Filesystems & SMR at scale - Zoned Storage in the Linux kernel

Christoph Hellwig

Storage Days Tallinn 2026

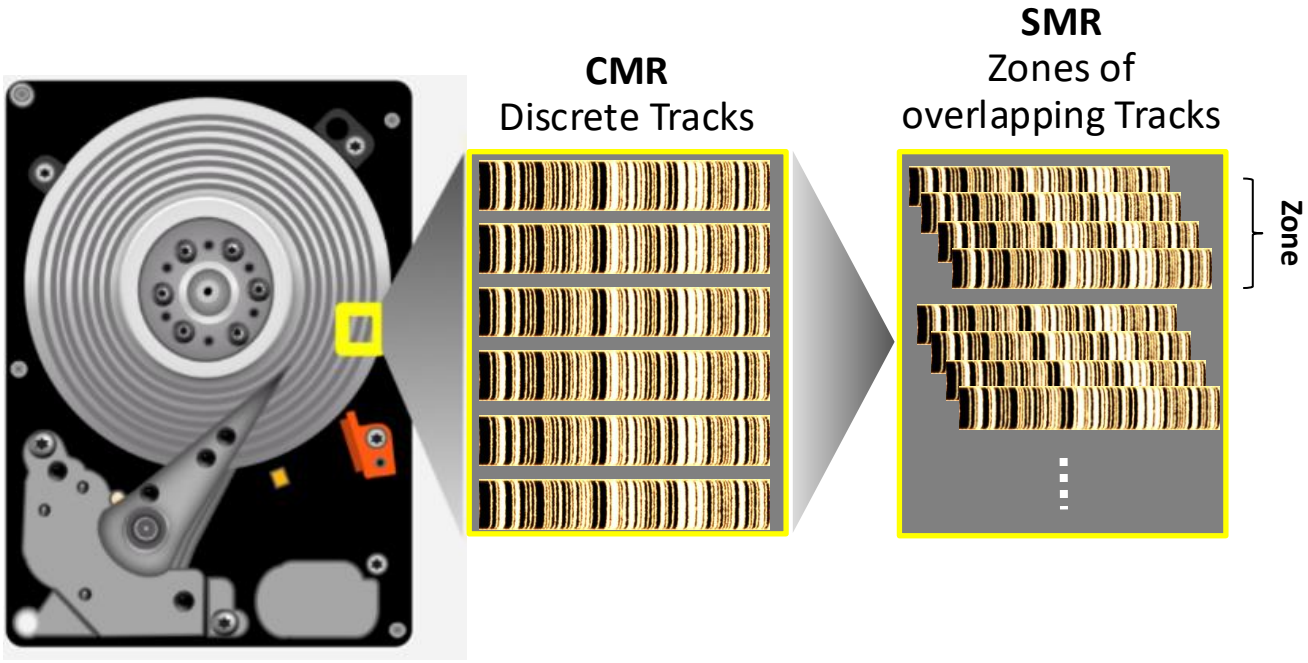
Outline

- What is SMR and where does it fit?
- Zoned Storage support in the Linux kernel
- XFS background
- XFS SMR support
  - Architecture and data block placement
  - Garbage collection processing
  - Performance
- Upcoming features

What is SMR and where does it fit?

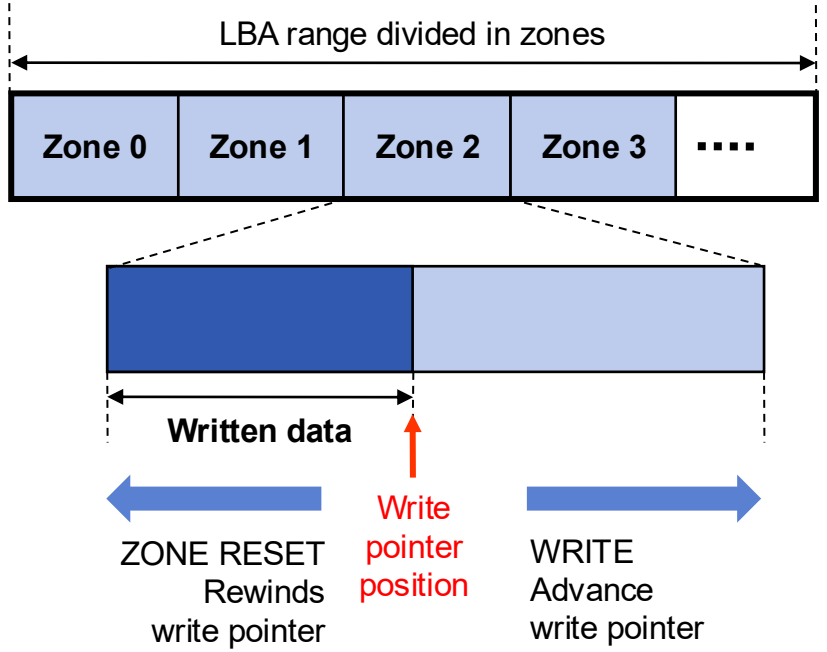
Conventional vs Shingled Track Format

- Conventional Magnetic Recording (CMR): non-overlapping discrete tracks
  - Sectors can be read and written randomly
- Shingled Magnetic Recording (**SMR**): zones of overlapping tracks
  - Sectors can be read randomly but must be written sequentially in each zone
- Applies to all recording technology (e.g. SMR on HAMR)



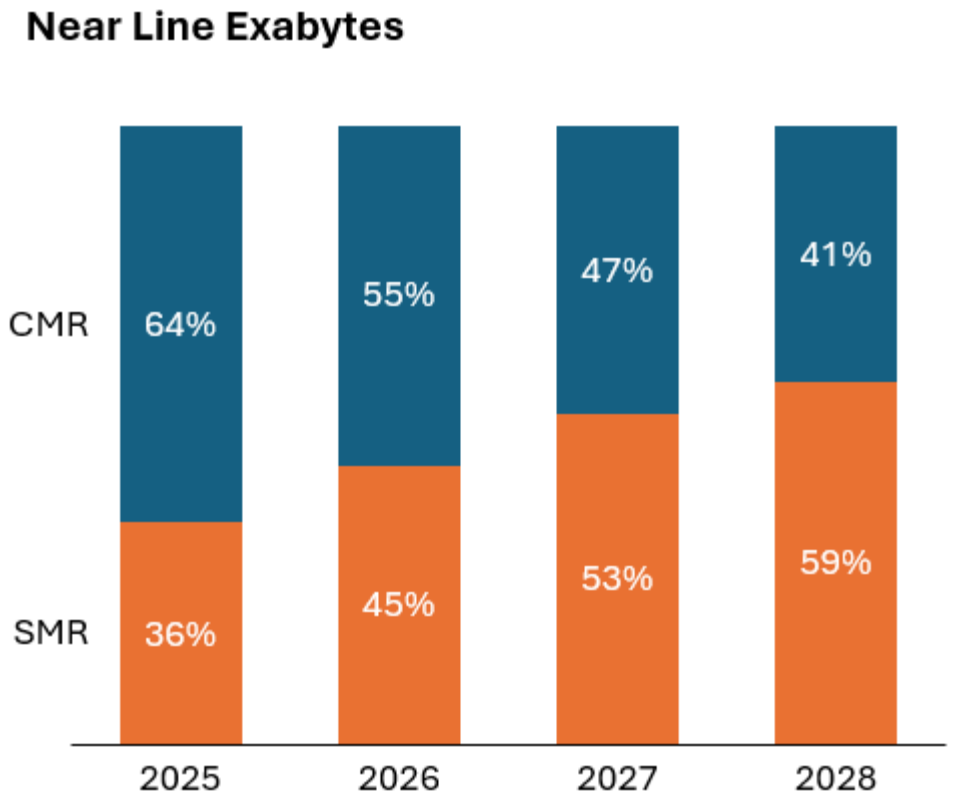
SMR Interface

- SMR HDDs divide the LBA address space into zones
  - Random reads, but sequential writes: writes must be issued sequentially starting from the write pointer
  - Conventional zones (optional): accept random writes
- Zone management commands defined by the ZBC (SCSI) and ZAC (ATA) standards
  - Report zones: zone type, write pointer location, ...
  - Reset zone: ”erase” a zone and rewind its write pointer
  - Finish zone: transition a zone to full state



SMR Adoption is Rising Globally

- SMR advantages
  - Lower costs with higher capacity in same form factor
    - 20 to 25% capacity increase over CMR
  - More stable performance
    - No adjacent track interferences
  - Better reliability and local error recovery
    - Sequential write pattern enables stronger ECC/data encoding
- SMR adoption has accelerated
  - The majority of the Exabytes deployed in data centers is already SMR

