STORAGE DEVELOPER CONFERENCE

SDC 23

BY Developers FOR Developers

Bridging the Gap Between Host Managed SMR Drives and Software-Defined Storage

leil STORAGE

Presented by
Piotr Modrzyk

Speaker

Piotr
Modrzyk

Principal Architect at Leil Storage

X-googler and Creator of LizardFS

Outline

- Brief intro to SaunaFS
  - Simplified SaunaFS architecture
  - Chunks
- SMR restrictions
  - Problems for conventional Chunks
- Solution
  - Divide the Chunks into Metadata and Data
  - Handle non-sequential writes: fragment the chunks & garbage collection
- SMR libraries overview and why ZoneFS
- Testing framework extension for zoned devices
- Inspecting the content of zones with a graphical UI tool

Brief intro to SaunaFS

- SaunaFS is a Distributed File System written mostly in C++ which implements concepts introduced by Google File System.

- SaunaFS is divided into:

  — Metadata Servers (master, shadows and metaloggers)

  — Data Servers (chunkservers)

  — Clients (native Linux/Windows, NFS)

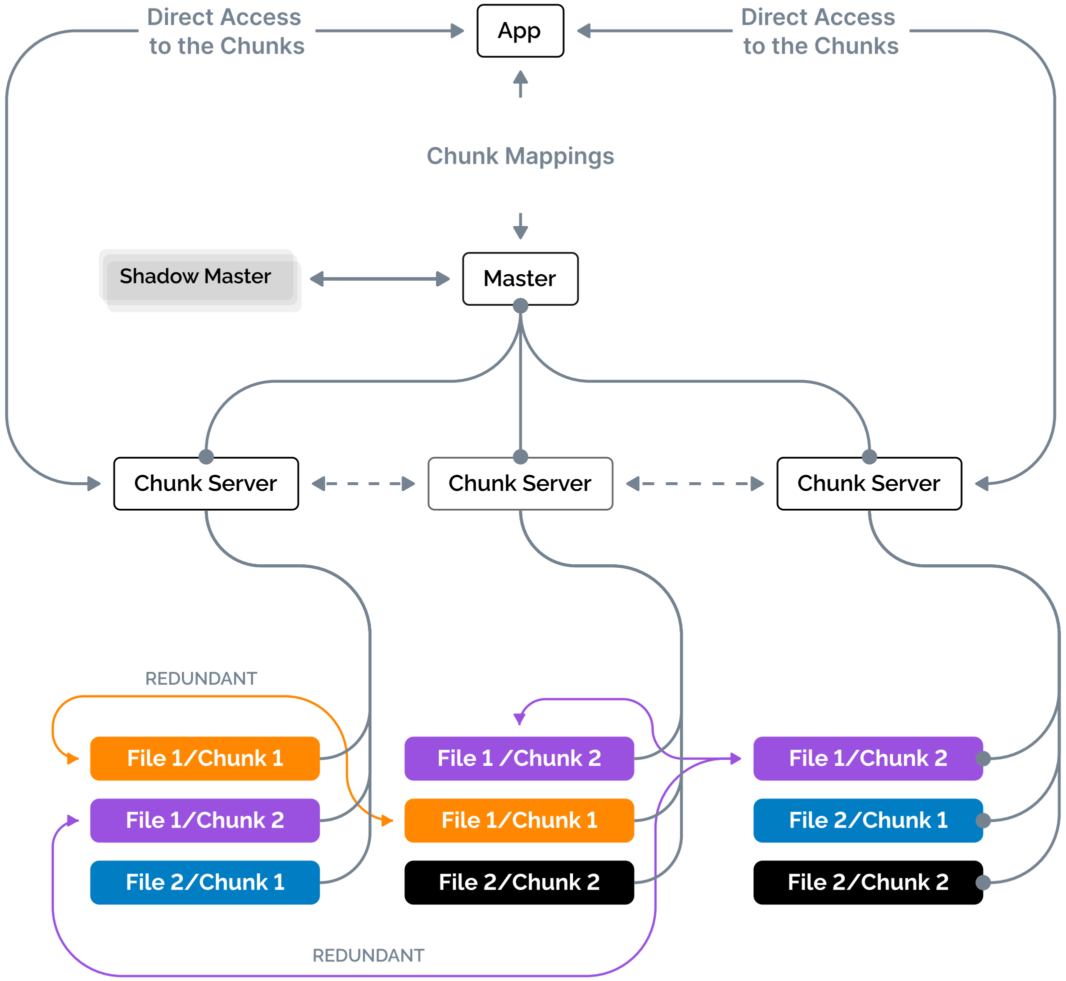
- In the Chunkserver side:

  — Files are divided into Chunks (up to 64 MiB)

  (chunks are logically divided into **Blocks of 64 KiB**,

  which is the minimum block size)

For each block, 4 bytes of CRC are also stored in the Chunk metadata.



See https://en.wikipedia.org/wiki/Google_File_System

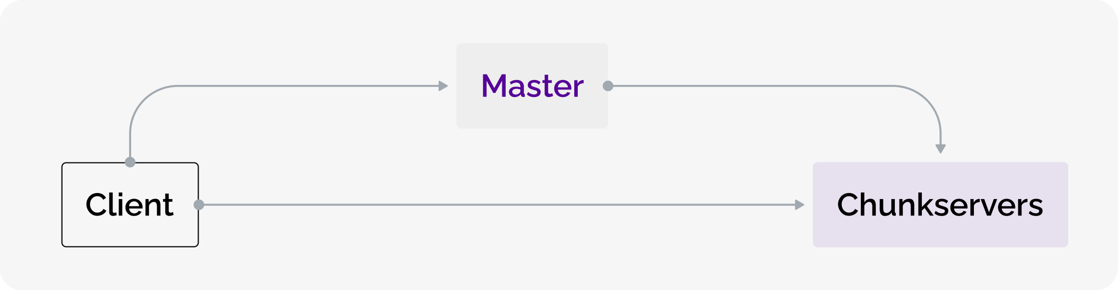
Simplified writing process

The client wants to create a file and to write data to this file:

- The client asks the Master server where (Chunkservers) to put the first chunk.

- The client connects directly to the Chunkservers and starts sending the data in Blocks of 64 KiB + 4 B of CRC.

- The Chunkservers check the Block’s CRC against the received data and write to the storage devices, metadata is updated if needed.



