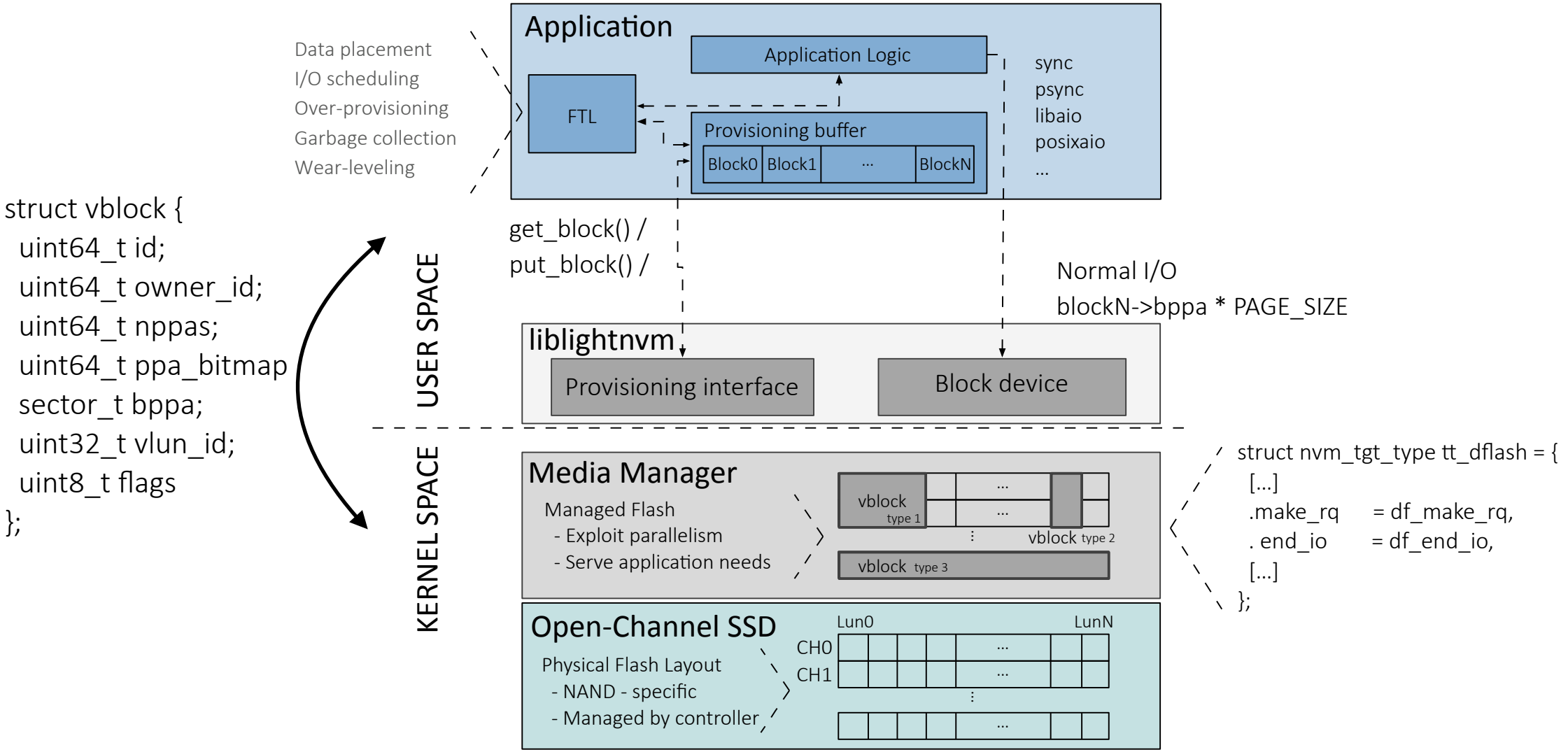
Open-Channel SSDs share control responsibilities with the Host in order to implement and maintain features that typical SSDs implement strictly in the device firmware