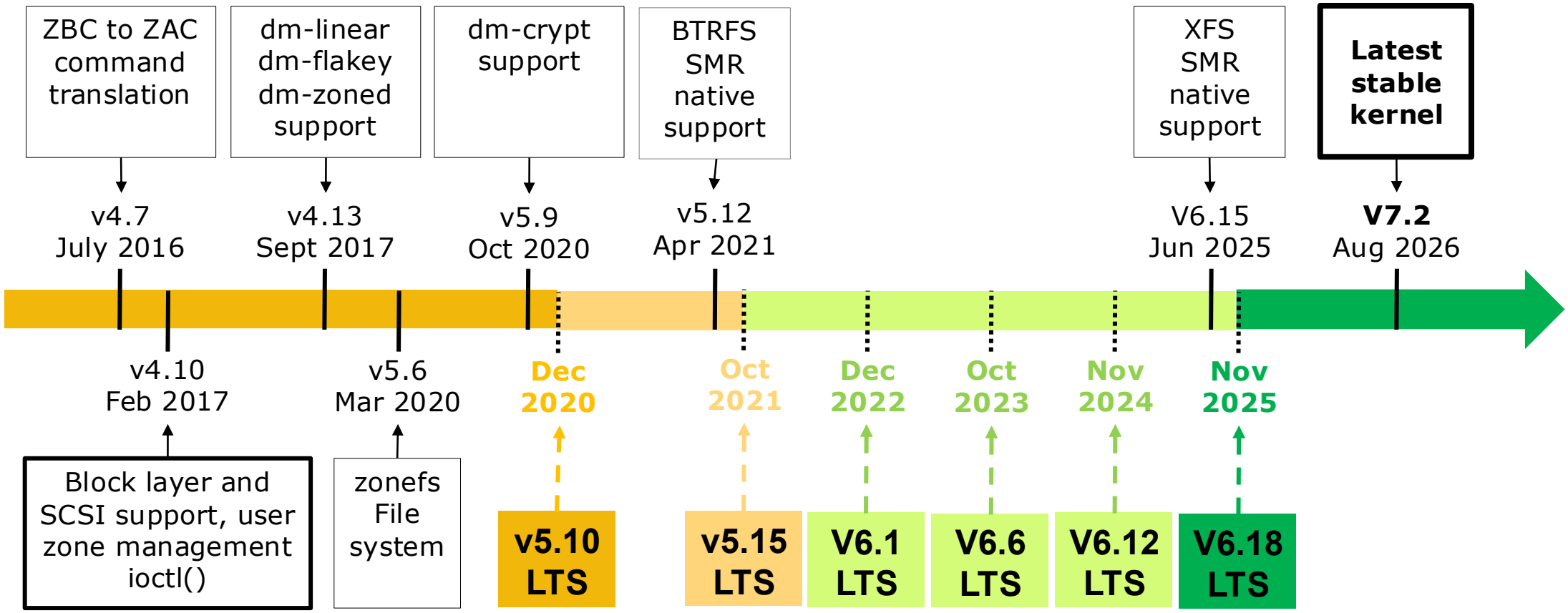


Source: TrendFocus Feb'2026 HDD Long Term Forecast

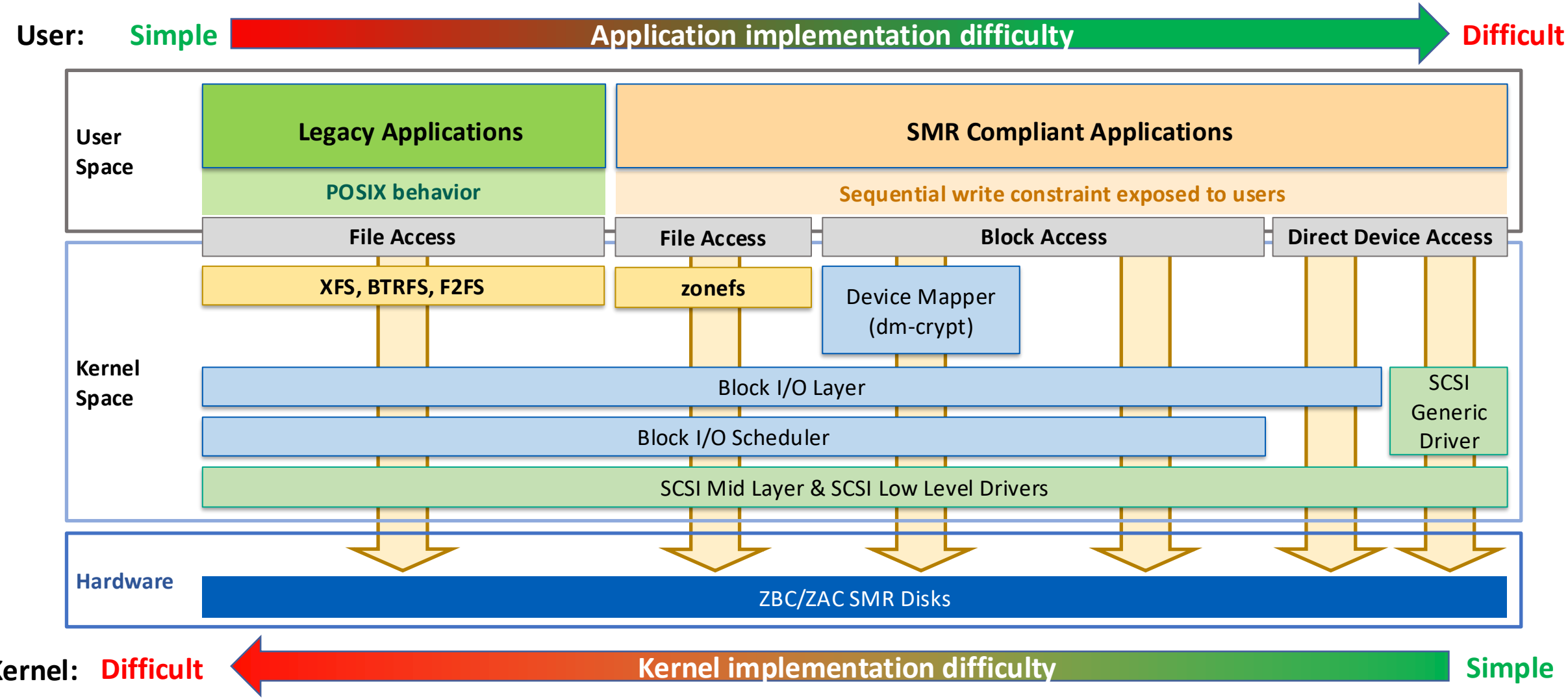
Barrier to adoption: software changes needed to implement sequential write patterns per zone!

Zoned Storage in the Linux kernel

SMR: 10 Years of Linux Kernel Support!

