SDC 20
BY Developers FOR Developers
Storage Developer Conference
September 22-23, 2020

File System Native Support of Zoned Block Devices: Regular vs Append writes

Naohiro Aota
Western Digital Research, System Software Group

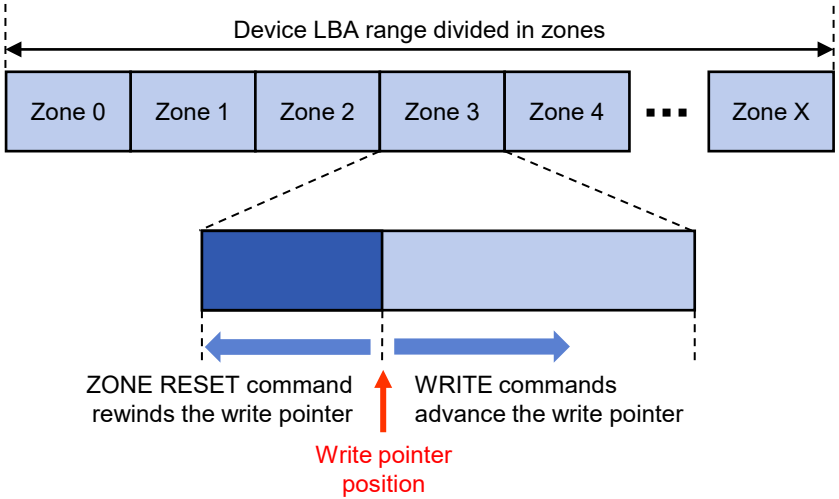
Outline

- Zoned Block Device (ZBD) Overview
- Linux ZBD support
  - F2FS, zonefs (in the previous talk)
- Btrfs ZBD support
  - Overview of btrfs and its IO system
  - ZBD support design
    - Device Extent and IO submission
    - Regular write vs Zone Append write
    - Log structured super block updates
- Performance Evaluation Results

Zoned Block Devices

Random reads but sequential writes

- Standard with
  - ZBC & ZAC for SMR hard-disks (Shingled Magnetic Recording)
  - NVMe ZNS for SSDs
- LBA range divided into zones
  - Conventional zones
    - Accept random writes
  - Sequential write required zones
    - Writes must be issued sequentially starting from the "write pointer"
    - Zones must be reset before rewriting
      - "rewind" write pointer to beginning of the zone
- Users of zoned devices must be aware of the sequential write rule
  - Device fails random writes