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)



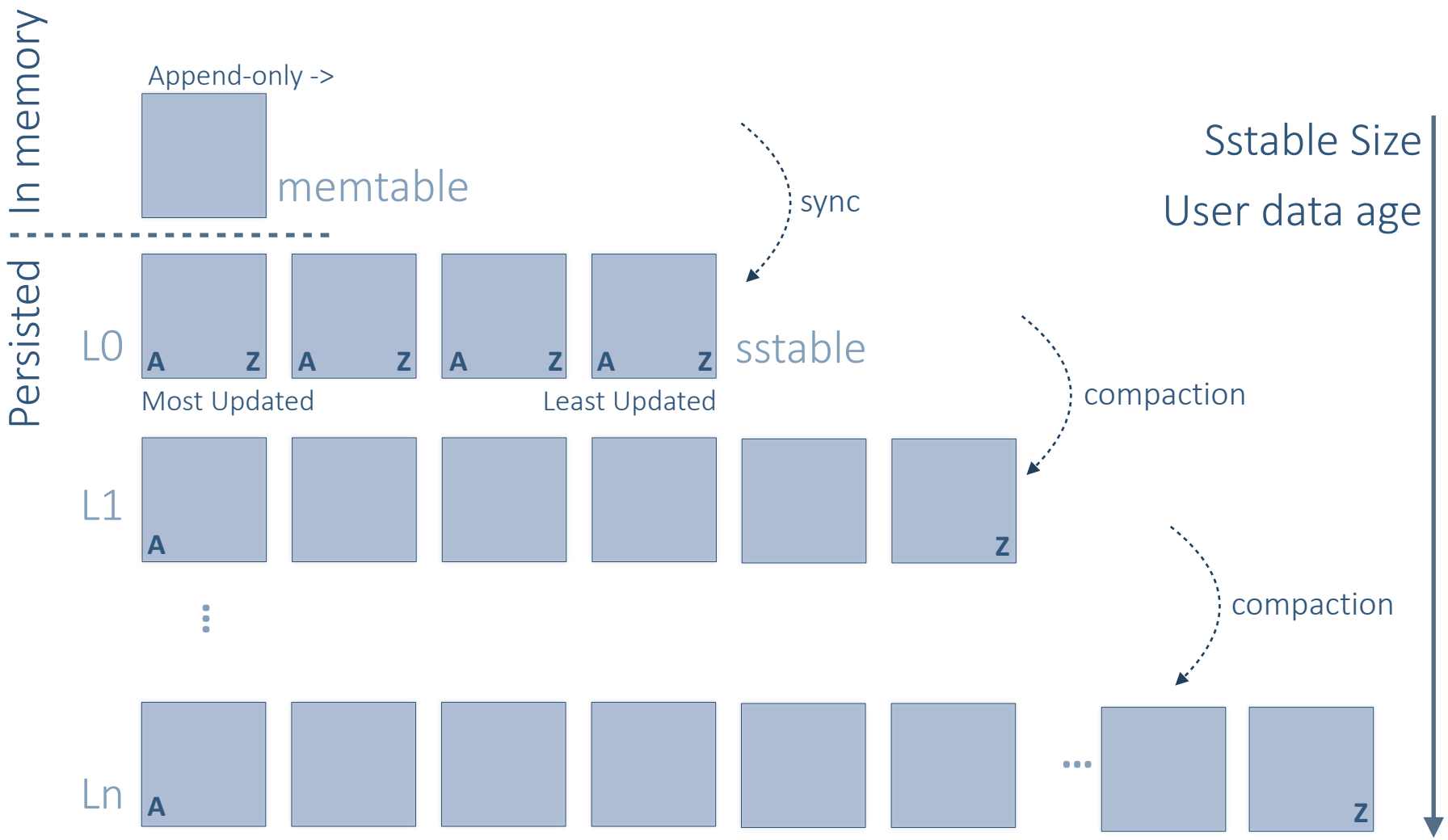
Application-driven FTLs




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

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

RocksDB: Log-Structured Merge Tree

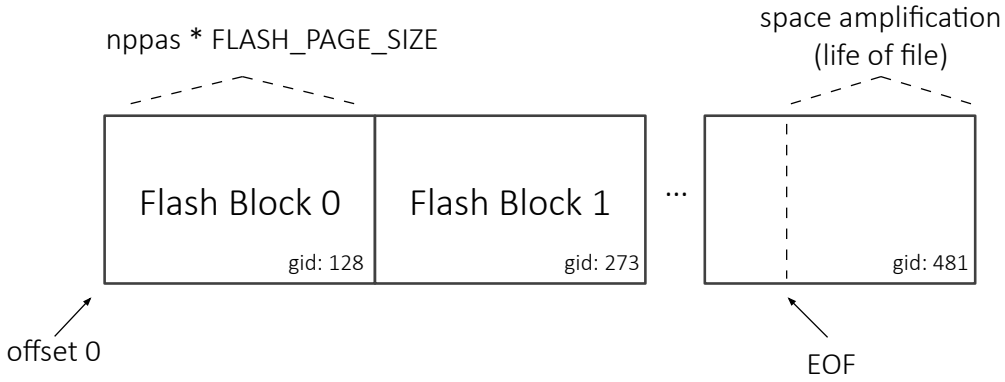


RocksDB: Using Open-Channel SSDs

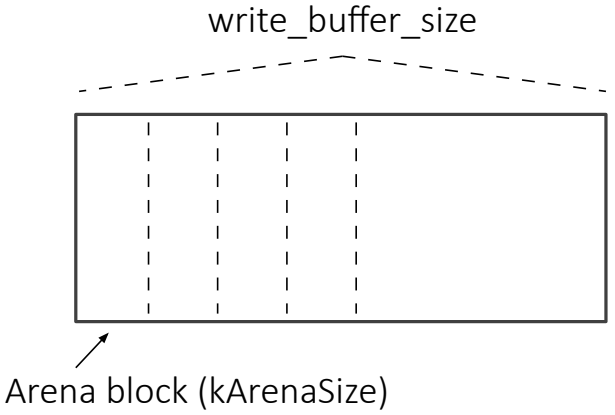
- Sstables
  - P1: Fit block sizes in L0 and further level (merges + compactions)
    - No need for GC on SSD side - RocksDB merging as GC (less write and space amplification)
  - P2: Keep block metadata to reconstruct sstable in case of host crash
- WAL (Write-Ahead Log) and MANIFEST
  - P3: Fit block 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()