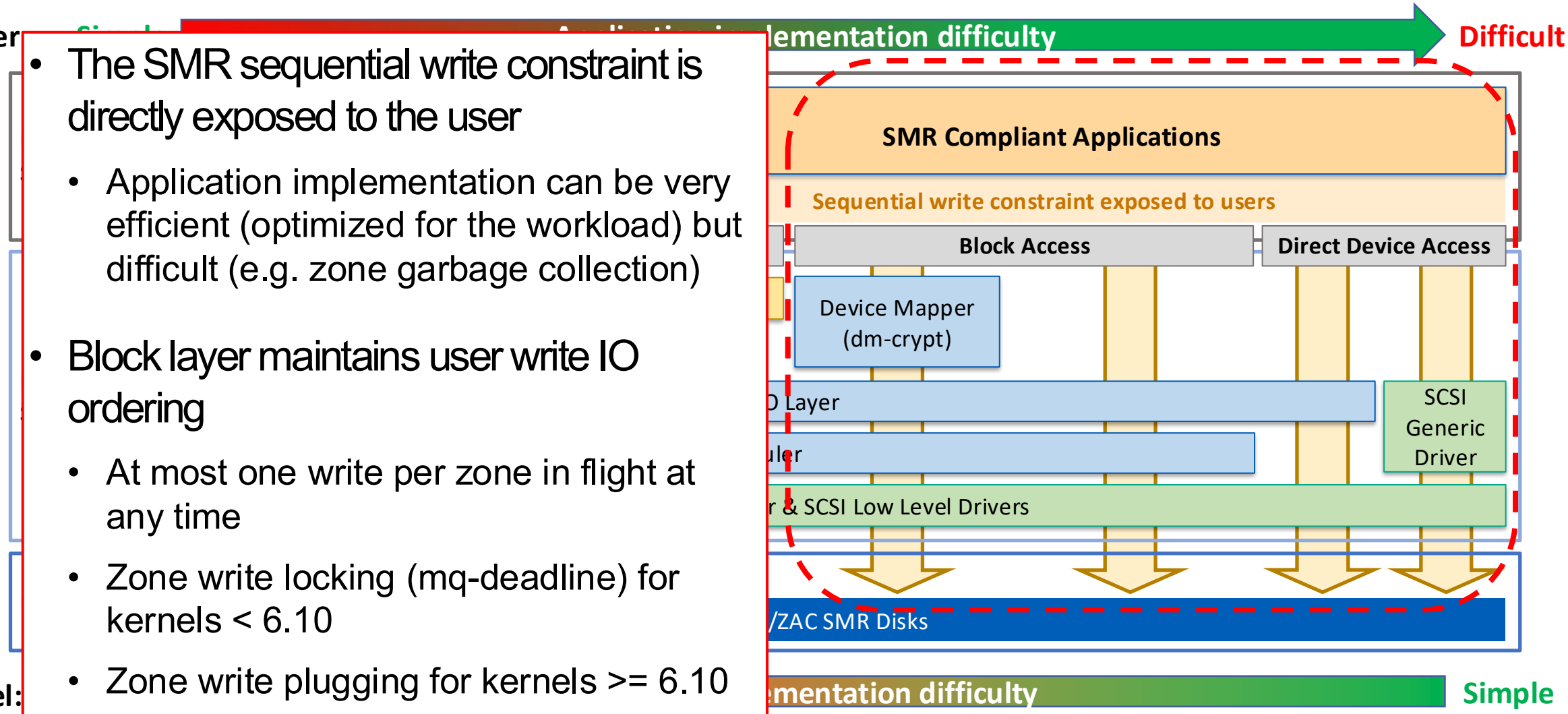


Kernel Support Current Status

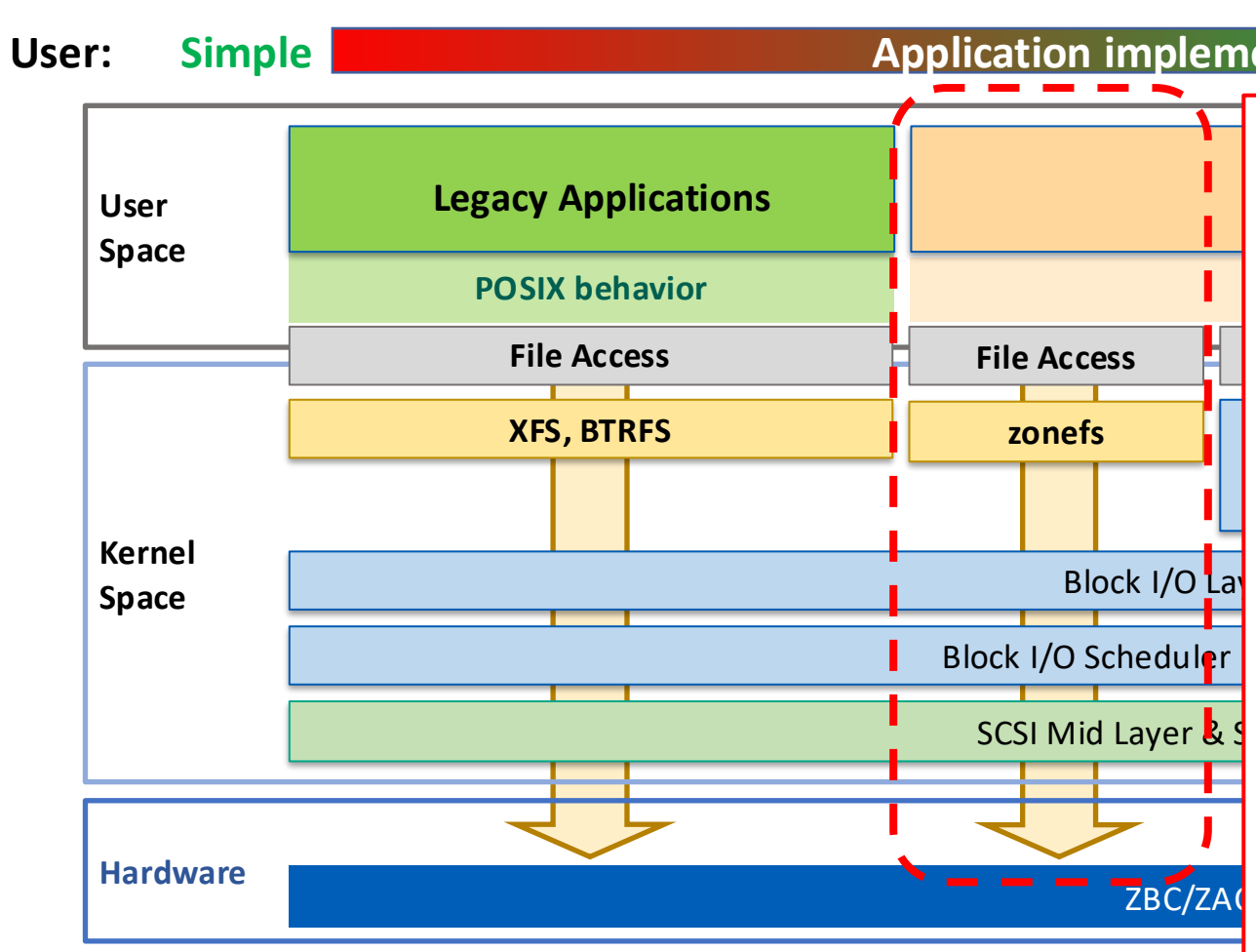


Kernel Support: Block Device

- The SMR sequential write constraint is directly exposed to the user
  - Application implementation can be very efficient (optimized for the workload) but difficult (e.g. zone garbage collection)
- Block layer maintains user write IO ordering
  - At most one write per zone in flight at any time
  - Zone write locking (mq-deadline) for kernels < 6.10
  - Zone write plugging for kernels >= 6.10

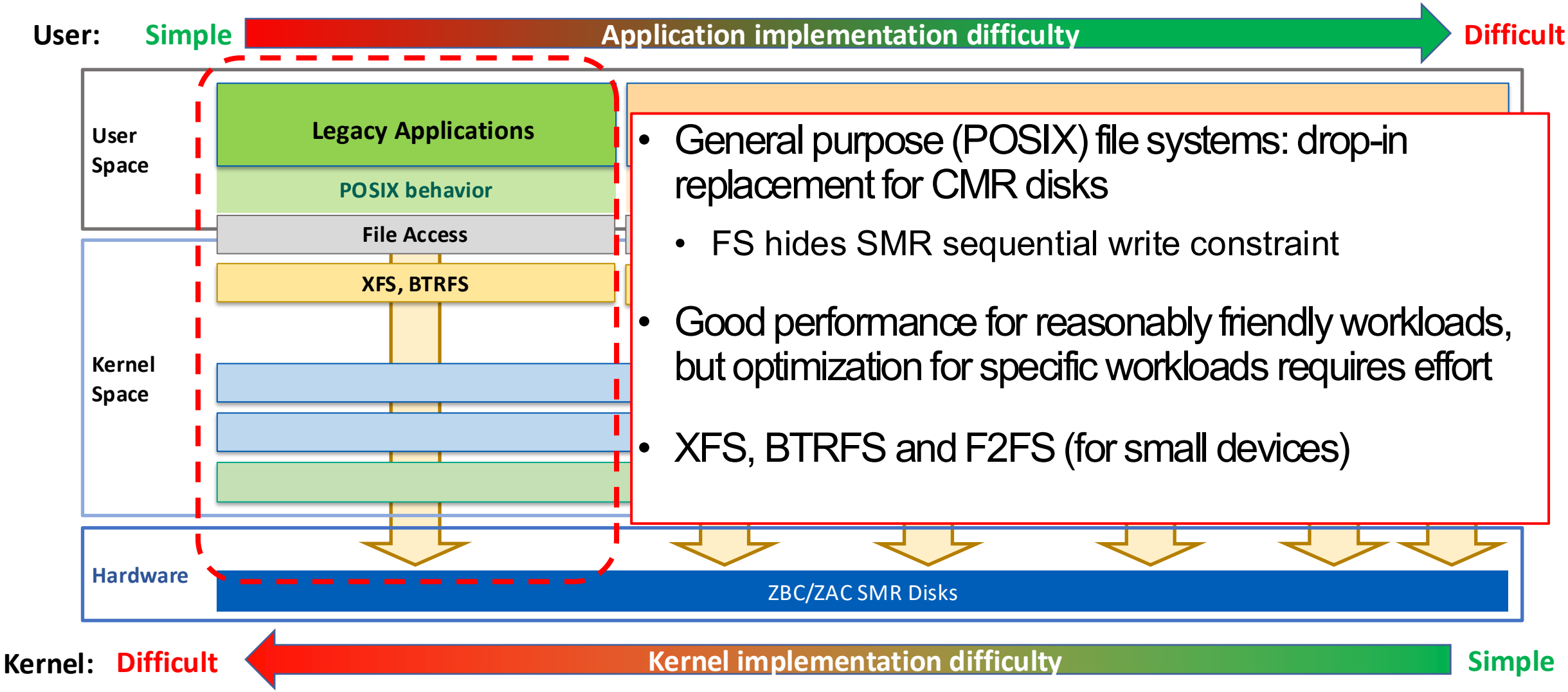


Kernel Support: Block Device-like File System



- Zones are exposed as append-only files
  - No intelligence: sequential write constraint directly exposed to the user
- But no zone management command use necessary
  - Zone management commands hidden under regular system calls
  - E.g. truncate file -> zone reset

Kernel Support: POSIX File Systems



- General purpose (POSIX) file systems: drop-in replacement for CMR disks
  - FS hides SMR sequential write constraint
- Good performance for reasonably friendly workloads, but optimization for specific workloads requires effort
- XFS, BTRFS and F2FS (for small devices)

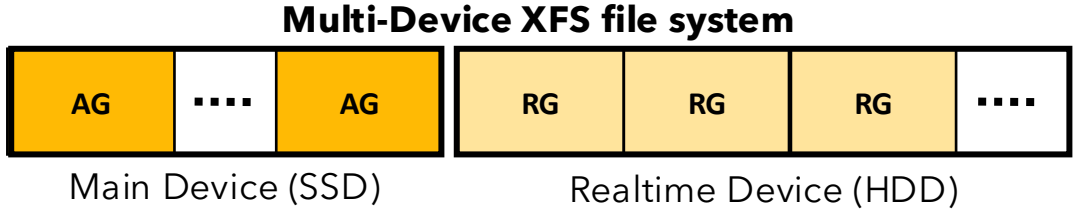
XFS Background

XFS Background

- Most scalable enterprise-grade Linux file system
  - 30 years old now: started on SGI/IRIX in 1993 and ported to Linux in 2001
  - Heavily uses b+trees and variable sized extents
- Space managed using allocation groups (AGs)
  - Equally sized linear regions with separate inode and free space management
    - Enables excellent (scalable) performance with concurrent I/O threads
- File data Copy-on-Write with *reflink* feature
  - Out-of-place file data block updates
  - Metadata is always updated in-place

XFS Background

- Reverse mapping tree
  - Mapping from physical blocks to logical blocks using variable length extents
    - Extents are indexed with a per-AG B+tree
- "Realtime" subvolume
  - Allows separating metadata and data onto different devices (main device and real-time device)
    - Data on the real-time device is managed using real-time groups (RTG)
  - Originally used for HD Video recording and encoding on HDDs back in the late 1990s
  - Recently most often used for separating metadata on SSD and file data on HDD
    - e.g. Facebook/Meta "Hybrid XFS" used for Tectonic distributed filesystem