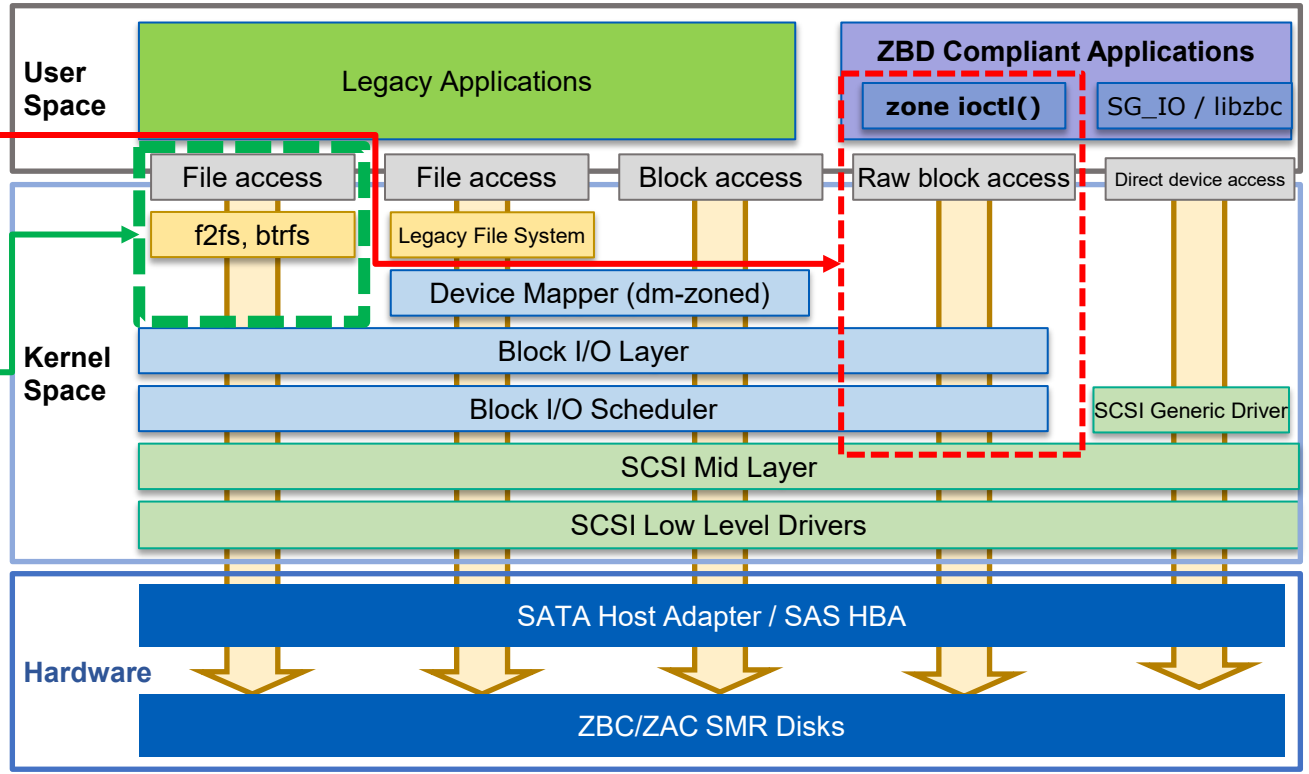


Native File System Support

F2FS and zonefs upstream, Btrfs is on-going work

Minimal support:
- Device sequential write constraint exposed to users
- Zone management API (ioctl) and write ordering guarantees

Advanced support:
- **User gets POSIX defined behavior**
- **No sequential write constraint**



F2FS Support For ZBDs

- F2FS natively support ZBDs since Linux 4.10
  - Based on F2FS “lfs” mode
    - Pure log-structured operation
    - No optimization with update-in-place for metadata blocks
- Sections (group of 2MB segments) aligned to device zones
  - Block allocation is sequential within and among segments of a section
- Atomic block allocation and write I/O issuing
  - Per section (i.e. per zone)
  - Ensures sequential write ordering derived from sequential block allocation
- Requires conventional zones !
  - To accommodate updates to fixed location metadata blocks
  - ZNS needs multiple devices to work because there are no conventional zones

Btrfs Native ZBD Support

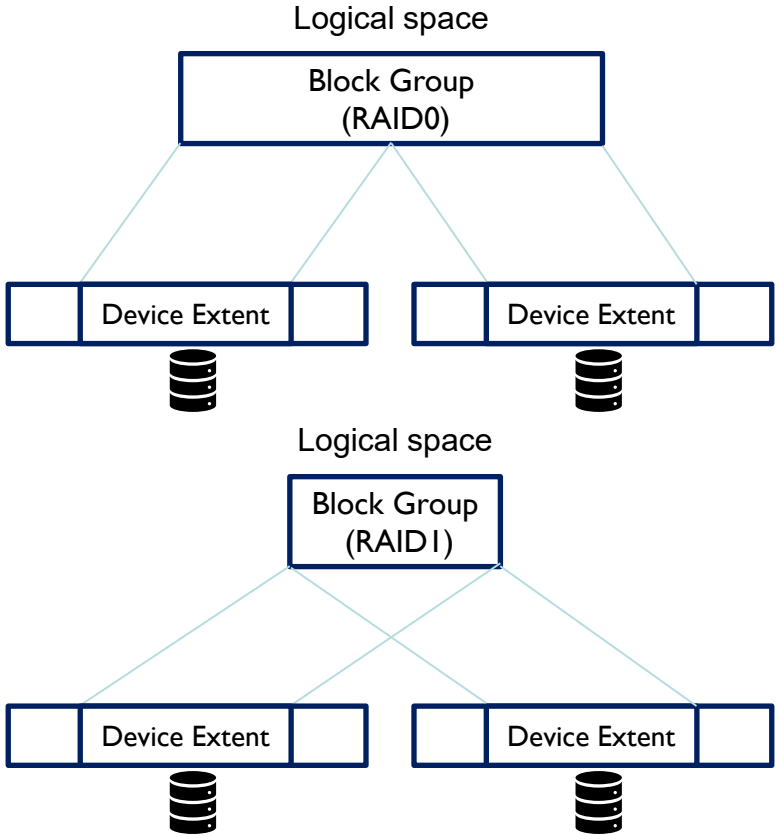
We need to satisfy ZBD sequential write constraint

- But, btrfs is not zone aware:
  - The size of device extents may not necessarily be aligned to zones
    - Typical (fixed) SMR disk's zone size: 256MB
    - But data extents may be 1GB, and 256MB for metadata
- Copy on write is **not** the same as sequential write
  - Block allocation not always sequential within a block group
    - Reuse of lower addresses of freed blocks within a group
  - Not all blocks are CoWed
    - Super block at fixed location is overwritten
- Two areas need modifications to solve these problems
  - Device extent and block group layout
  - Blocks writeback (data and metadata) must be sequential per zone
    - Regular write operations vs zone append writes

Zone Aware Extents and Block Groups

Align extents to device zones

- Device extent always aligned to device zones
  - One extent == one zone
  - 256 MB for most SMR disks on the market today
- As a result, block groups naturally align to zones
  - For all RAID levels
- Sequential use of blocks within a group implies sequential use within its device extent(s)
  - E.g. sequentially writing to a block group satisfies the device zone sequential write constraint