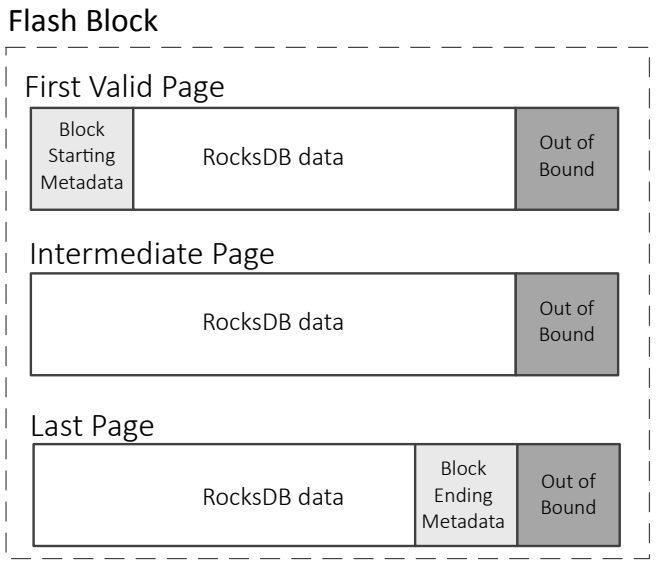


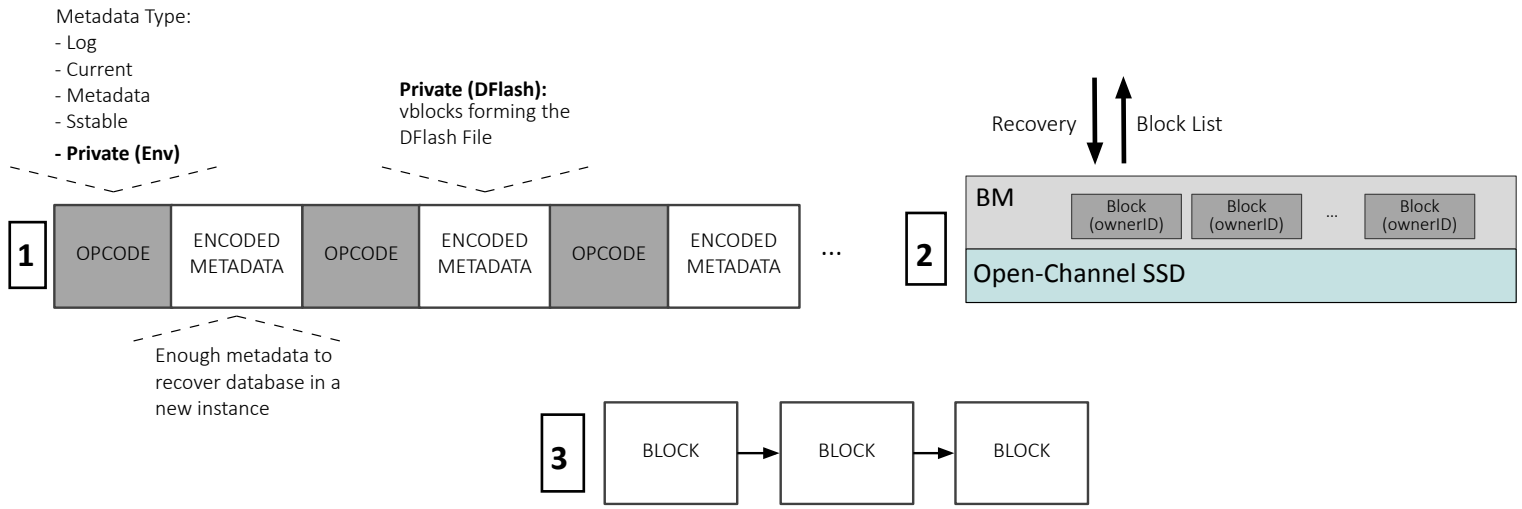
P1:

- 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



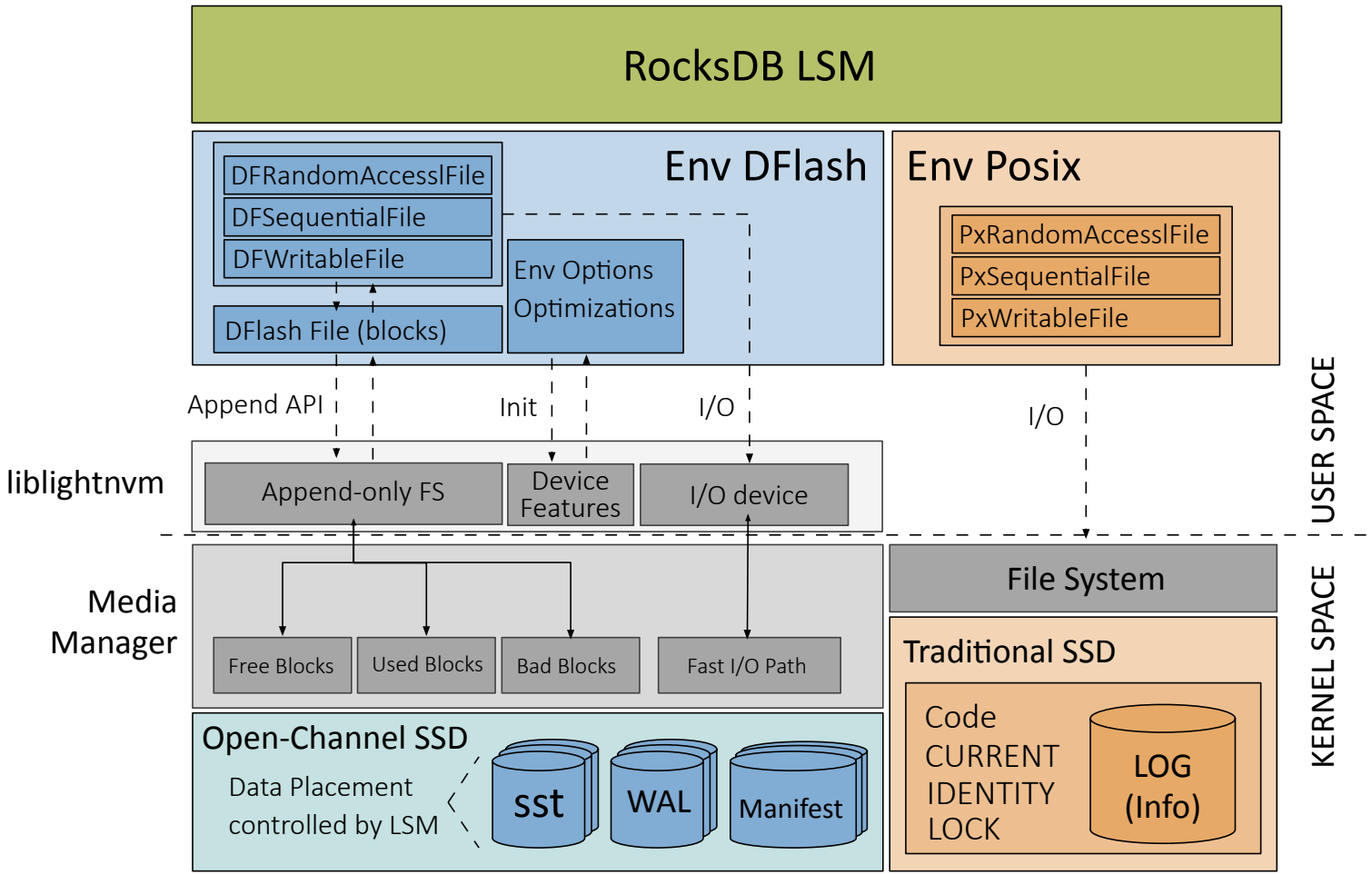


- Blocks can be checked for integrity
- New DB instance can append; padding is maintained in OOB (**P6**)
- Closing a block updates bad page & bad block information (**P6**)