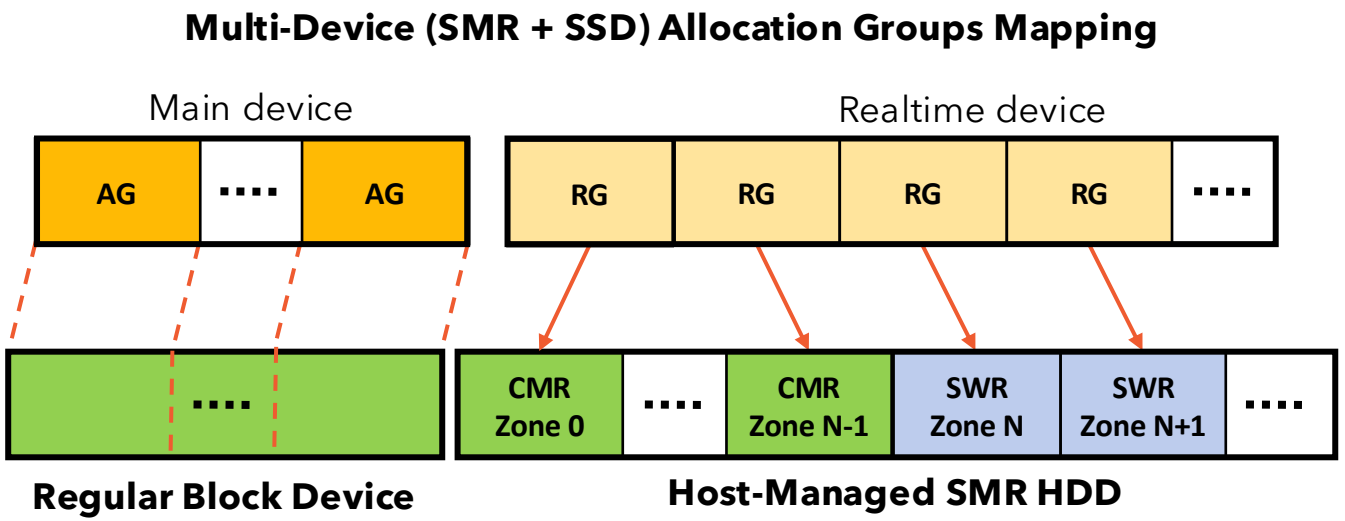
XFS on SMR HDDs

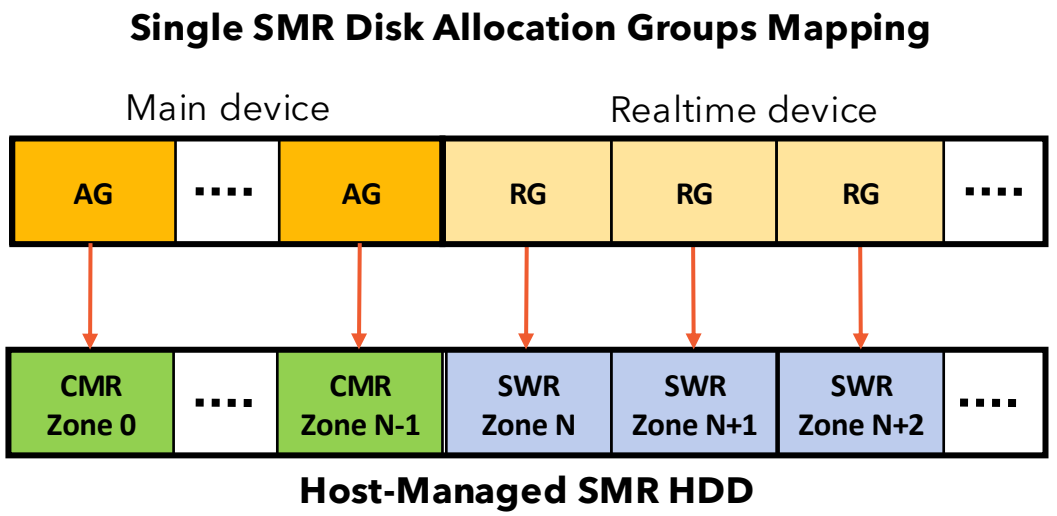
Zoned XFS Overview

- Extension of XFS to support zoned block devices (SMR HDDs and ZNS SSDs)
  - Drop-in replacement for CMR disks
- Uses the existing XFS real-time subvolume to separate data and metadata
  - XFS metadata requires random-writable (“conventional”) space
  - File uses is placed in the realtime subvolume and written sequentially
  - Each zone maps to an XFS realtime group (RG)
- Updates of existing file data write out of place
  - Reusing logic for writing to reflinked (copy on write cloned) files
- Garbage collection evacuates valid data from partially filled RGs
  - When space is reclaimed to create free zones (real-time groups) on the real-time device
  - Uses the existing XFS reverse mapping btree

Device Configuration

- For SMR HDDs with conventional zones, XFS can be self-contained
  - The conventional zones are used as the main device for metadata and the SMR zones are used as the real-time device (but beware of the limited storage space of conventional zones!)
- An SSD can also be used as the main device for metadata
  - Allows using a 100% SMR (higher capacity) HDD without conventional zones for file data





File Data Write I/Os

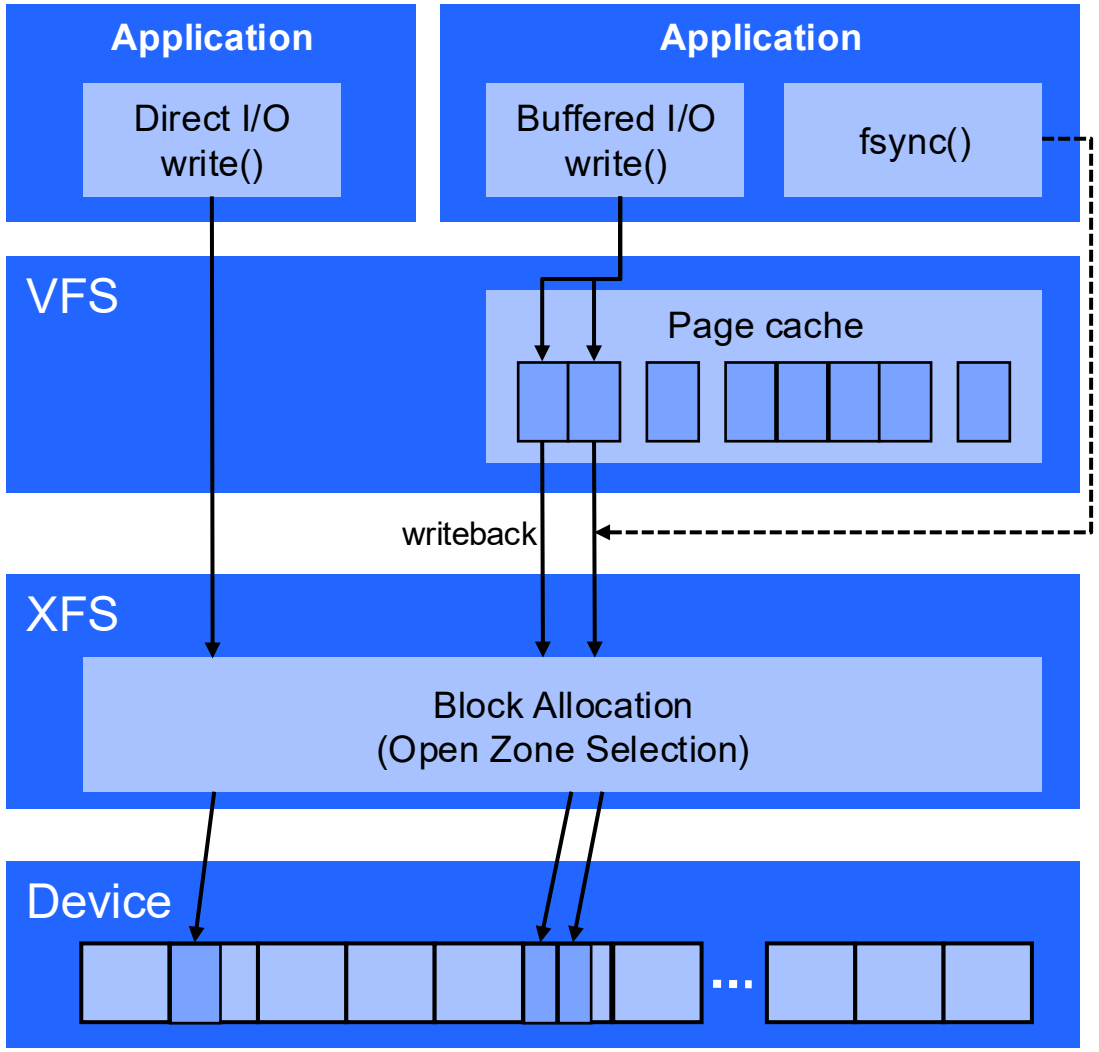
- XFS uses the Zone append primitive, which is directed to a zone instead of to an exact block location
- For devices that natively support this operation (e.g. NVMe ZNS SSDs), the device assigns the written block and returns it
  - Allows a high queue depth write operation
- For SMR HDDs, this Zone append is emulated by the block layer using regular write commands

File Data Mapping Metadata

- The zone append I/O completion handler records the logical-to-physical and physical-to-logical mappings of the written file data
  - There is no chance to have invalid data in the mapping, and thus no need to allocate unwritten extents that cannot be read before the I/O completes
- File data blocks that are freed or overwritten become unusable
  - These can only be reused after a zone reset
  - Records of unusable blocks are deleted from the reverse mapping btree, and the used blocks counter for the RG is decremented, but the free block counter is incremented only once the unusable blocks are reclaimed with garbage collection

Space Allocation

- Metadata reuses the regular XFS space allocator
- For file data, space allocation uses a new zone allocator
- Data block allocation happens when blocks are written to the device
  - Immediately with direct I/O writes (O_DIRECT) and with sync I/O writes (O_SYNC or O_DSYNC)
  - On dirty page writeback with buffered I/Os
    - After a timeout or under memory pressure, and when the application executes fsync()



File Data Space Allocation

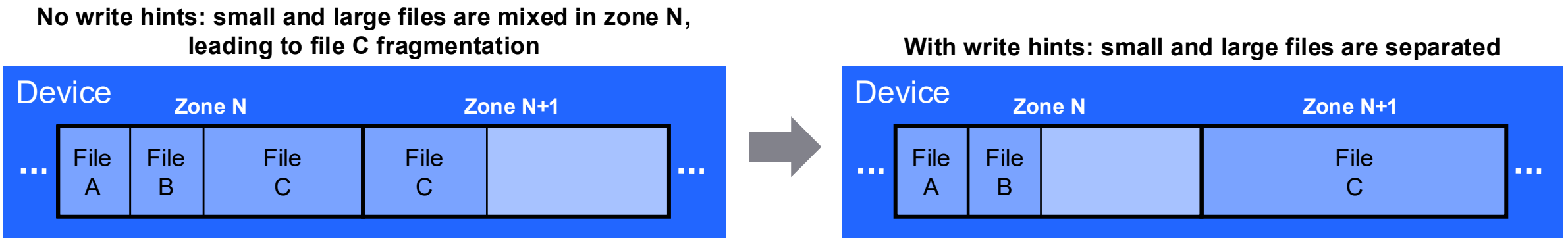
- For file data, space allocation is reduced to choosing a zone
  - SMR HDDs limit the number of zones that can be partially written (open zones)
    - This limit generally is hundreds of zones for SMR HDDs
  - All available open zones can be used, within the device limits
  - One open zone is set aside for garbage collection
    - The rest is available to user data writes
- File data is spread out to different open zones on a per inode (file) basis by default
  - Separating files into different zones reduces write amplification
  - A few heuristic allow grouping data from multiple files into the same zone

File Data Space Allocation

- Untar optimization
  - Unpacking of archives files, software installation or similar workloads can create many small files that are written once and rarely, if ever, updated
  - In this case, XFS tries to reuse the same zone to tightly store files that are written back by the VM after they have been closed
- Data temperature hints
  - The F_SET_RW_HINT *fcntl()* system call allows setting a data temperature for a file (inode)
  - XFS tries to collocate all data of files with the same data temperature (lifetime) in the same zones, if files cannot be spread out to different zones
    - All hot data is always co-located in the same zones such data is not expected to stay around for a long time

Mixing Small and Large Files

- If small files are also written on the same file systems as large files, fragmentation of large files may happen
  - Open zones are limited and many may already be used to store small files
- Solution: use write hints to indicate to XFS that files should be separated (grouped) in different zones (“small files” vs “large files”)