Sequential Block Writeback

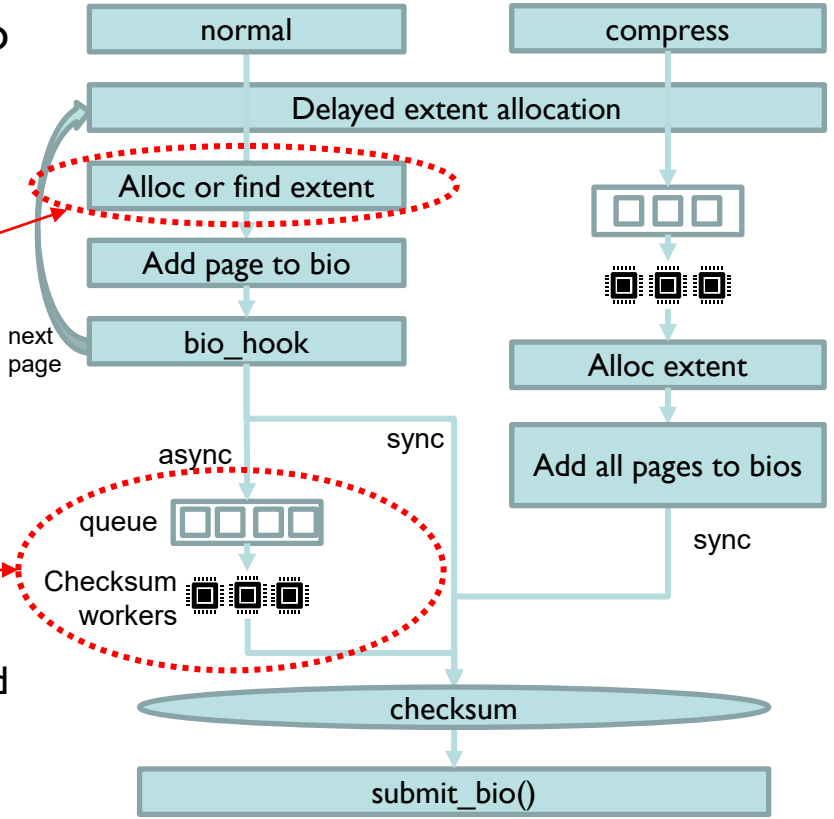
Regular write operations vs zone append writes

- Two possibilities
  1) Regular write operations (WRITE command) directed at zone write pointer
  2) Zone Append Write
     - No explicit target LBA specified, only a zone (using its start LBA)
     - The device automatically writes data at the zone write pointer
     - LBA position of written blocks returned in command reply
       - Similar to nameless write
- For regular write operations, we need:
  - Sequential block allocation **and** sequential write BIO issuing
    - The same as in F2FS
- For zone append write operations, we need:
  - Reserve blocks in a block group and write BIO issuing
    - Order does not matter
  - Update block allocation information on the write completion, after end_bio()

Data Write I/O Submission Overview

Highly asynchronous operations result in random write sequences

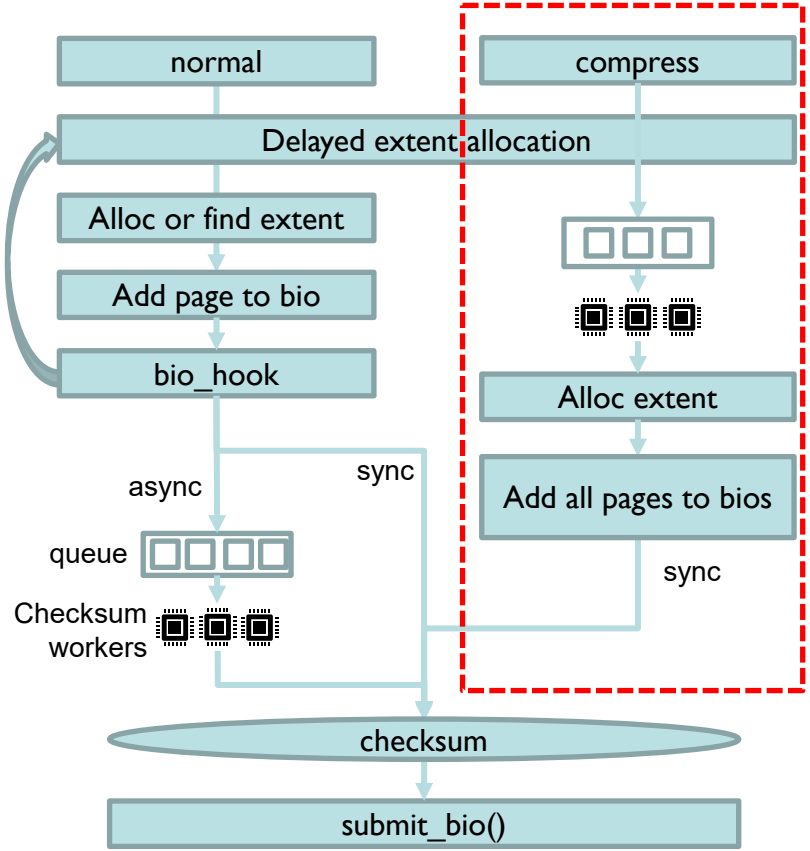
- Data write IO can take different paths to the device depending on its type
  - Normal or compressed data, pre-allocation (falloc), Direct IO
- Block allocation and IO submission are not atomic
  - IO submission outside of block group lock can result in random write sequence for a zone even with sequential allocation
- Asynchronous block checksum can reorder write IOs
  - Worker context different from allocation and issuing context