EC4+2 file with 2 Chunks Example

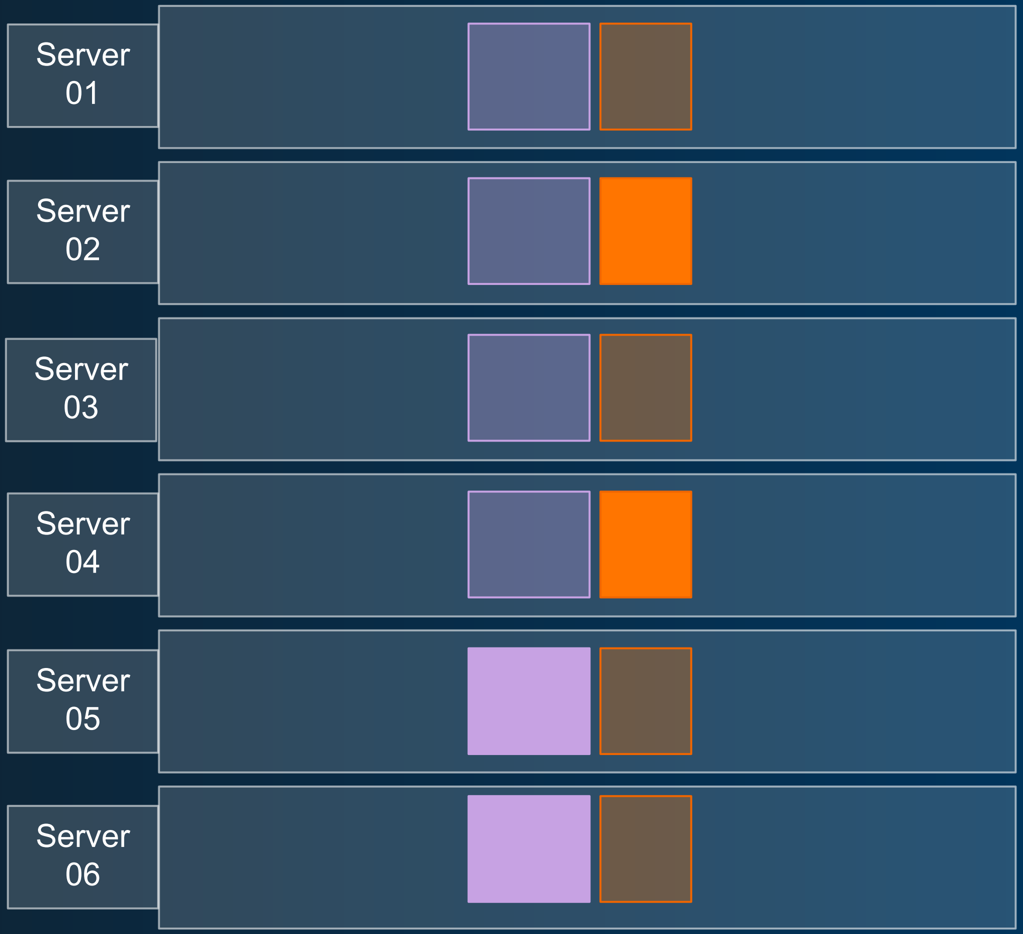
- File_01_100_MiB_goal_ec42.dat
  - The Chunks will be divided into 4 data parts containing up to 16 MiB of data = 4 x 16 + 4 x 9MiB = 100MiB of DATA:

16 MiB	16 MiB	16 MiB	16 MiB	9 MiB	9 MiB	9 MiB	9 MiB

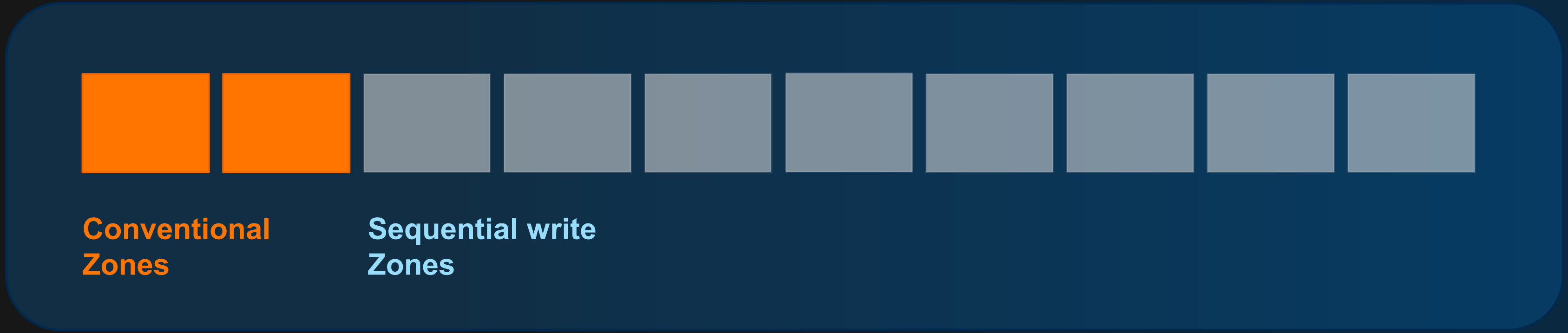
64MiB + **36MiB**

  - Two PARITY for every 4 pieces of data, with same size will be created:

16 MiB	16 MiB	9 MiB	9 MiB



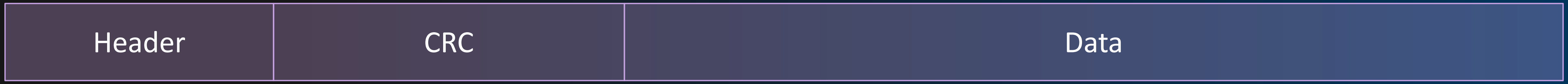
SMR restrictions



The client wants to create a file and to write data to this file:

- The Sequential Zones can only be appended at the write head.
- The IO operations must be aligned to the device IO block size (usually 4KiB).

Conventional chunks



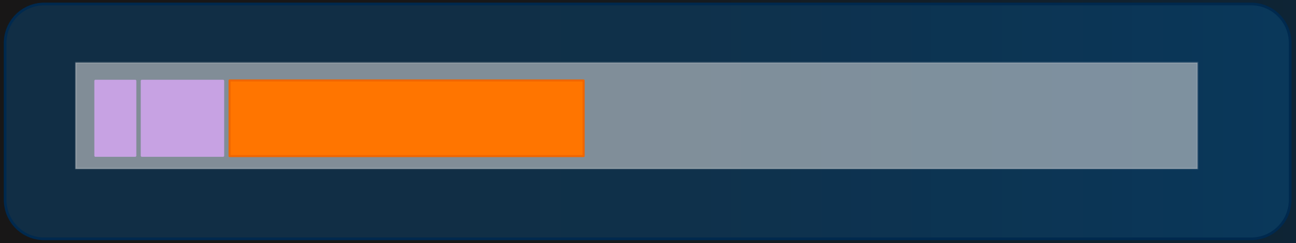
Header

CRC

Data

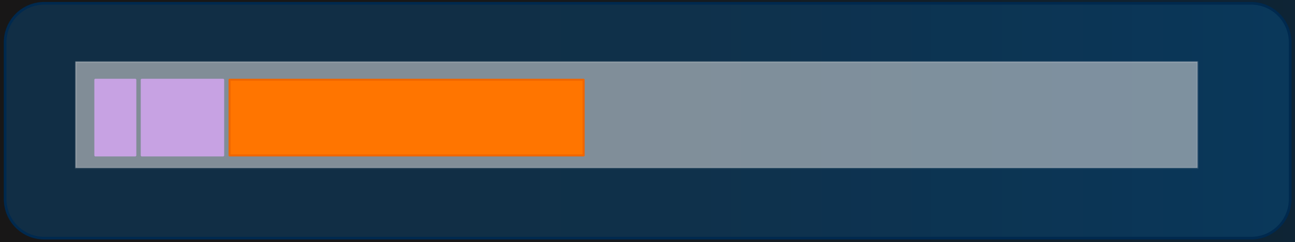
- Header 1 KiB
  - Id, version, type.
- CRC
  - Up to 1024 Blocks of 4B.
- Data
  - Up to 1024 Blocks of 64 KiB.