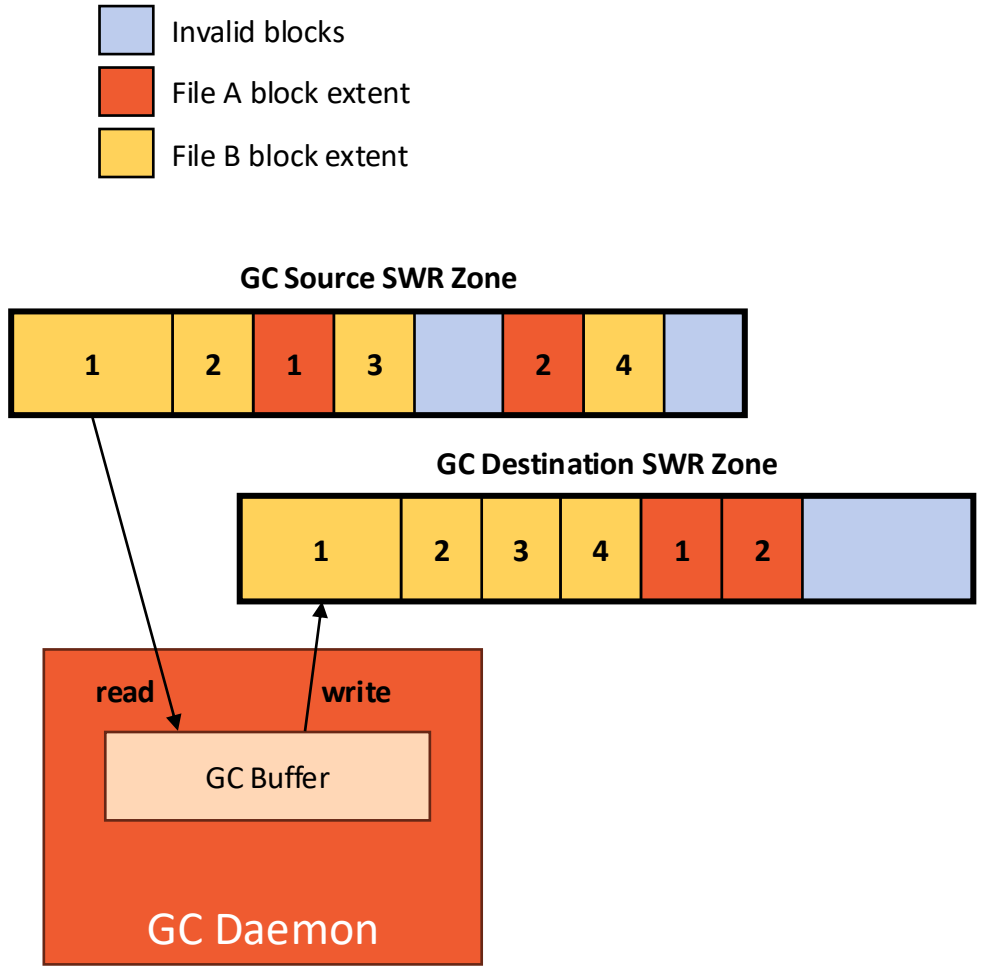


Garbage Collection

- Garbage collection (GC) is required to reclaim the space occupied by unusable blocks
  - Blocks that belonged to files that were deleted or data blocks of files that have been overwritten
- Zones that do not contain any valid block are reset immediately and made available for writing new user data
- For zones containing a mix of valid and unusable blocks, a single thread GC daemon is used to copy (evacuate) valid blocks to a GC reserved zone
  - The GC daemon is woken up when the number of free zones falls below a predefined (configurable) threshold
  - GC Candidate zones are sorted in buckets based on their number of valid blocks
    - The first zone with the least number of valid blocks is always selected first

Garbage Collection

- Valid data blocks are identified using the reverse mapping B-tree
  - Valid data blocks are evacuated to zones dedicated to garbage collection
  - New user data and garbage collected data is not mixed
    - Garbage collected data can be assumed to be longer lived, so this provides a simple hot/cold separation heuristic, reducing write amplification
  - File data blocks are sorted by inode and offset before copying, providing defragmentation “for free”
- The GC data copy process is pipelined and adds minimal locking overhead

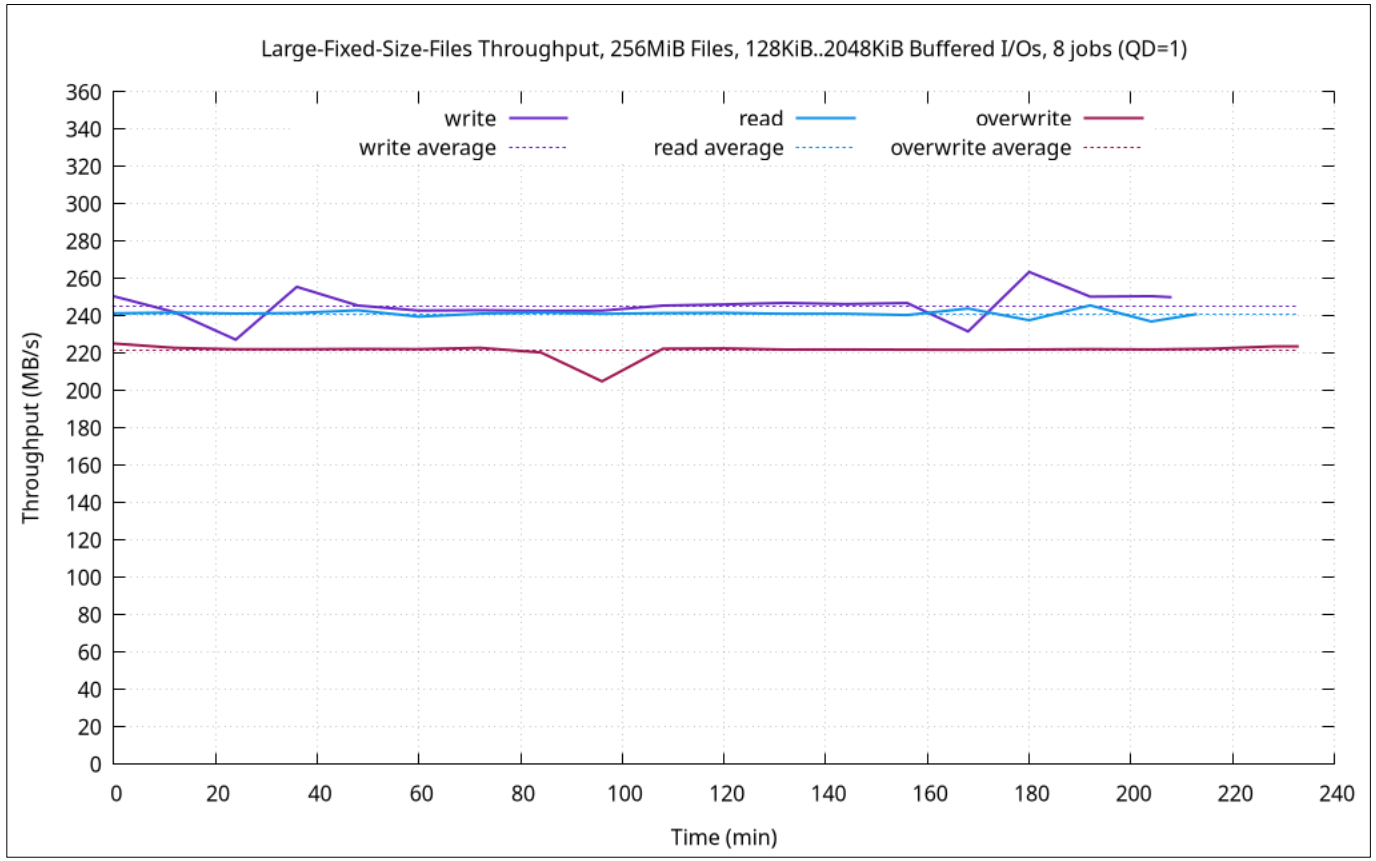


Zoned XFS Performance

- Environment:
  - Fedora 44 distribution, kernel version 7.1
  - Intel Xeon single socket, 8-cores (16 threads) CPU with 128 GiB of DRAM
  - SATA 30 TB SMR HDD connected through an Intel C620 Chipset AHCI adapter
- Workload
  - fio file write, read and overwrite using buffered IOs of random size between 128 KiB and 2 MiB

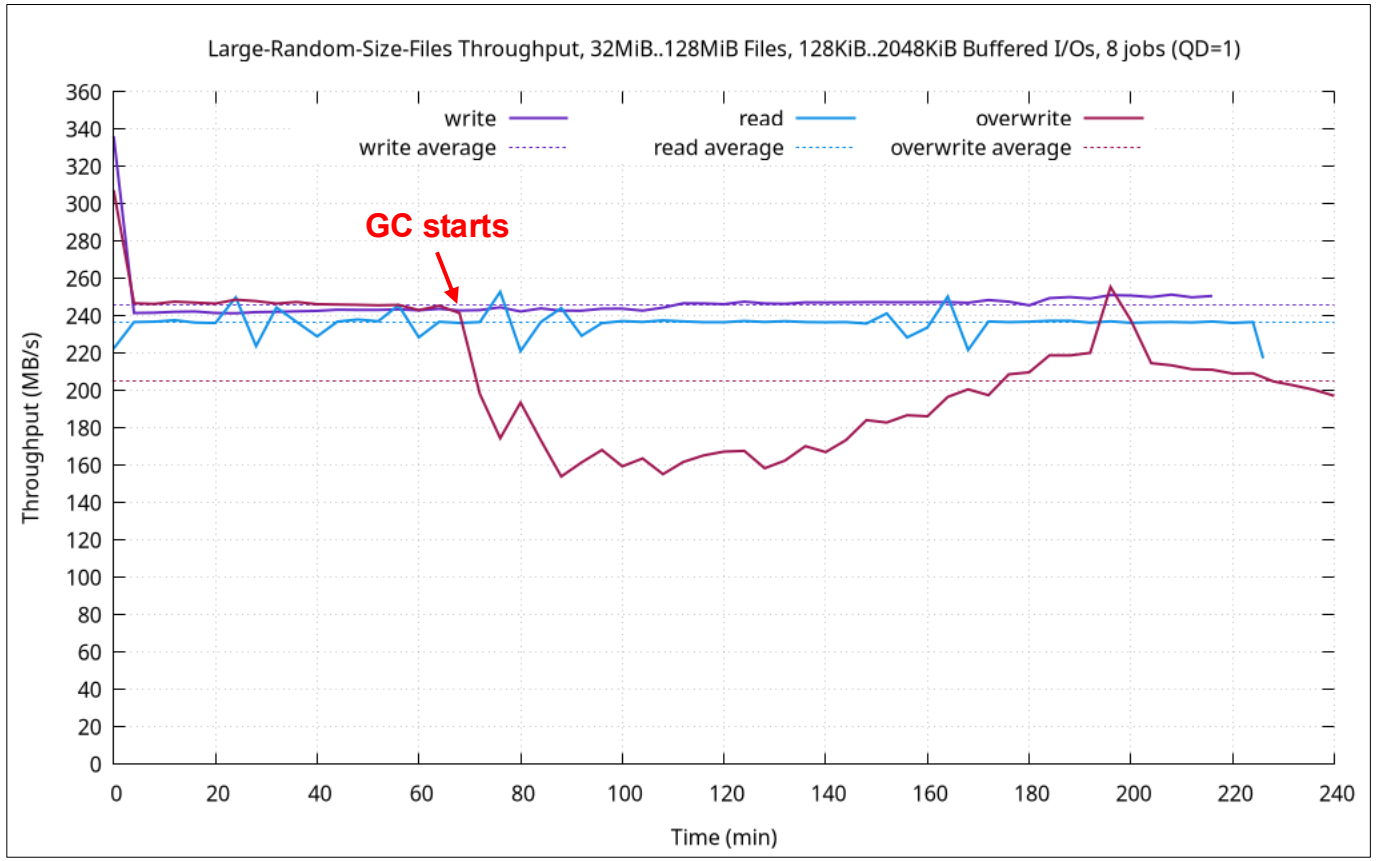
Large Fixed Size Files

- Ideal 256 MiB files case: each file is stored in exactly one zone
  - No file fragmentation: there is no GC overhead to reuse the disk space when files are deleted
  - Slight drop in throughput for overwrites compared to regular writes due to metadata handling and zone resets