Regular Data Write I/O Submission

Near-ideal IO path exists for sequential writes

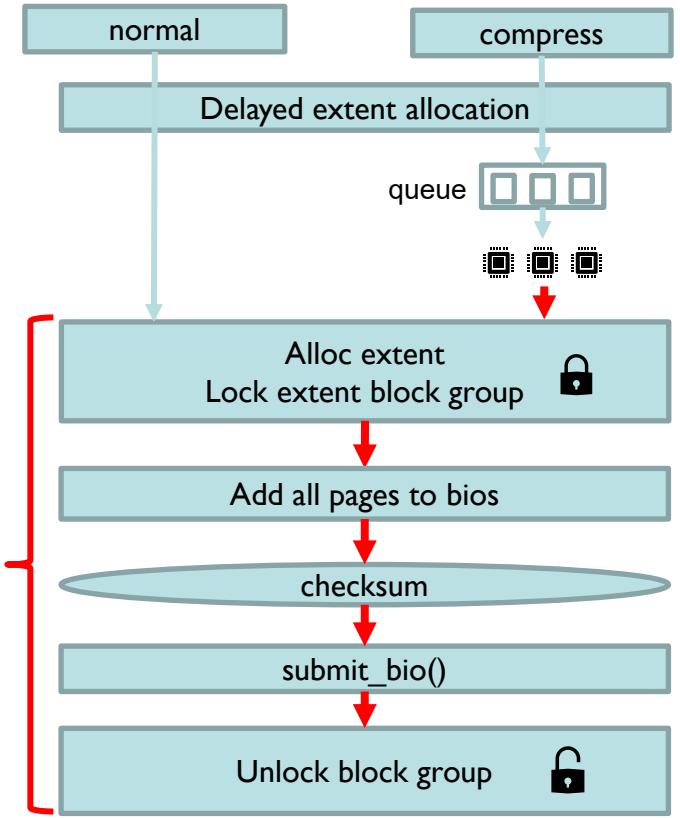
- Compressed data write IO path has a sequential behavior
  - Same context allocation and write IO submission
  - Sync block checksum
- Disabling asynchronous checksum results in a similar path for normal data writes
  - Serialized block allocation and write BIO issuing in the same order per context



Regular Data Write I/O Submission

Atomic block allocation and write IO submission

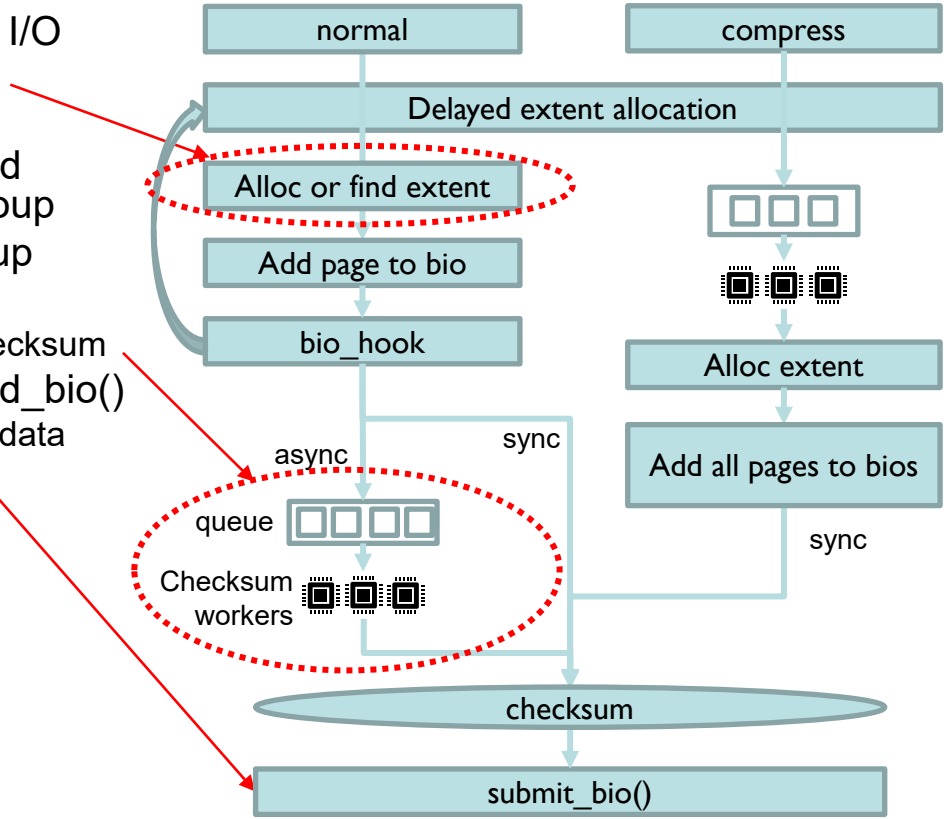
- Disable asynchronous block checksum
- Add per block group mutex lock
  - Serialize different write contexts
  - A file extent allocation locks the block group
  - Unlock only when all bios for the extent are submitted
- Preserves user facing features
  - Normal and compressed data support
- Parallel operations still possible
  - Granularity: block group (device extent zones)
  - Increased file parallelism can be trivially added with small changes to block allocator
    - Do not wait for a block group lock and use an unlocked block group



Zone Append Data Write I/O Submission

Simpler and no additional lock !