Problems for Conventional Chunks

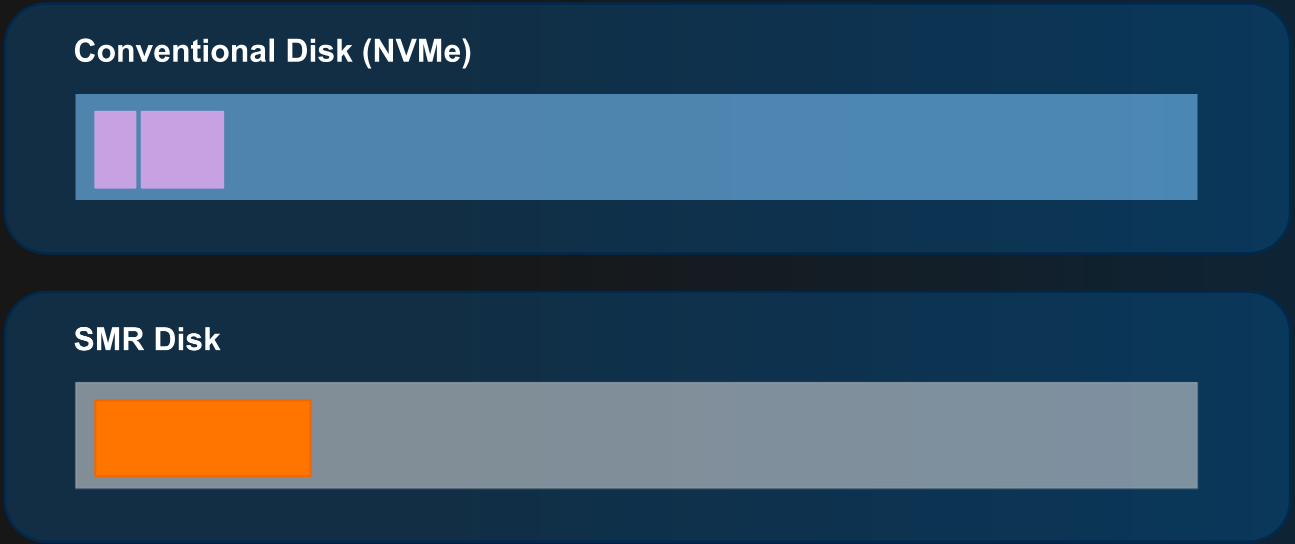


- CRC must be updated with each Block write, which implies non-sequential writes to the Zone.
- **Header + CRC = 5 KiB,** which is not aligned to the 4 KiB IO block size of many SMRs.
- The Zone write head is always **moved by 64 KiB**, which only works for write block sizes **multiple of 64 KiB.**

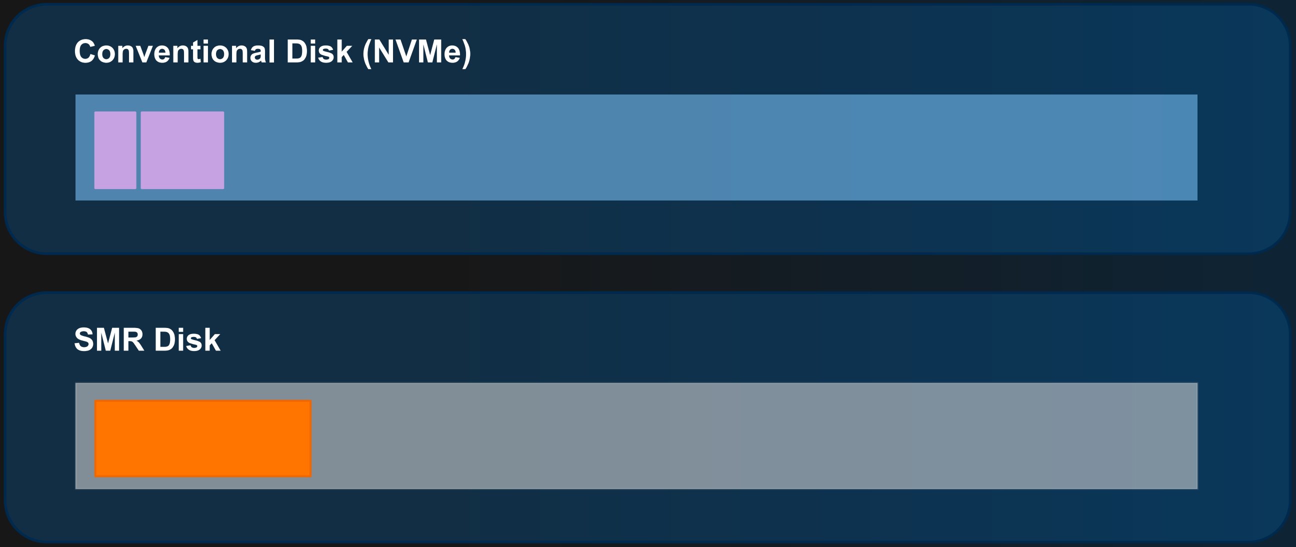
Solution: Divide Chunks into Metadata & Data



Split the Chunks into Metadata and Data.



Solution: Divide Chunks into Metadata & Data



- The metadata is now in another (NVMe) disk, which eliminates the problem of writing the CRC in Sequential Zones.
- Data can be aligned now into the Zones with **64 KiB Blocks.**
- **The Zone Write Head is always moved by 64 KiB, which only works for write block sizes multiple of 64 KiB.**

Solution: Handle Non-Sequential Writes

Introduce Chunk fragmentation.

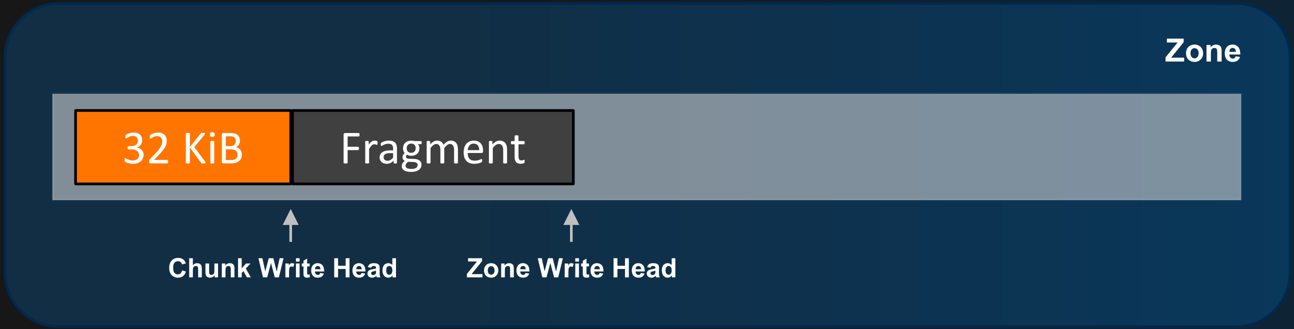
Header	CRC	Data

- The Header is modified to contain information about the Fragments:
  - Id, version, type**, number of fragments, list of fragments.**
- Metadata about the Fragments contain 12Bytes each:
  - Zone(4B), offsetInZone(4B), first block(2B), number of Blocks(2B).
- New Fragments of same chunk are preferred to be stored in the same Zone if possible.
- A Chunk with more than one Fragment is considered fragmented.

Solution: Handle Non-Sequential Writes

Example of non-sequential write into the Zones:

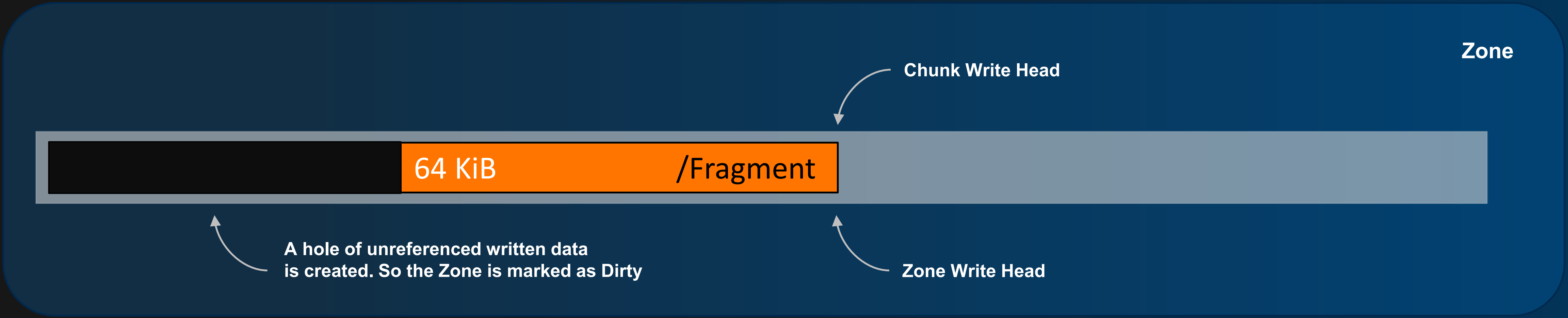
- Create a file and write 32 KiB.
- A new Chunk is created with 1 Fragment containing 1 Block of 64 KiB, but only 32 KiB belongs to the file.