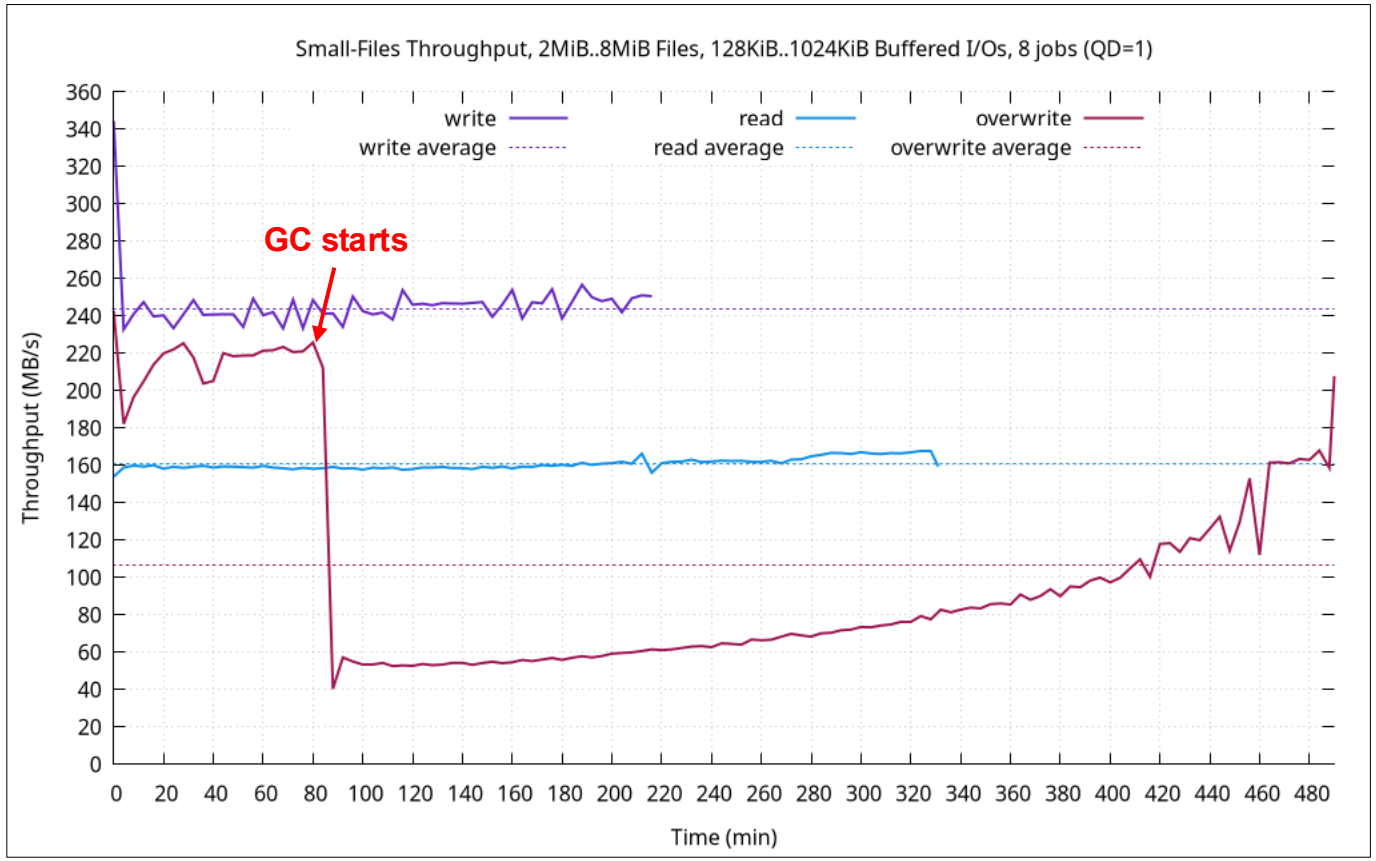
Large Random Size Files (32 MiB...128 MiB)

- Filling the file system is achieved at the maximum device write throughput
  - Random file read performance is also excellent
- However, overwriting randomly selected files leads to lower performance once GC triggers
  - Overhead of evacuating valid data blocks from zones that are being reclaimed



Small Random Size Files (2 MiB...8 MiB)

- Like with large files, filling the file system is achieved at the maximum device write throughput
  - Higher randomness of the read workload naturally leads to lower throughput
- But GC overhead on overwrite is much higher, resulting is significantly lower throughput
  - In average, compared to larger files, a higher number of valid data blocks needs to be evacuated from zones that are being reclaimed



Performance Summary

- XFS allows delivering excellent read and write performance with SMR HDDs
- For workloads that frequently delete files, large files lead to a much smaller garbage collection overhead
  - Almost no penalty with the ideal case of files using entire zones
- Workloads using small files or frequently modifying a small amount of data in files can suffer from significant performance degradation under GC conditions
  - Under heavy write workload and with the disk used at nearly full capacity

XFS on SMR: Status

- Support merged in Linux kernel v6.15 and xfsprogs 6.15
  - No longer experimental as of Linux 7.2
  - Major garbage collection performance improvement and file fragmentation fixes since initial merge
  - Slowly trickling into distributions (e.g. Fedora 43 and later)
- Full usage of disk capacity
  - Needs about 1% of capacity for metadata (workload dependent)
  - Requires a small fixed number of zones (4) reserved for GC
- Very stable and good performance
  - First large scale production customer backing a distributed object store
  - Under production evaluation at world leading companies and research institutes

