# BlueStore SMR Support
Shehbaz Jaffer
March 24, 2016

## Proposal

Create an SMR friendly block allocator scheme for Object Files in Bluestore.

## End Objective

*1. Minimum Goal (Must Have)*

1. The project will help bluestore file store allocate or deallocate blocks on a single raw file that emulates SMR drive.
2. A group of files emulated as multiple SMR drives shall be exposed as one allocation unit. Allocation, deallocation would then be possible on these group of drives.
3. Write Unit Test cases to verify correctness for single threaded operations. Add tests for multiple thread operations. Do performance evaluations using basic PUT/GET operations.

*2. Extended Goal (Should have)*

1. The project will help bluestore file system allocate, deallocate and manage blocks on group of real HGST or Seagate SMR drive, that is compliant with ZAC/ZBC standards. (*this will require sending drives to students work location*)
2. Performance Evaluation on real drives - these may be published and shared with the Ceph community or HDD vendor community.

*3. Future Work* (Could Have)

1. Provide support for CMR + SMR Drives: We will provide allocation scheme such that both CMR drive and SMR drive may be used simultaneously for storing Object files. Here, given the size and number of CMR and SMR drives, the Allocator will perform allocate() and release() operations, among other operations.
2. Performance evaluation - This will involve performance evaluation for single emulated drive and multiple emulated drives. We *may* use COSBench to do performance evaluation.

## Implementation Details

### *1. Component Level Additions*

1. Allocator changes

    a. *allocation/deallocation* - Add `SMRAllocator.cc` in os/bluestore. It will add another `class SMRAllocator` derived from `class Allocator`.

    b. *StupidAllocator.cc* Existing allocator uses a `btree map` for maintaining `free`, `uncommitted` and `committing` extents. The allocation is based on the length of the extent, and the minimum allocation size (default 64K). For allocation, we will have to maintain data structures pertaining to SMR zones, so that allocation can be made zone friendly, i.e. try to fill more data in 1 zone before closing it. This will avoid multiple partially filled zones and give a bimodal zone distribution. This idea is tried and tested before [5].

    c. *Garbage collection* The `release`() fuction in current Allocator only updates the btree map with the offset and length of the extent to be deallocated. We will have to do the corresponding extent free() on SMR drives atomically, which will involve clearing a zone and rewriting undeleted zone atomically.

2. SMR drive API/ Interfaces -

    a. SMR APIs have to be written to call out for querying disk status before allocations are made within or across zones.
    b. *Company Compliant interfaces* - An input from mailing list was that different SMR vendors have their own specifications. The project will follow the ZBC/ZAC standards that are followed by libzbc library. The SMR APIs will include extensible modules that can add any vendor specific features that may aid allocation/deallocation mechanism for SMR drives in future.

### *2. BlueStore File Store*

*Description:* The BlueStore file store consists of a KV store maintained in RocksDB, and a bunch of Object Files. The scope of the project is to perform allocation/deallocation for only object files.

*BlueStore Allocator*: Current Allocation scheme is created for CMR drives. It does not take into consideration the restrictions that are imposed by Host Managed SMR drives (sequential writes within zones etc.)
We will add more intelligence to bluestore Allocator to query an SMR drive before making allocation decisions based on

- extent resides on single zone
- extent spans multiple zones

We need to design appropriate data structures that can be created and freed on every alloc or free call. *Note: (Design for this will be shared with mentor before any final implementation)*

## 3. *SMR Drives*

*Description:* Shingled Magnetic Recording drives are a new class of drives that have disk tracks aligned in the form of shingles. Tracks on a platter of SMR drive overlap, hence writes need to be done sequentially over an area of disk. This area of disk is called a ***zone***. Within a zone, only sequential writes may be performed. For any block update within a zone, entire zone needs to be opened, re-read and written to a new zone, with the updated block. This behaviour is similar to Flash devices - hence we may borrow update and GC logic from File systems specific for Flash Drives (F2FS).

SMR Drives comply the ZAC/ZBC standards, which are a group of standards made to query Zone semantics before reads and writes can be made. Few restrictions for writing data to SMR drives are:

1. Random writes cannot be done within a zone
2. A write on a zone can only be done when the zone is “open”
3. A limited number of zones can be kept “open” at the same time.

## *4. Libzbc*

We propose using libzbc - an open source library https://github.com/hgst/libzbc as an SMR drive emulator. This library is *thread safe*. Libzbc exposes the following interfaces to the calling APIs:

zbc_device_is_smr , zbc_open , zbc_close, zbc_get_device_info, zbc_report_zones, zbc_report_nr_zones, zbc_list_zones, zbc_open_zones, zbc_close_zones, zbc_finish_zones, zbc_reset_write_pointer, zbc_pread, zbc_pwrite, zbc_write, zbc_flush

The project will involve building APIs that would query the SMR drive based on the above mentioned SMR drive interfaces. For instance, we can query for zbc_get_device_info and check the amount of data blocks that are remaining in the Zone before physically allocating blocks to the drive.

## RoadMap

*Present - 21st Apr*

- Read Bluestore allocator code.
- Trace how current allocation for CMR Drives work
- Get more familiar with libZBC and SMR drive interface. Start designing interfaces that will be compatible with SMR Drive, *share design with Mentor*.

*22nd Apr (Results Announced) - 23rd May (Coding Begins) -*

- Finalize proposal with mentor.
- Identify regions in bluestore allocator code that we need to add/modify.
- Identify API calls to alloc that we need to handle.
- Write down overall design and *verify with mentor.*

*23rd May (Coding Begins) to 15 June (Mid Term Evaluation)*

- Support for single SMR drive alloc / dealloc added, verified and tested.

*15 June - 20 June (Mid Term Evaluation Begins)*

- Write Unit Tests. Procure real Host Managed SMR Drives from HGST/Seagate.

20 June - 27 June (Mid Term Evaluation End)

- Write Mid-term report. *Seek mentors advice on progress*.

*27th June - 20 July*

- Work on multiple SMR drive support in emulated work.
- See if single allocator works for real single SMR drive, try for 4-5 physical drives.
- Benchmark using standard Ceph Benchmarks (CosBench)

*20 July - 10 Aug*

- Work on SMR + CMR drive support (require inputs from mentor)

*10 Aug - 15 Aug(Final Evaluation Begins)*

- Test Cases, More bug resolutions - final code refactoring
- (Optional) Run Ceph on multiple Host Managed SMR drive and CMR drive.

*15 Aug - 23 Aug (Final Evaluation Ends)*

- Final Documentation. SMR support code submitted for integration in BlueStore.

## References


1) http://www.slideshare.net/sageweil1/ceph-and-rocksdb
2) https://github.com/intel-cloud/cosbench
3) https://github.com/hgst/libzbc
4) http://www.storagereview.com/what_is_shingled_magnetic_recording_smr
5) https://www.cs.berkeley.edu/~brewer/cs262/LFS.pdf