- The next bytes to write will trigger a non-sequential write into the Zone.

Solution: Handle Non-Sequential Writes

If the Fragment contains only one block, we can reuse the Fragment and update the location (same or different Zone).

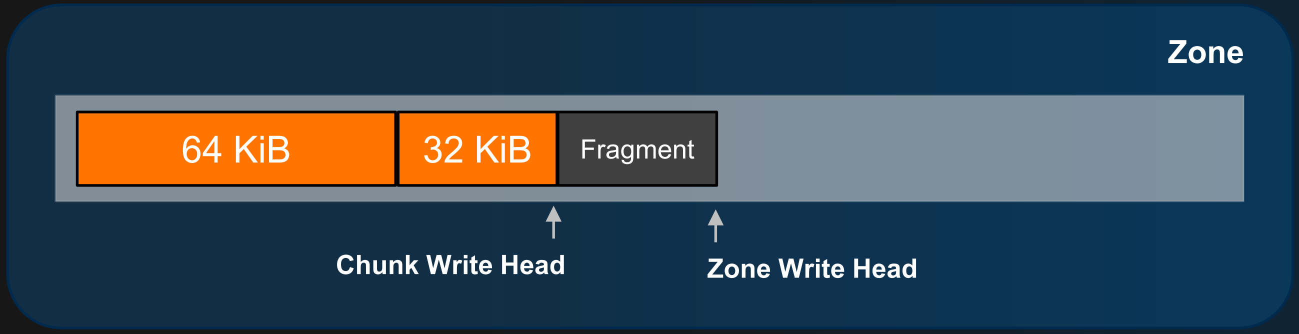


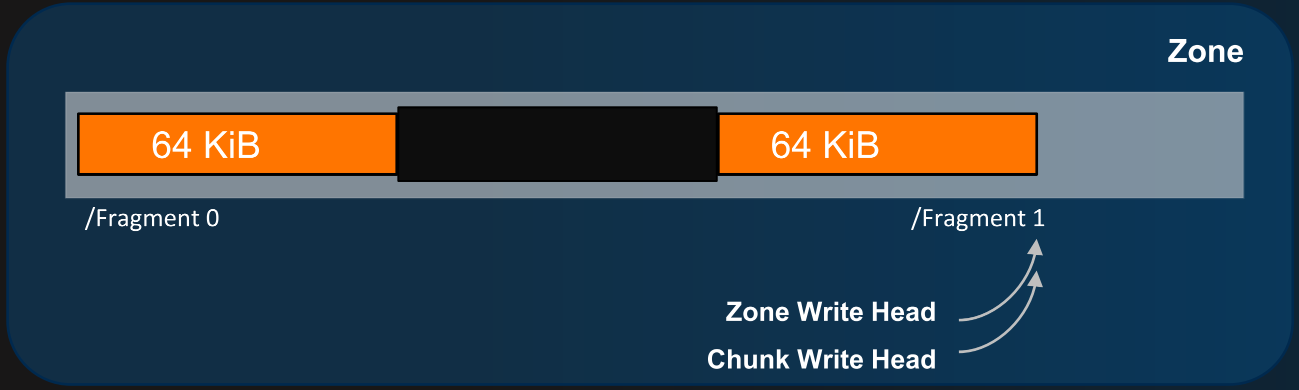
A hole of unreferenced written data is created. So the Zone is marked as Dirty

Note: the Chunk is still not fragmented.

Solution: Handle Non-Sequential Writes

If the Fragment contains more than one block, we need to create a new Fragment, preferably in the same Zone.





A hole of unreferenced written data is created, and the Zone is marked as Dirty. The Chunk is now considered fragmented.

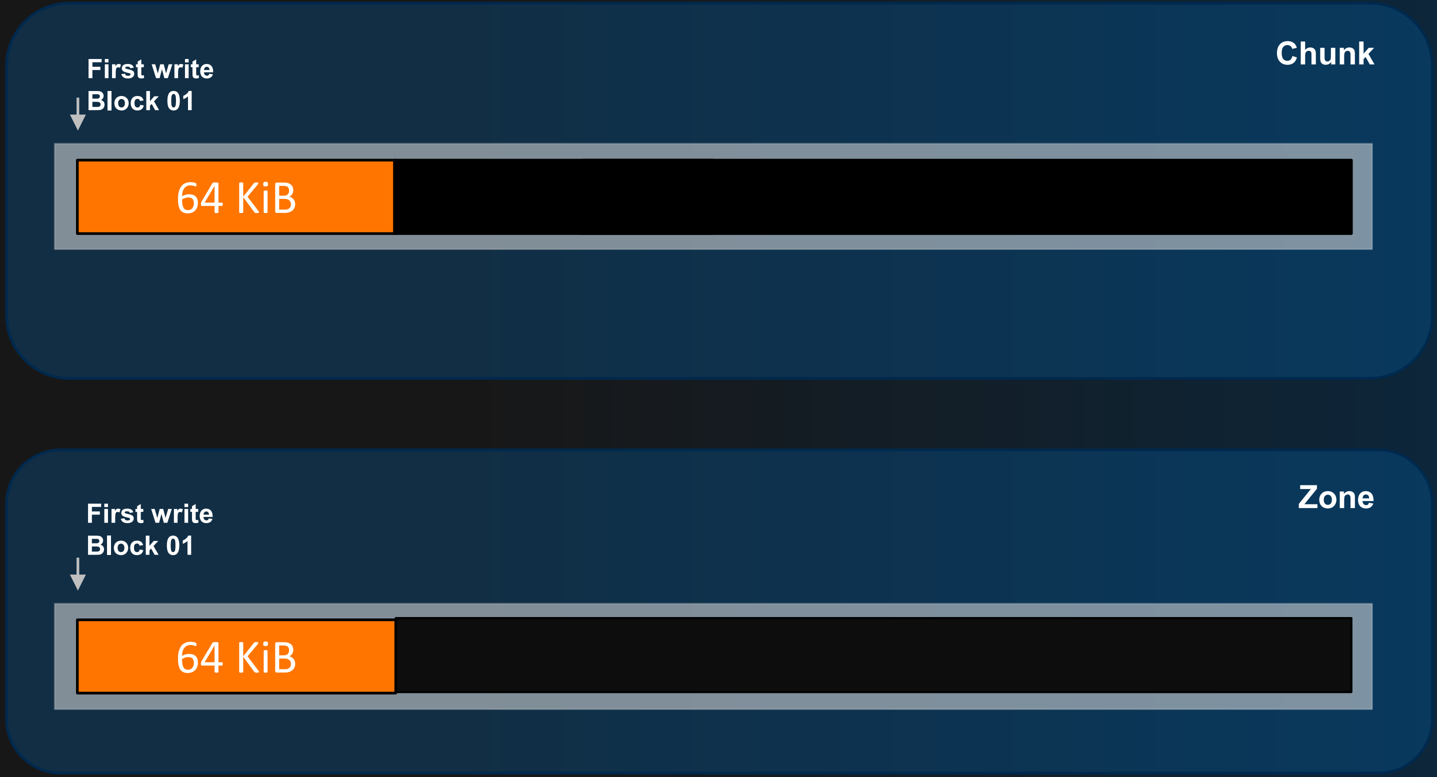
Solution: Handle Random Writes

Random write:

```
fio --name=fiotest_rand_write_QD5 --directory=/mnt/saunafs --size=1G
--rw=randwrite --numjobs=1 --ioengine=libaio --group_reporting --bs=8M
```