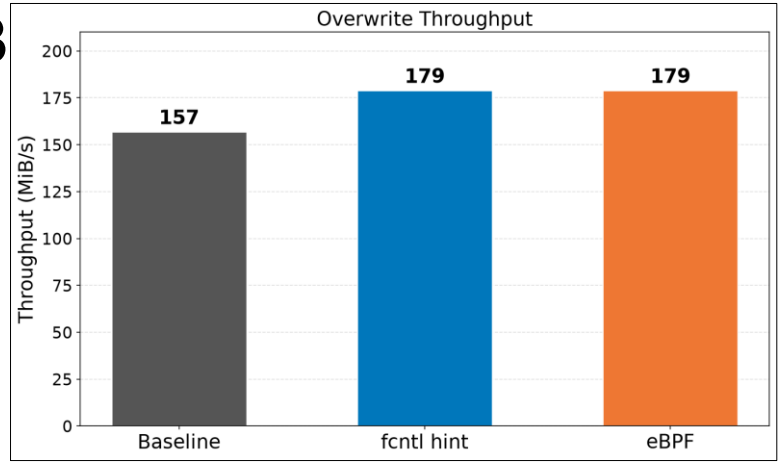
Upcoming features

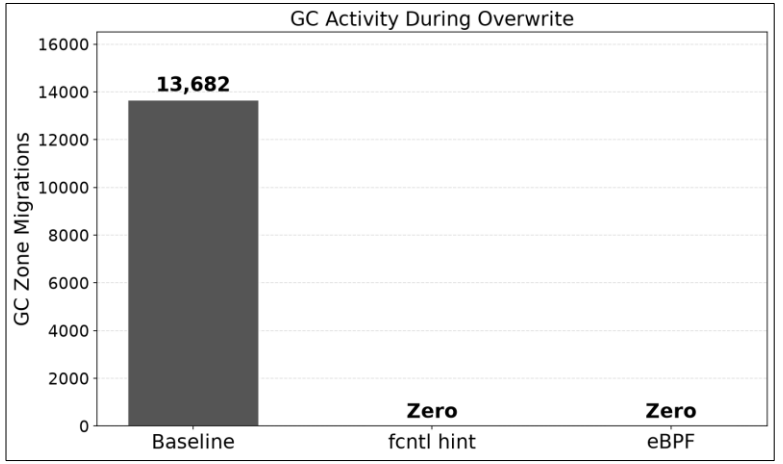
Data Placement: eBPF based Hints

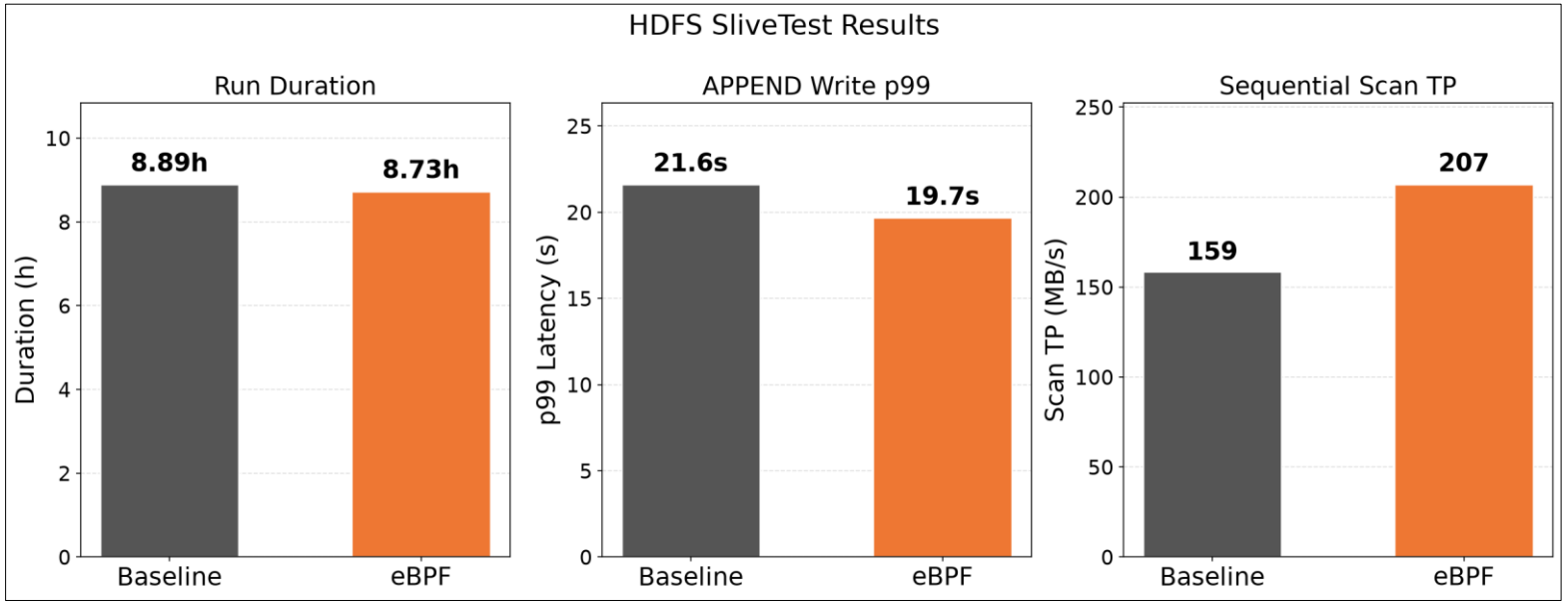
- Better data placement can reduce and even remove the need for expensive GC
  - But generic in-kernel data placement policies may not be optimal for all workloads
  - Missed opportunities for better performance
- XFS supports data placement hints through data temperature hints
  - F_SET_RW_HINT *fcntl()* system call provides excellent results
  - Application must be modified: cumbersome and workload dependent !
- Solution: **eBPF injection** for user-programmable data placement hints
  - Avoids direct application modification
  - Attached to the file system open method to enable data placement decisions based on:
    - *file name, file parent directory, process ID, cgroup*

Data Placement: eBPF Hints prototype results

- Micro benchmark: pairs of 256 MiB and 1 MiB files
  - Separating files by size reduces GC activity and leads to +14% overwrite throughput
- HDFS SliveTest
  - Separation of NameNode journal files, DataNode block files and block metadata files
  - ~9% decrease in latency with 30% increase in throughput
- More tests are on-going
  - Cassandra, RocksDB, ...





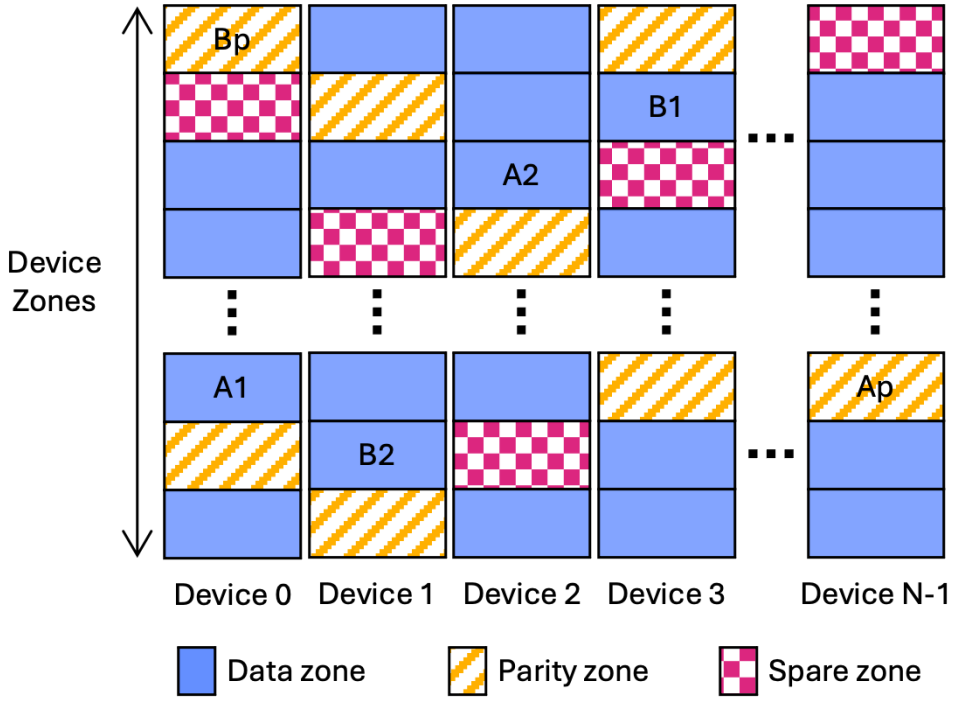


Zoned XFS Data Checksums

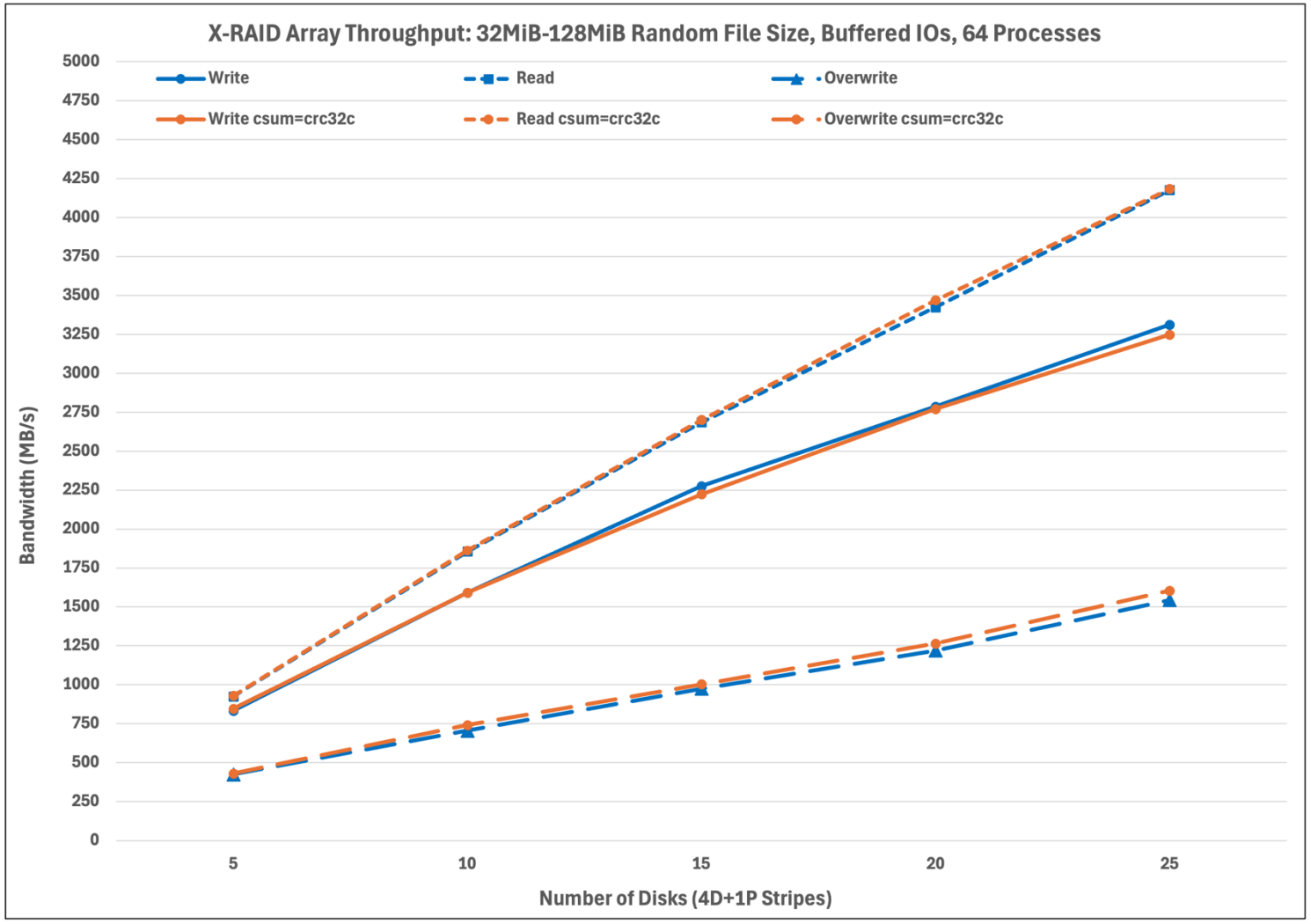
- Some use cases require end-to-end checksums
  - Protect against cable/HBAs faults
  - Error detection in distributed systems
- Checksums in the file system provide benefits over application checksums
  - Data placement
  - RAID repair
- RT checksum feature adds data checksums to zoned XFS
  - Easy to implement because of out of place writes
  - CRC32C or CRC64
- Exposes checksums to application (*ioctl*, *io_uring*)

XRAID: declustered parity RAID for Zoned XFS

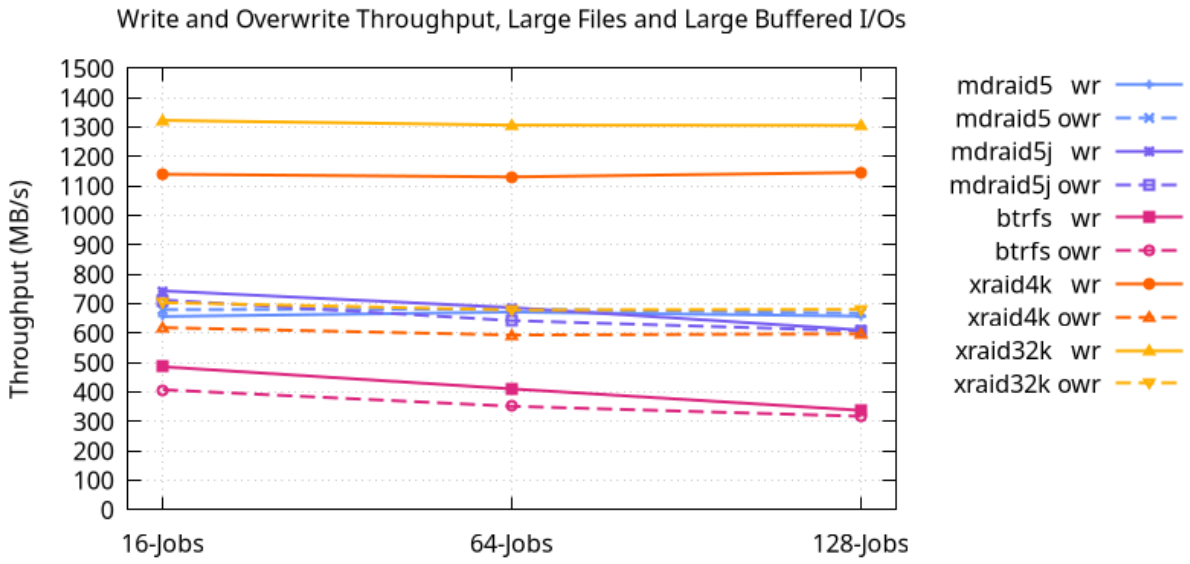
- Each RG maps to D+P data and parity zones
  - Declustered random placement to allow spreading over more than D+P disk *(think large enclosures)*
- No write hole due to out of place writes
- Partial parity is logged in the existing XFS log
  - Requires separata metadata device (*e.g. mirrored SSDs*)