- Using Zone Append Write for Data I/O
  - No need for allocation before submitting bio
  - Only choose a block group and reserve blocks in the block group
  - No need to lock the block group
  - BIO order does not matter
    - Can utilize asynchronous checksum
  - Blocks used retrieved from end_bio()
    - Update block allocation metadata



Current Limitation of Zone Append Data Writing

- No RAID level supported
  - Cannot handle two Zone Append Writes sent to different zones/devices
    - They might return different LBA positions
- Still use dedicated write path
  - Always write whole range of delayed allocation
    - Normal write path may skip write out data e.g. outside of fdatasync() range
  - To use normal write path, we need to split existing file extent information before submitting bio
- Fragmentation of file extents
  - One IO is limited to zone_append_max_bytes (e.g. 512KB)
  - Currently, one IO is submitted per one file extent
    - Thus, file extent is limited to the max bytes as well
  - Increases number of file extents compared to regular btrfs

Metadata Write I/O Submission

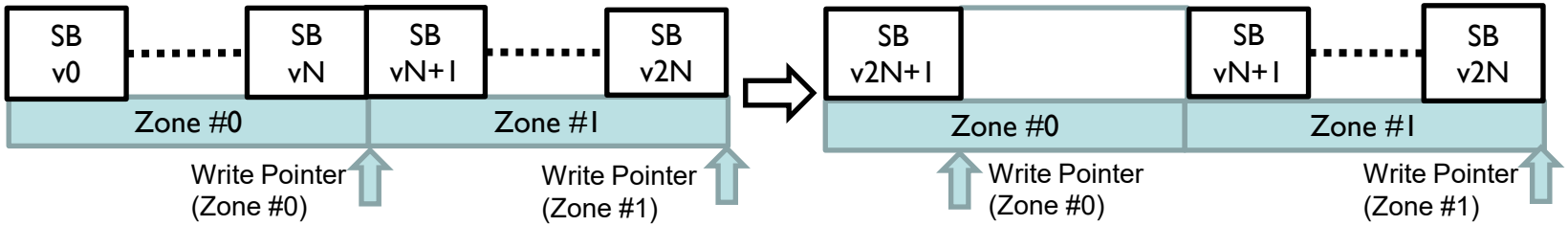
Rely on transaction ordering

- Similarly to regular data write I/O, all allocations in metadata block groups are changed to a purely sequential pattern
  - No reuse of freed block below current allocation position
  - Cannot use Zone Append Write
    - Because B-tree root and intermediate nodes need to record the writing address of underlying nodes
- No need for atomic allocation + write I/O submission under a single lock
  - Metadata writes are grouped per transaction, and all meta blocks added to the active transaction are sequentially allocated
  - Cleaned blocks in the active transaction are ignored but a zero-filled dummy block is written to preserve sequential write pattern

Log Structured Superblock Updates

Copy-on-Write superblock

- Superblock is the only fixed location data structure in btrfs
  - In-place updates require a conventional zone
  - Limiting superblock location to conventional zone have problems
    - Reduced number of superblock copy: only two copies are available per device
      - Second copy location (256GB) is on sequential write required zone
    - Cannot support device without conventional zones
- Employ superblock log writing
  - Use two zones as a ring buffer
    - Once the first zone is filled up, write in to the second zone and reset the first one
  - Device write pointer tell us where the latest superblock is.