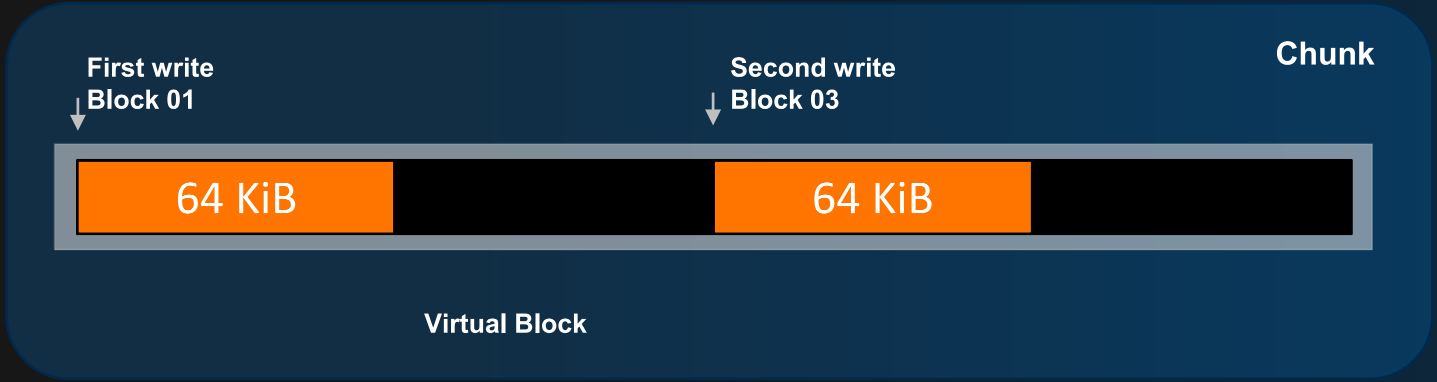
Solution: Handle Random Writes

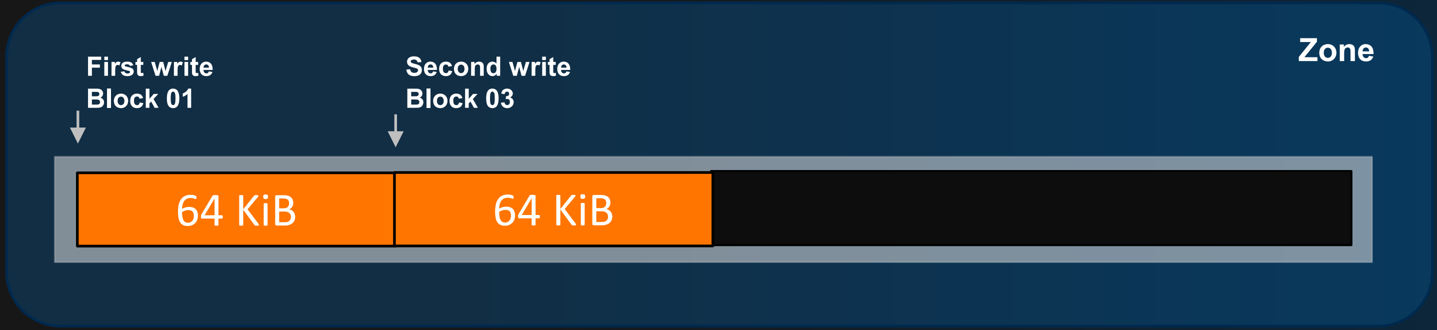
Random write:



Solution: Handle Random Writes

Random write:



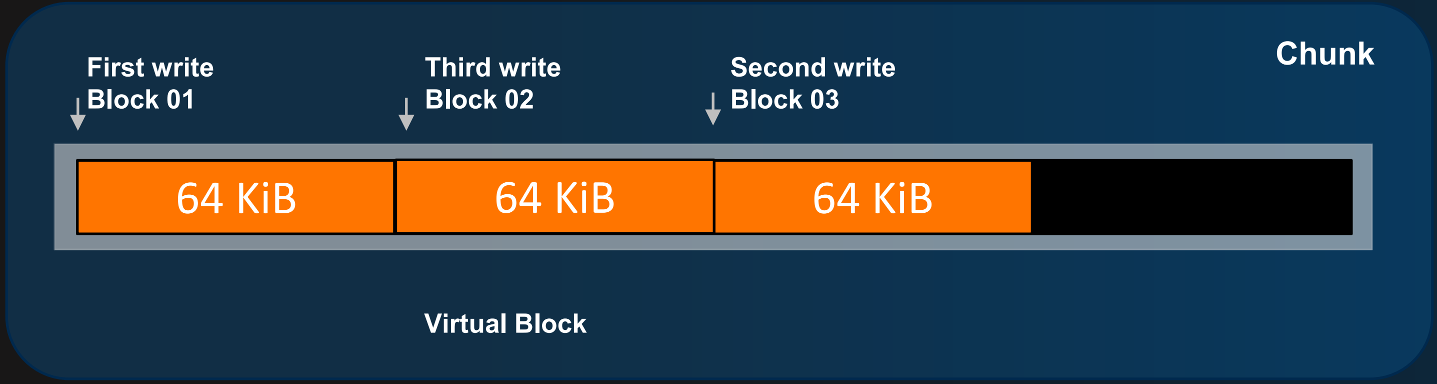


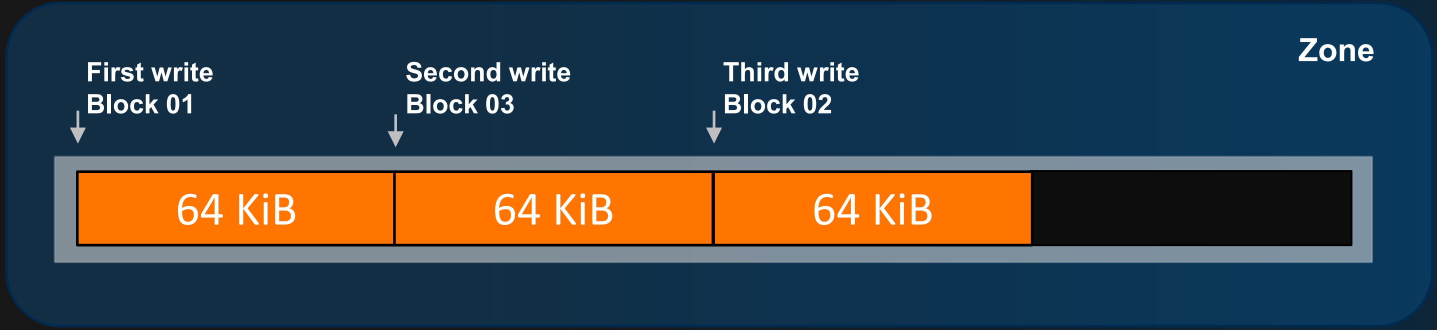
Notice the incorrect order of the blocks in the zone.

The order will be fixed during defragmentation.

Solution: Handle Random Writes

Random write:





Notice the incorrect order of the blocks in the zone.

The order will be fixed during defragmentation.

Solution: Handle Random Writes

Random write:

- Since we have Virtual Blocks now, defragmentation should be fragment-based instead of the block-based.
  This way, we can avoid creating unnecessary Blocks full of zeroes each time we would need to deal with virtual block (full of nulls).

- Chunk testing can still be per block, Virtual Blocks will return zeros.