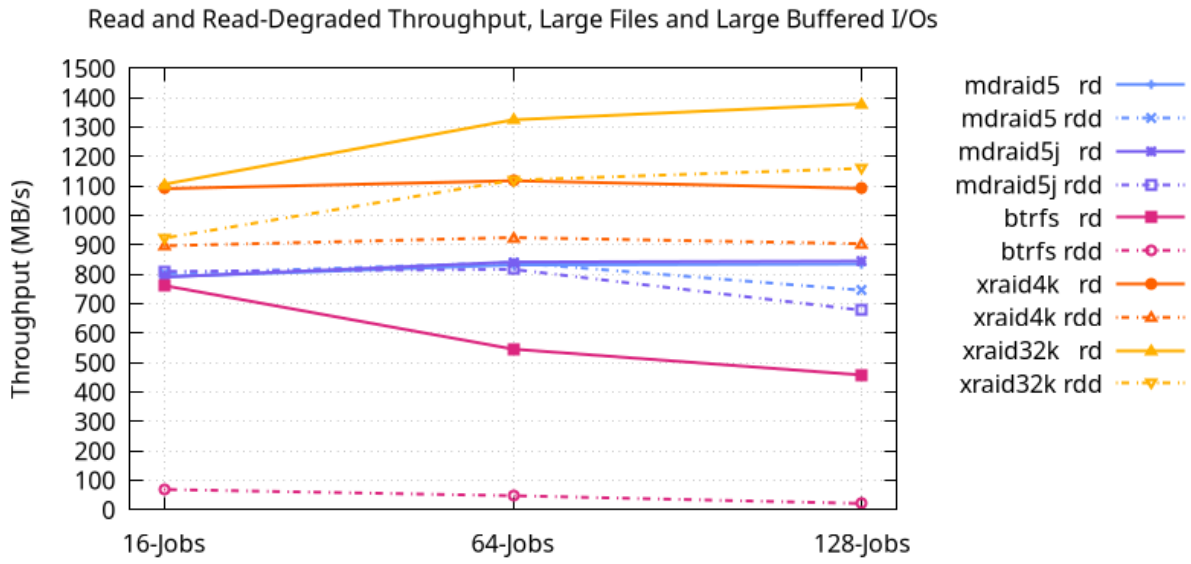


XRAID performance (SMR)

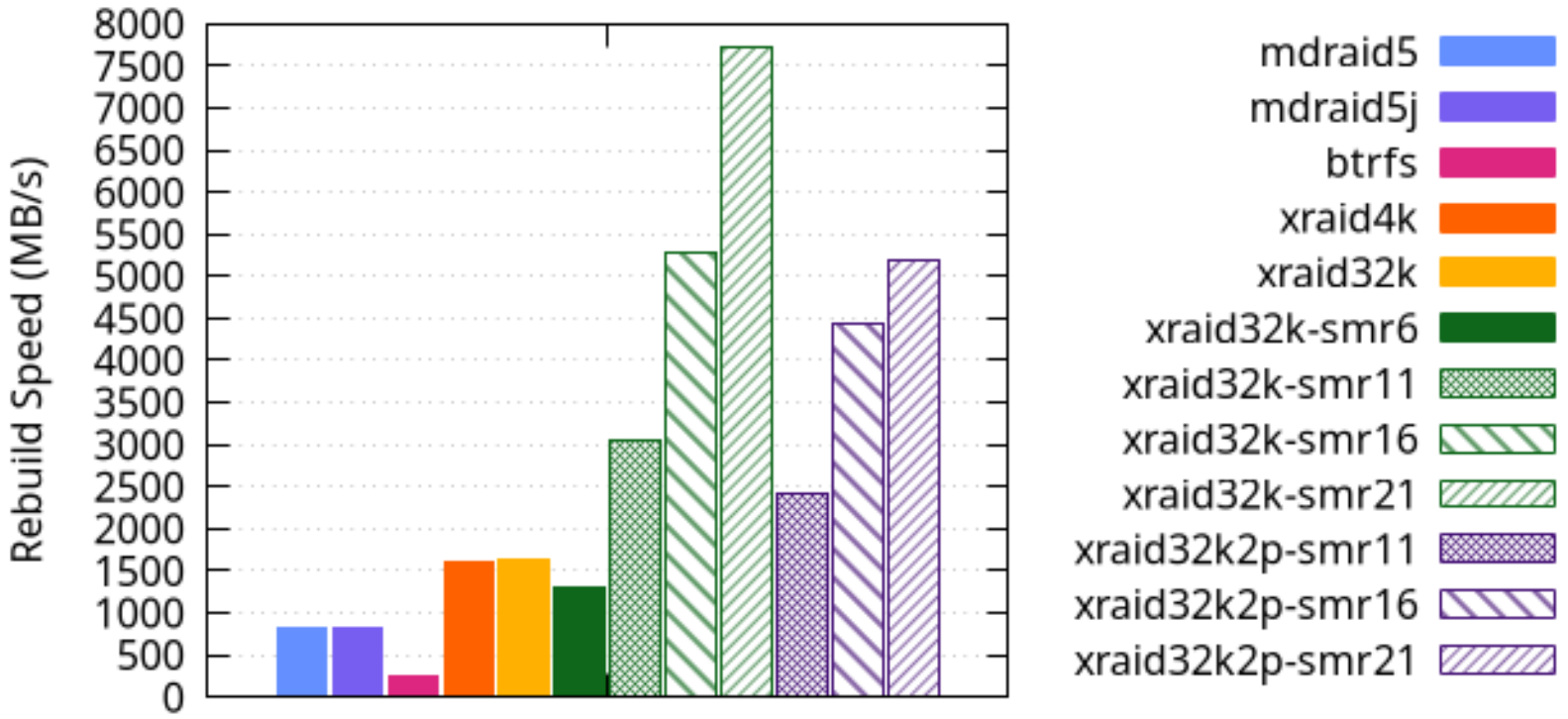


XRAID performance (CMR)





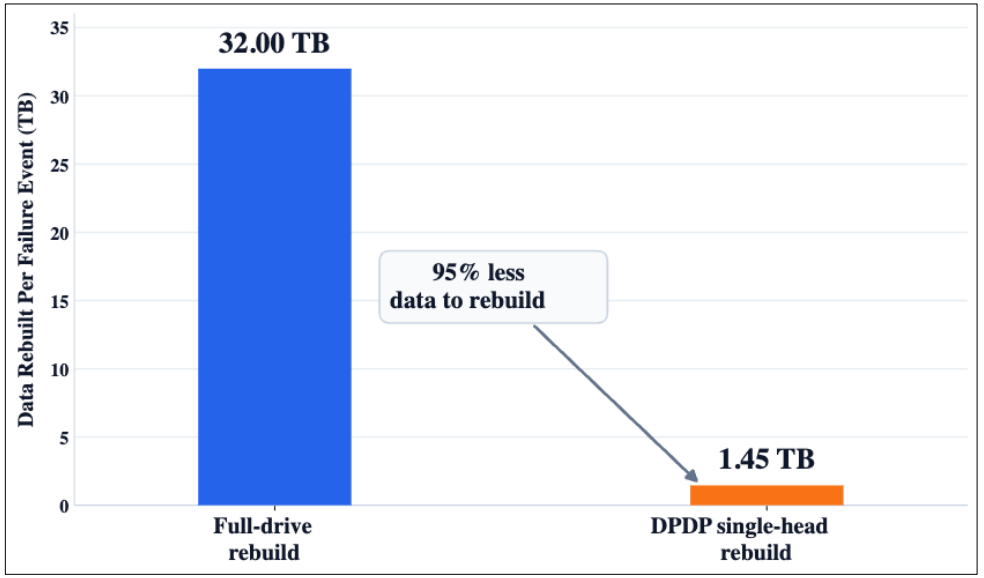
XRAID rebuild performance



Data Preserving Depopulation (DPDP)

- Disabling a failing head (head depopulation) enables keeping HDDs in the field longer
- SMR head depopulation preserves data
  - On a head depopulation event, REPORT ZONES shows affected zones with the “Offline” Zone Condition
  - Massive savings in rebuild time and background I/O, leading to gains in system durability
- Future-Proof
  - Mitigate reliability risk with head failures (e.g. in HAMR)
  - As platter count grows, DPDP advantage gets even bigger

Example: 32TB HDD, 11 disks, 22 heads



Data Preserving Depopulation Kernel Integration

- Lower level building blocks:
  - Depopulation feature discovery and depopulation command support in SCSI/ATA
    - First batch of patches already queued for 7.3, more to follow
  - Kernel API and user ioctl interface to inspect and trigger HDD head depopulation
    - In progress for 7.4
- RAID integration
  - XRAID tracks per-zone offline state, can issue depop and per-zone partial rebuild
- Data loss notification when no redundancy exists
  - Through fanotify *FAN_ERROR* event, or XFS healthmon
  - XFS translates physical blocks lost to file+offset using reverse mapping tree

Questions?

More information

- **Zonedstorage.io** *XFS File System:* https://zonedstorage.io/docs/filesystems/xfs
- **Apsys2025:** *Staying in the Zone - Retrofitting Zoned Storage into a Scalable Enterprise File System* https://dl.acm.org/doi/10.1145/3725783.3764399)
- **SNIA SDC 2025:** *Zoned XFS: Transparent and Efficient Zoned Storage Support For Scalable Storage Systems* *https://www.snia.org/sniadeveloper/session/19297*
- **FOSSY 2026:** *RAID sucks (and what we can do about it)* *http://verein.lst.de/~hch/raid-sucks.pdf*