Performance Evaluation

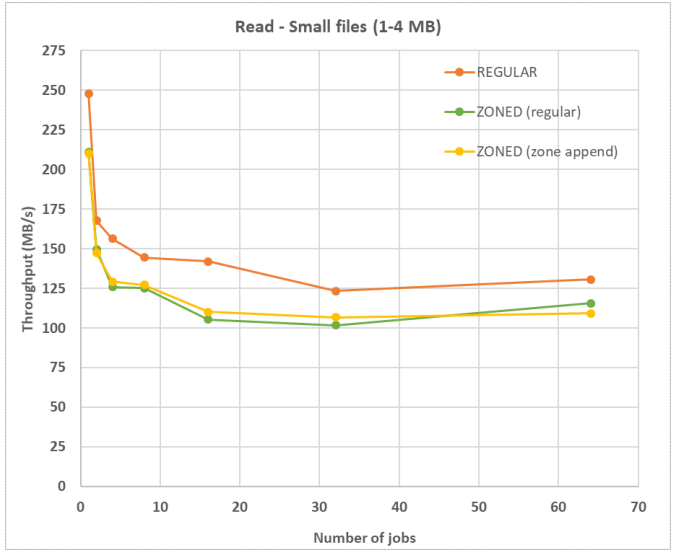
Regular disk vs zoned disk

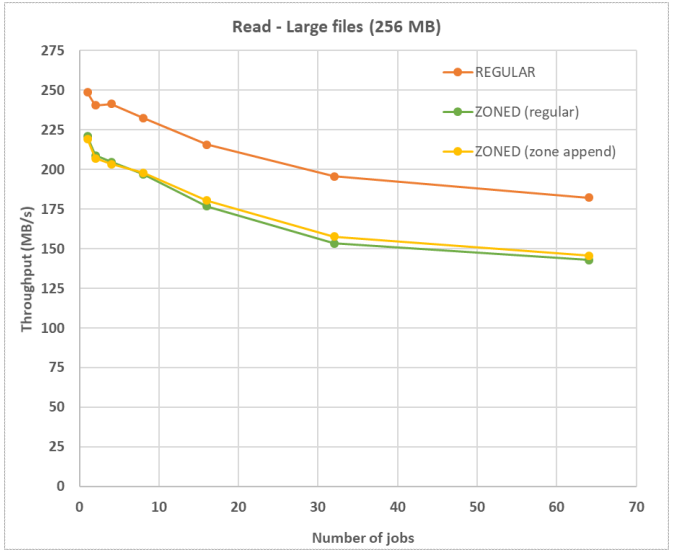
- Regular write operation IO path
  - Patch series posted on btrfs mailing list
- Hardware
  - REGULAR case: Regular 12 TB SATA disk
  - ZONED case: 16TB SATA disk with ZBC firmware (host managed model)
    - Same mechanics as REGULAR device
    - 55880 zones of 256 MB
    - 1% of conventional zones are CMR at LBA 0
  - Use dm-linear to create 0 conventional zone 40GB disk
    - To show performance of zone append writing which only works on sequential write required zones
    - REGAULAR also maps the same LBA range as in ZONED
  - **REGULAR and ZONED have different sequential write speed**
    - **ZONED is slower than REGULAR by around 10-15%**
- Fio: data workload
  - Operations: Write, Read, Read-Write
  - File size: 1-4MB, 256MB, jobs: 1, 2, 4, …, 64
- Dbench: fsync() heavy workload
  - Clients: 1, 2, 4, … , 32

fio – Files Read

1 to 4 MB random file size and 256 MB fixed file size

- ZONED degrades 10-20% from REGULAR
  - Due to the SMR disk used is 13% slower (sequential reads) than REGULAR disk
- ZONED regular write and zone append are competitive

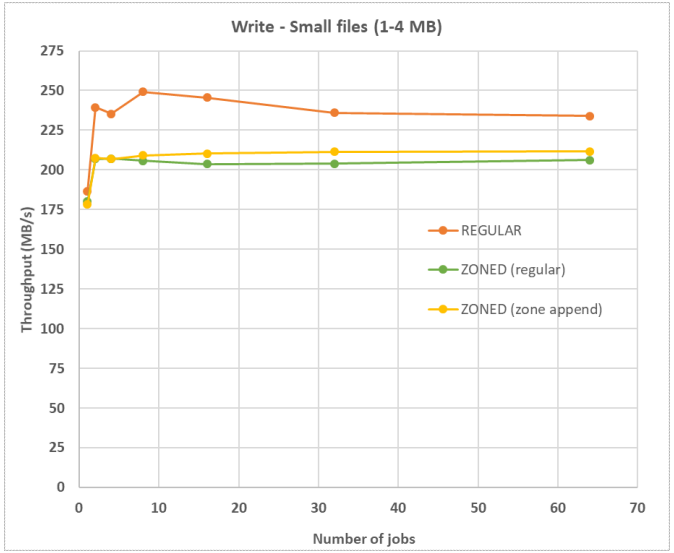


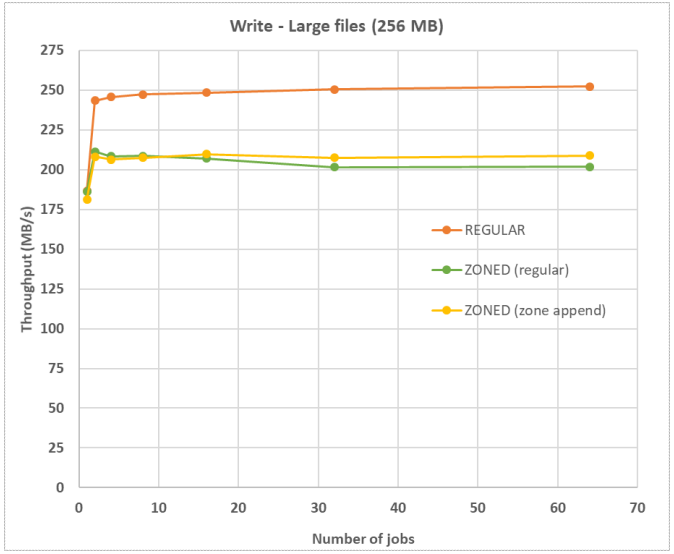


fio – Files Write

1 to 4 MB random file size and 256 MB fixed file size

- Same degradation as in the read case
- ZONED regular and zone append are competitive
  - Both are stable performance