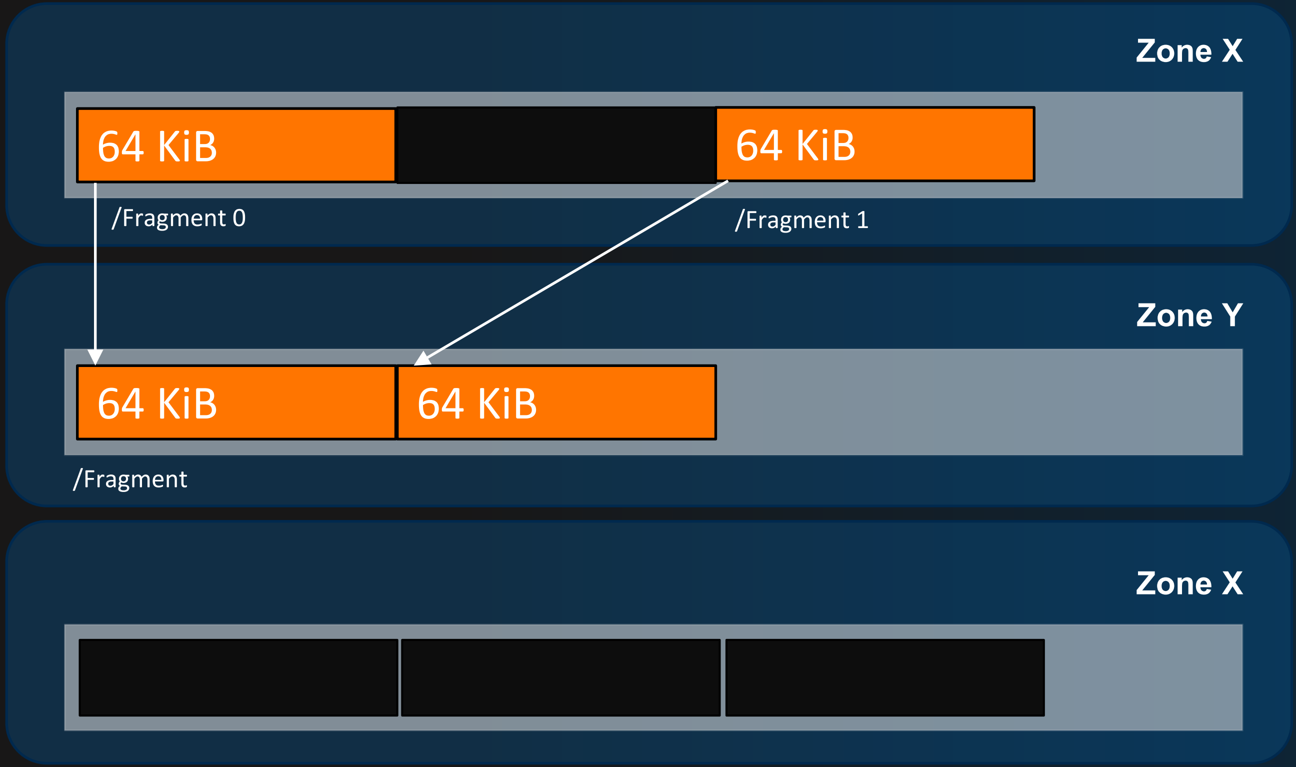
Solution: Garbage Collection

Garbage collection is divided into:

- Defragment the Chunks.
- Reset unreferenced Dirty Zones.

Solution: Chunks Defragmentation in chunk-test thread

Defragment the Chunks by extending our test-chunk thread, with defragmentation task.



The Zone X can be reset now, because it does not contain any valid data.

Garbage Collection Algorithm Review

Issues with GC during chunk-test thread algorithm:

- Depending on the speed of the Chunk Testing Thread, which can be testing millions of Chunks (and most of them may not be fragmented).
- The Zones are only reset if no more Chunks are referencing them, and the random nature of the chunk-test thread is selecting chunks from different (random) zones.

New Garbage Collection Algorithm is needed

Goals for the new approach:

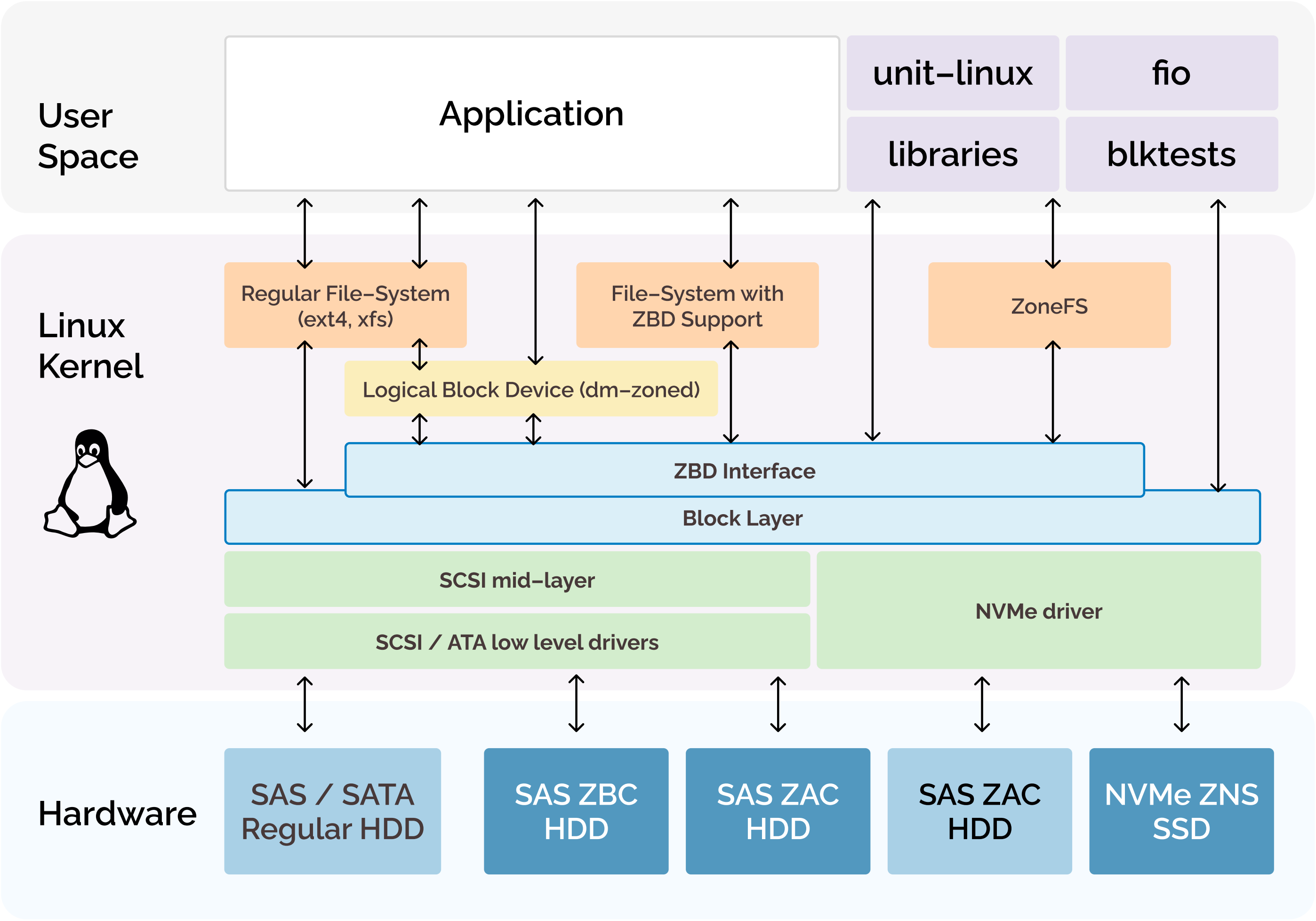
- Control the speed of the cleaning process.
- Prioritize the defragmentation of Chunks belonging to the same Zone, to maximize the Zone resets (faster reclaiming of the unreferenced space).
- Reduce the number of Dirty Zones.

Available SMR libraries/tools

libzbc	libzbd	ZoneFS
Provides functions for manipulating ZBC and ZAC disks directly.	Provides functions for manipulating zoned block devices (uses the kernel-provided ZBD interface that is based on the ioctl() system calls).	Exposes the zones as files (from kernel 5.6.0). Uses `mkzonefs` to format the drive and then `mount -t zonefs`. Provides aggregation for conventional zones, file ownership and file access permissions.
Contains an emulation mode to mimic HM zoned devices.	**No** (but null_blk can be used).	**No** (but null_blk can be used).
Graphical Interface: **gzbc**.	Graphical Interface: **gzbd**.	**No** (gzbc and gzbd works).