- 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

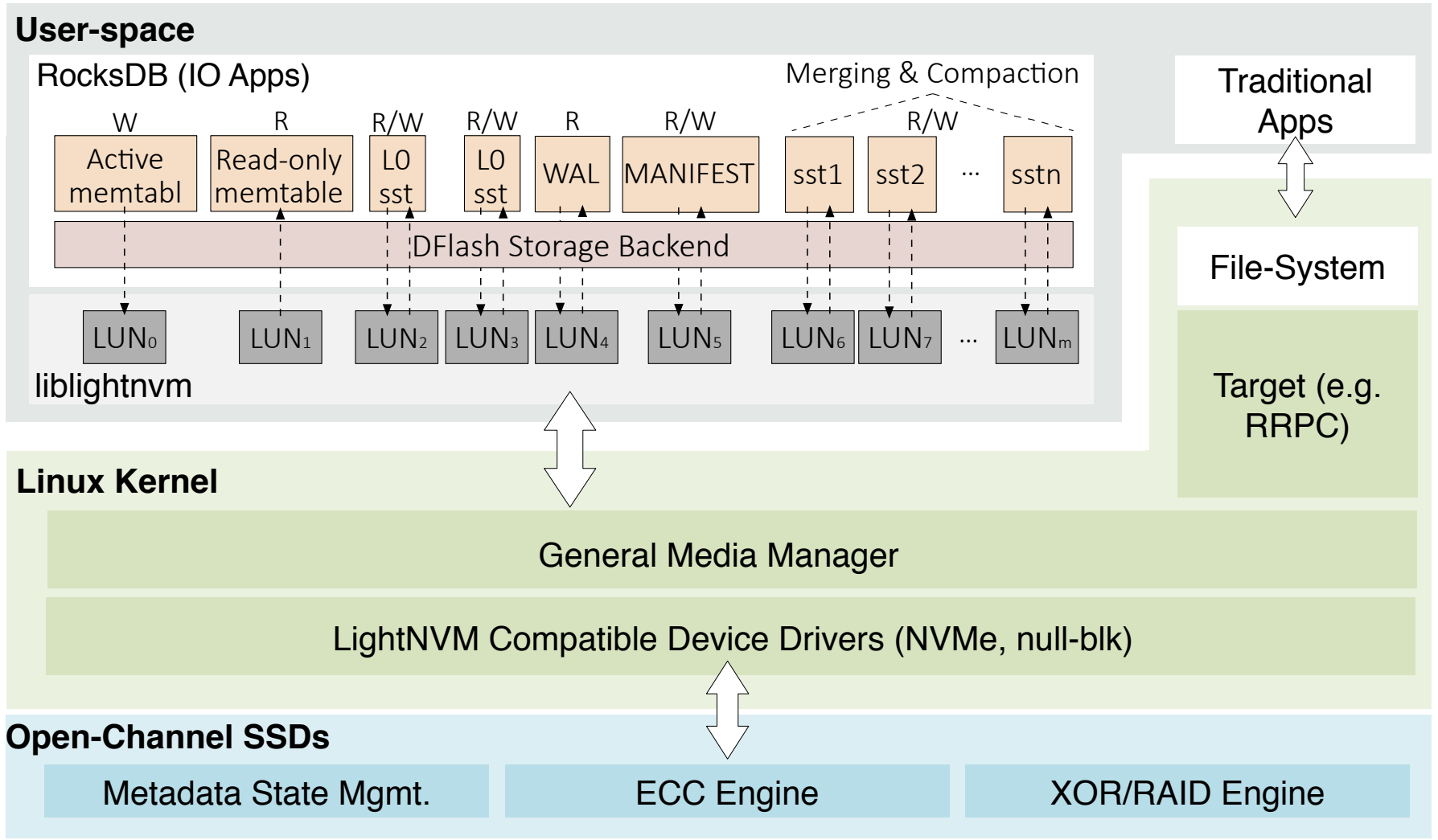
Work Upstream

Architecture: RocksDB + liblightnvm



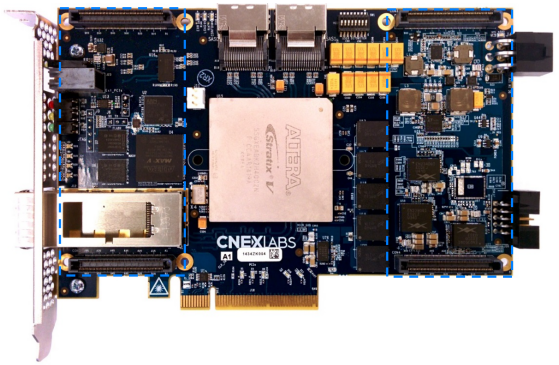
- LSM is the FTL
  - Tree nodes (files) control data placement on physical flash
  - Sstables, WAL, and MANIFEST on Open-Channel SSD - rest in FS
  - Garbage collection takes place during LSM compaction
- LightNVM manages flash
  - liblightnvm takes care of flash block provisioning; Env DFlash works 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)