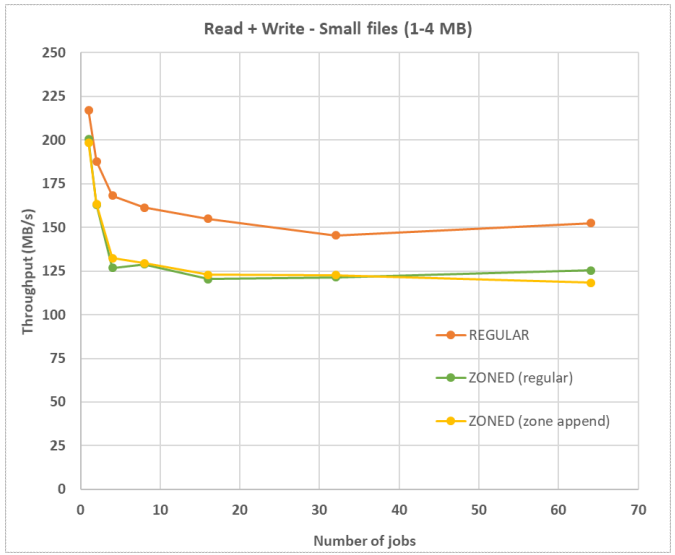


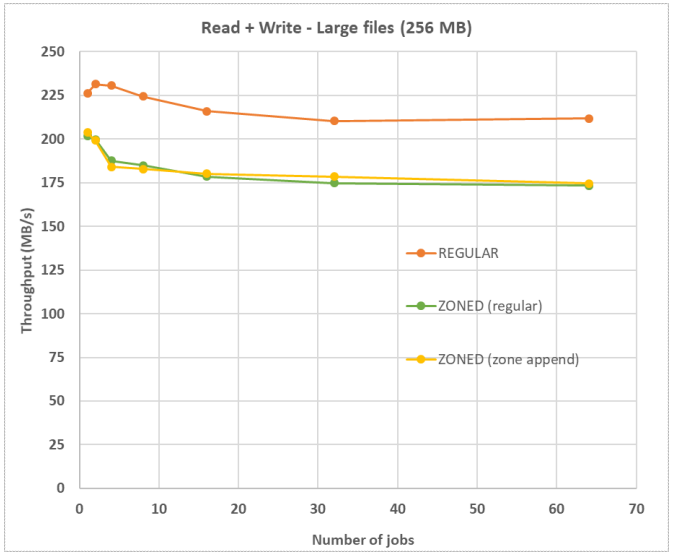


fio – Files Read/Write

70 % reads / 30 % writes

- Same degradation
- Zone append is up to 6% better than ZONED regular

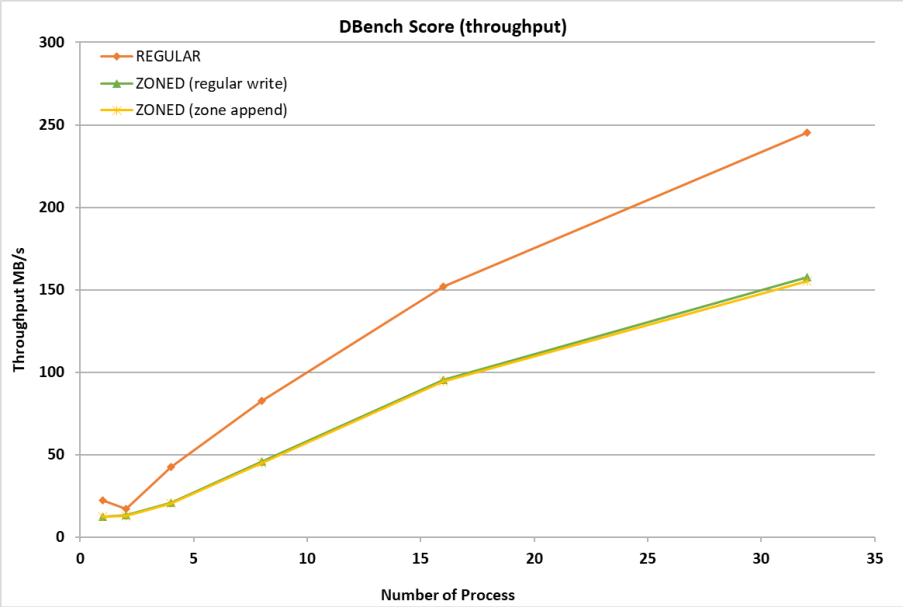




dbench

fsync() heavy workloads

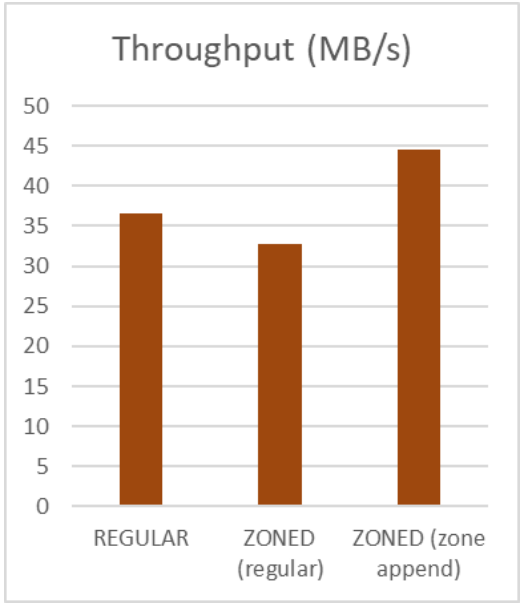
- ZONED degrades 20-50% from REGULAR
  - ZONED failed to scale
  - Linux 5.7 improved btrfs's fsync() performance
    - ZONED does not catch up with it
- Same here, ZONED regular and zone append are competitive