Evaluation: CNEX WestLake FPGA SDK overview



FPGA Prototype Platform before ASIC:

- PCIe G3x4 or PCI G2x8
- 4x10GE NVMoE
- 40 bit DDR3
- 16 CH NAND

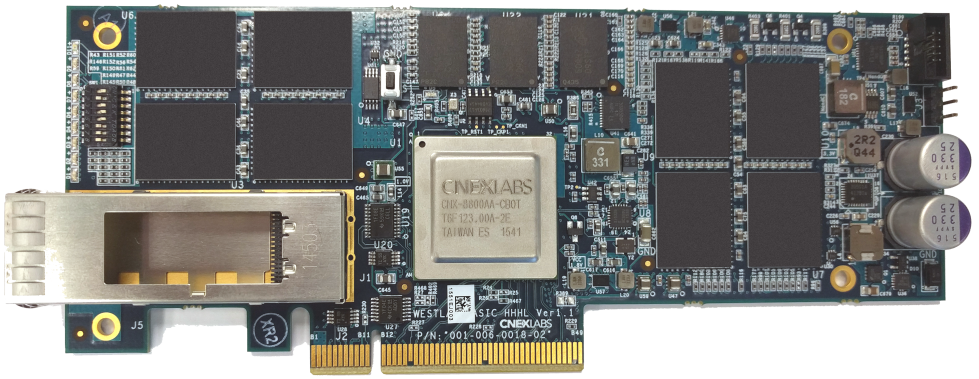
* Not real performance results - FPGA prototype, not ASIC

Evaluation: RocksDB on CNEX WestLake FPGA SDK

RocksDB make release	ENTRY KEYS with 4 threads	WRITES (1 LUN)	READS (1 LUN)	WRITES (2 LUNS)	READS (2 LUNS)	WRITES (64 LUNS)	READS (64 LUNS)
RocksDB DFLASH	10000 keys	23MB/s	40MB/s	35MB/s	X	X	X
	100000 keys	26MB/s	40MB/s	33MB/s	X	X	X
	1000000 keys	25MB/s	40MB/s	32MB/s	X	X	X
Raw DFLASH (with fio)		32MB/s	64MB/s	64MB/s	128MB/s	920MB/s	1,3GB/s

Status & Ongoing work

- Status:
  - LightNVM is in the upstream Linux Kernel (4.4)
  - 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 logic to liblightnvm
  - Improve I/O submission
    - Submit I/Os directly to the media manager
    - Support libaio on the traditional stack
  - Exploit device parallelism within RocksDB's LSM



Tools & Libraries

lnvm – 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

Open-Channel SSD Project

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>
NVMW'16

Liblightnvm APIs

Raw I/O API

- int **nvm_get_block**(int tgt, uint32_t lun, NVM_PROV _*prov);
- int **nvm_put_block**(int tgt, NVM_PROC *prov);

- int **nvm_target_open**(const char *tgt, int flags);
- void **nvm_target_close**(int tgt);

- int **nvm_flash_write**(int tgt, NVM_VBLOCK *vblock, const void *buf, size_t ppa_off, size_t count, NVM_FLASH_PAGE *fpage, int flags);
- int **nvm_flash_read**(int tgt, NVM_VBLOCK *vblock, void *buf, size_t ppa_off, size_t count, NVM_FLASH_PAGE *fpage, int flags);

Append-only API

- int **nvm_beam_create**(const char *tgt, int lun, int flags);
- void **nvm_beam_destroy**(int beam, int flags);
- ssize_t **nvm_beam_append**(int beam, const void *buf, size_t count);
- ssize_t **nvm_beam_read**(int beam, void *buf, size_t count, off_t offset, int flags);
- int **nvm_beam_sync**(int beam, int flags);