ZONED regular vs zone append

Zone append writing achieve 36% better in the best case

- Fio: 16 jobs random writing with 4KB IO size
  - Massive competing on the same zone
- Zone append is better than ZONED regular by 36%



Status and On-going Work

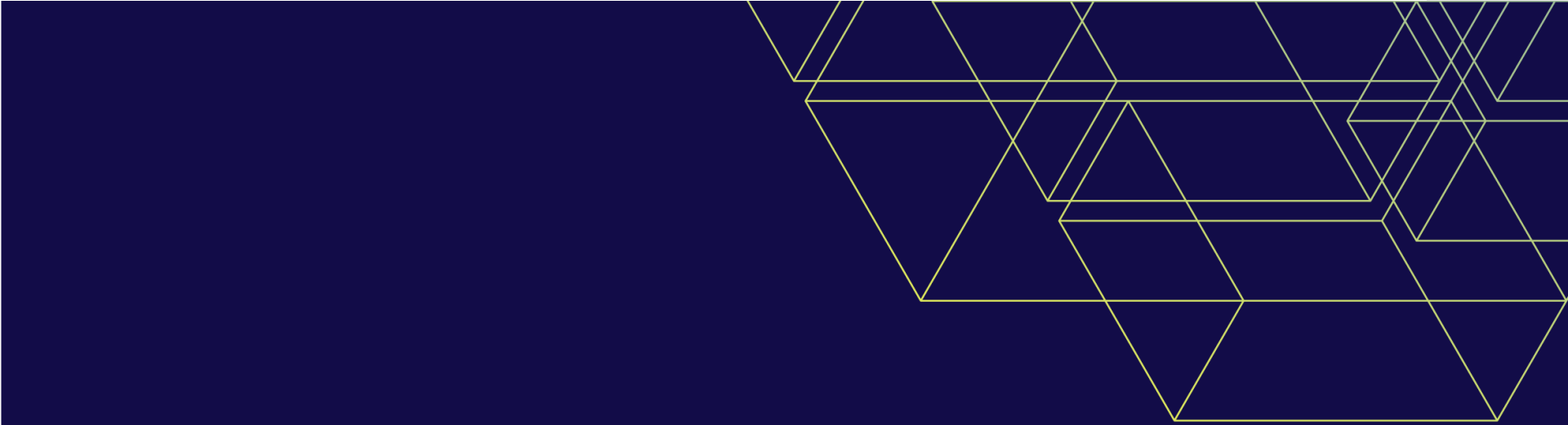
Zone append IO path implementation

- Next patch series targets zone append support in place of regular writes
  - Keep asynchronous check-summing (better CPU utilization)
  - No additional locking
- But need Zone Append support in SCSI to have a single code base for both SMR HDDs and ZNS SSDs
  - The SCSI disk driver (sd) can emulate it
- Planed optimization
  - Remove dedicated path for zoned data writes
    - Improve performance on small IO+fsync case
  - Opportunistic merging of file extents after end_bio
    - Useful also for regular btrfs

Thank you !

Questions ?

Western Digital®



Please take a moment
to rate this session.

Your feedback matters to us.

Disabled Features

- Currently:
  - RAID5/6
    - Non-full stripe write cause overwriting of parity block
    - Rebuilding on high capacity volume (usually SMR) can lead to higher failure rate
  - space_cache (v1), NODATACOW
    - In-place updates
  - Fallocate
    - Reserved extent creates a write hole
  - MIXED_BG
    - Allocated metadata region will be write holes for data writes

Data Write I/O Submission

Use conventional zones as sequential zones

- Conventional zones accept random writes
  - Do not need block allocation and I/O issuing atomicity
- The code can be simplified by treating conventional zone as sequential zone
  - Maintain allocation pointer
  - Per block group mutex lock
  - Allows mixing conventional and sequential zones within the same block group
  - Allocation/write state can be inferred on mount from the extent tree
    - Act as write pointer position information

Tree-log Block Write I/O Submission

Use a dedicated block group

- Use a dedicated metadata block group for tree-log blocks during fsync() processing
  - Results in 2 streams of sequentially allocated blocks from two different metadata block groups
    - One stream for tree-log blocks and another for other metadata blocks in transaction
  - Sequential writing of both streams is possible in any order
    - Each stream in each block group is sequential
- Serializing multiple fsync() transactions is still necessary
  - Blocks allocated in one transaction must written before the next transaction alloca blocks
- Small performance penalty but avoid the huge performance penalty introduced by disabling the tree-log