SMR libraries overview

Based on illustration from: https://zonedstorage.io/docs/getting-started



ZoneFS usage

- At Chunkserver start-up.
  - Fill a ZoneFSDisk class which contains all the Zones.

    The minimal information about the Zones are **number, type, and writeHead.**

- Every time a Zone is modified, some extra in-memory information is updated:
  - **isDirty:** the boolean attribute returns true if the Zone contains written data not referenced by any Chunk Fragment.
  - **blocks:** number of Blocks referenced by Chunks.
  - The next Zone to be chosen is also updated.

Why ZoneFS?

BUILT-IN IN MAINSTREAM KERNELS

NO NEW DEPENDENCIES FOR THE PROJECT

like ZBC or ZBD.

ALLOWS USAGE OF FAMILIAR FILE IO MODEL

which means less modifications to the current Chunkserver code.

ZoneFS usage

How to select the next Zone to write:

- The first element of a **std::set** will be the next Zone.
- Dirty Zones are penalised to be chosen as last resource.
- Available space is the next field used to decide.
- Zones with more available space are preferred.

SaunaFS tries to append new fragments into the same Zone, in order to:

- Avoid increasing Dirty Zones during Chunk defragmentation.
- Reduce the Zones to be open and close at reading or writing.

ZoneFS usage

Sequential Zones can only be written if opened with

O_DIRECT flag

O_DIRECT flag also implies that in-memory buffers for pread or pwrite must be properly aligned with the device IO block size (memalign family of functions).

Testing framework

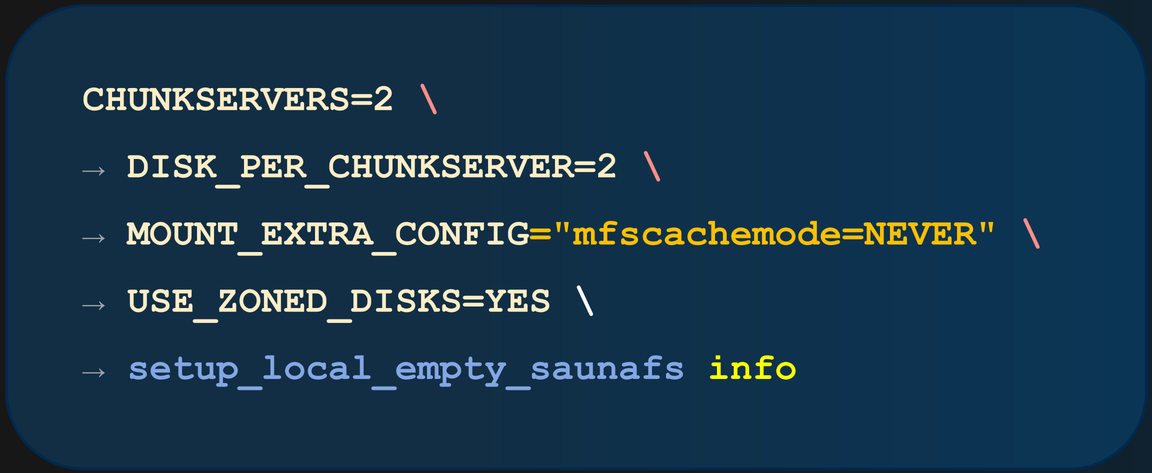
- SaunaFS contains a strong testing framework base on Google Test and bash.
- The tests run in a kind of sandbox inside /tmp and all the data is removed after each test execution.
- The data for conventional disks is stored in a RAM disk or in loop devices (always mounted).

The testing framework is able to:

- Create and run on demand the needed master, shadows, metaloggers, mounts, chunkservers and other tools implemented specifically for tests (file-generate, file-validate, etc).
- Stop, restart and start them at any moment and wait for them to be ready.
- Automatic clean-up after each test execution.

SMR driven modifications to our test framework

- The **null_blk** driver was selected to emulate the zoned devices.
- The emulated drives are created on demand and destroyed after each test execution.
- The testing framework was extended to accept the number of Conventional and Sequential Zones, the block size, and the zone size.



```
CHUNKSERVERS=2 \
→ DISK_PER_CHUNKSERVER=2 \
→ MOUNT_EXTRA_CONFIG="mfscachemode=NEVER" \
→ USE_ZONED_DISKS=YES \
→ setup_local_empty_saunafs info
```

Creates 2 Chunk servers with 2 emulated SMR drives each one.

SMR driven modifications to our test framework

Besides the 300+ of standard tests, we have 86 SMR integration tests related to:

- Write, read and overwrite sequential and random data in parallel.
- Multiple chunk truncation.
- Concurrent RW to the same zone.
- Sparse chunks.
- Snapshots.
- File descriptors leak check.
- Disk failures during write and read.
- Valgrind.
- CRC errors detection and fixing.
- Chunk versioning.

Graphical user interface (GUI)

- **libzbc** and **libzbd** provide a simple GUI which allows to visually represent the used space in Sequential Zones (based on the write head).

- The used space in conventional zones is not represented because the write head for this type of zone is always the zone size.

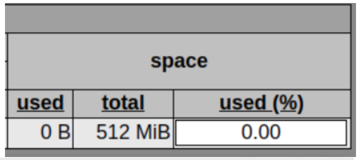
- We have crated our own GUI to highlight the data inside of the zones belonging to different chunks and to visualise our logical write heads for conventional zones.

Graphical user interface (GUI)

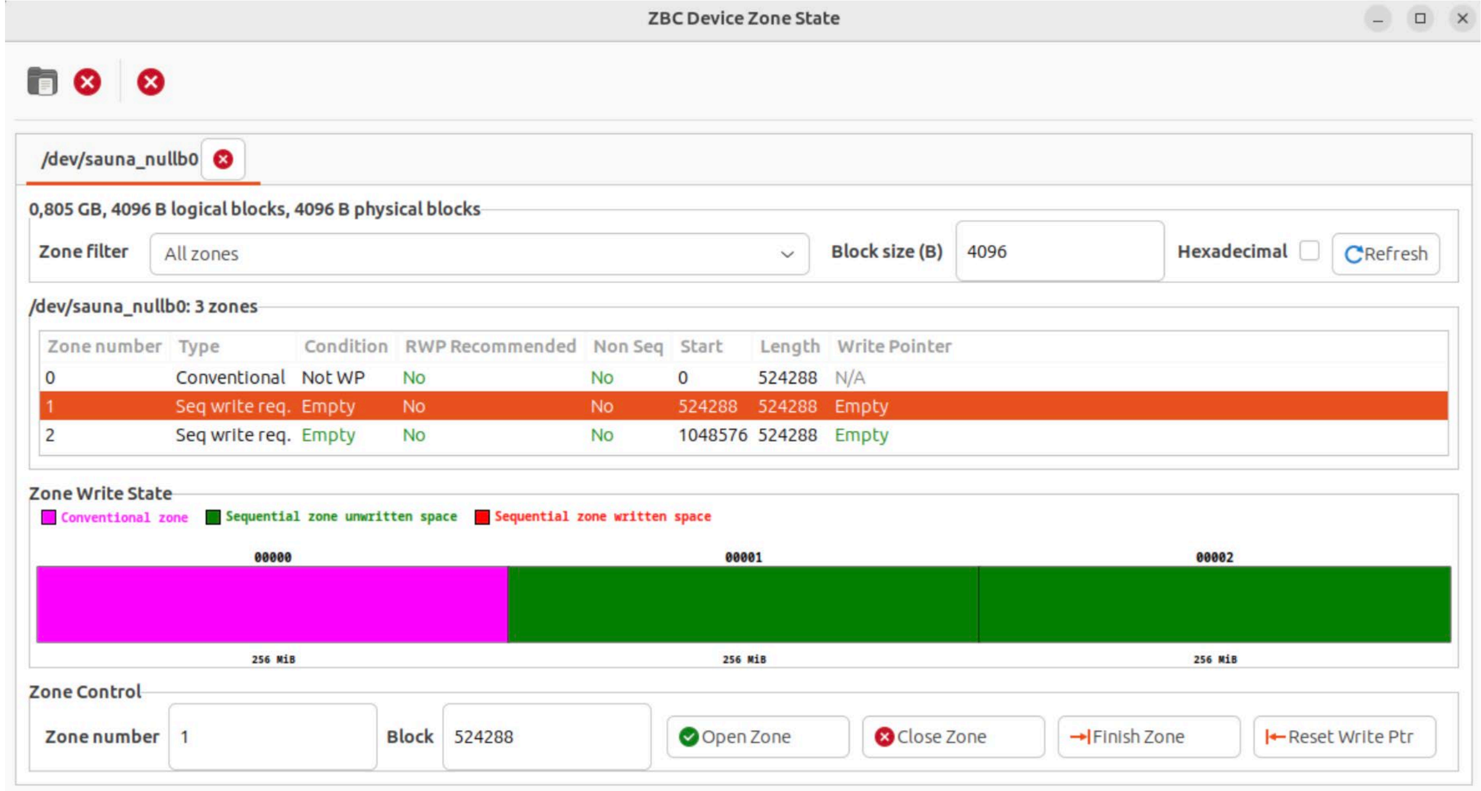
For simplicity, let's use a null_blk emulated drive with:

- Zone size: **256 MiB**
- Conventional Zones: **1**
- Sequential Zones: **2**
- Usable size: **512 MiB**

SaunaFS monitoring



space		
used	total	used (%)
0 B	512 MiB	0.00



ZBC Device Zone State

/dev/sauna_nullb0

0,805 GB, 4096 B logical blocks, 4096 B physical blocks

Zone filter: All zones — Block size (B): 4096 — Hexadecimal □ — Refresh

/dev/sauna_nullb0: 3 zones

Zone number	Type	Condition	RWP Recommended	Non Seq	Start	Length	Write Pointer
0	Conventional	Not WP	No	No	0	524288	N/A
1	Seq write req.	Empty	No	No	524288	524288	Empty
2	Seq write req.	Empty	No	No	1048576	524288	Empty

Zone Write State

🟪 Conventional zone 🟩 Sequential zone unwritten space 🟥 Sequential zone written space

Zone Control

Zone number: 1 — Block: 524288

gzbc

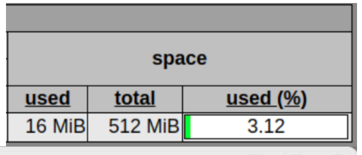
Graphical user interface (GUI)

Write 16 MiB:

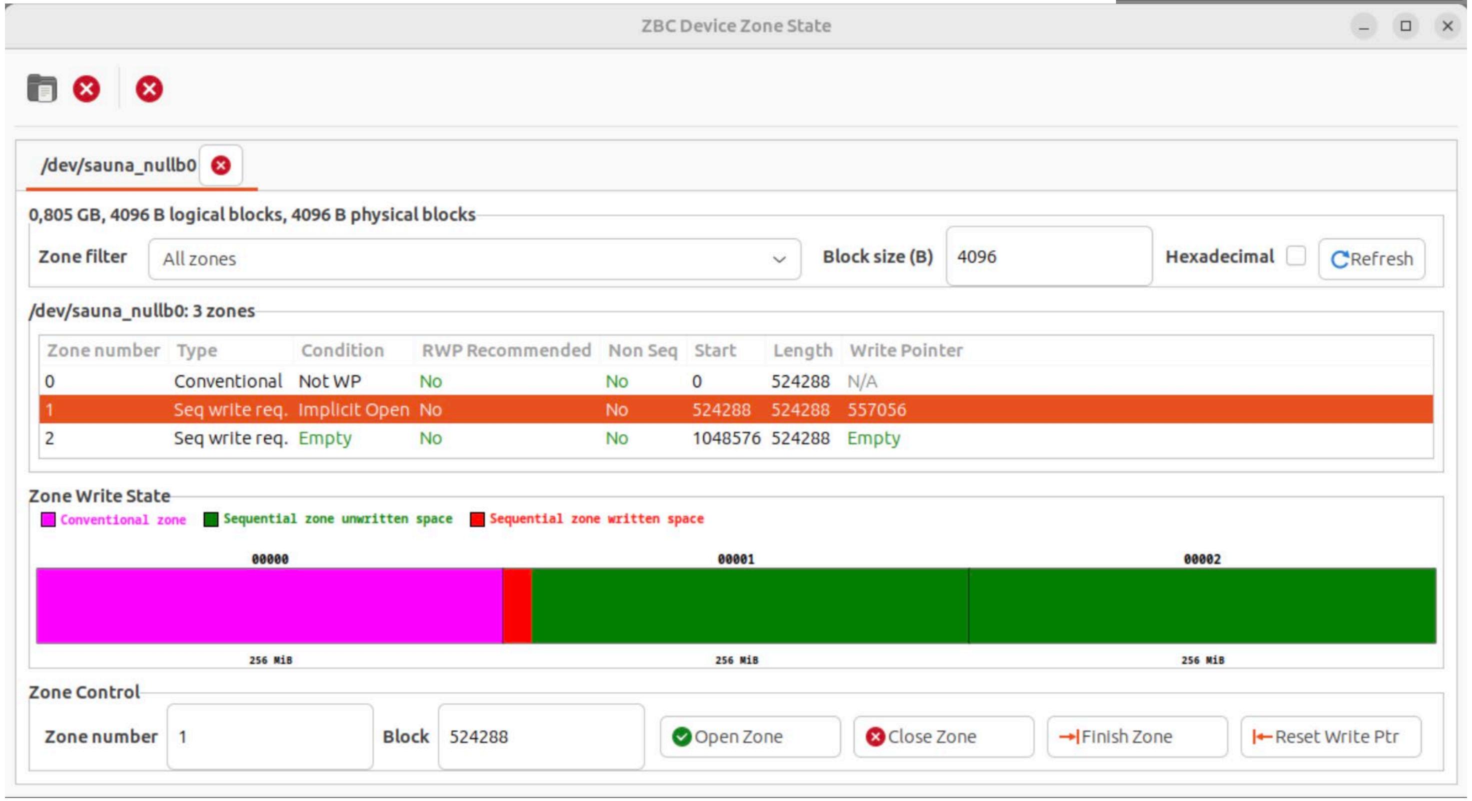
```
fio --name=file01 --
directory=/mnt/saunafs --
size=16M --rw=write --bs=64K
```

- The data is written to the Sequential Zone 1 (represented in red).

SaunaFS monitoring



space		
used	total	used (%)
16 MiB	512 MiB	3.12



/dev/sauna_nullb0

0,805 GB, 4096 B logical blocks, 4096 B physical blocks

Zone filter: All zones
Block size (B): 4096
Hexadecimal □

/dev/sauna_nullb0: 3 zones

Zone number	Type	Condition	RWP Recommended	Non Seq	Start	Length	Write Pointer
0	Conventional	Not WP	No	No	0	524288	N/A
1	Seq write req.	Implicit Open	No	No	524288	524288	557056
2	Seq write req.	Empty	No	No	1048576	524288	Empty

Zone Write State

🟪 Conventional zone 🟩 Sequential zone unwritten space 🟥 Sequential zone written space

Zone Control

Zone number: 1
Block: 524288

gzbc

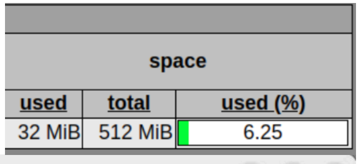
Graphical user interface (GUI)

Overwrite the file (16 MiB):

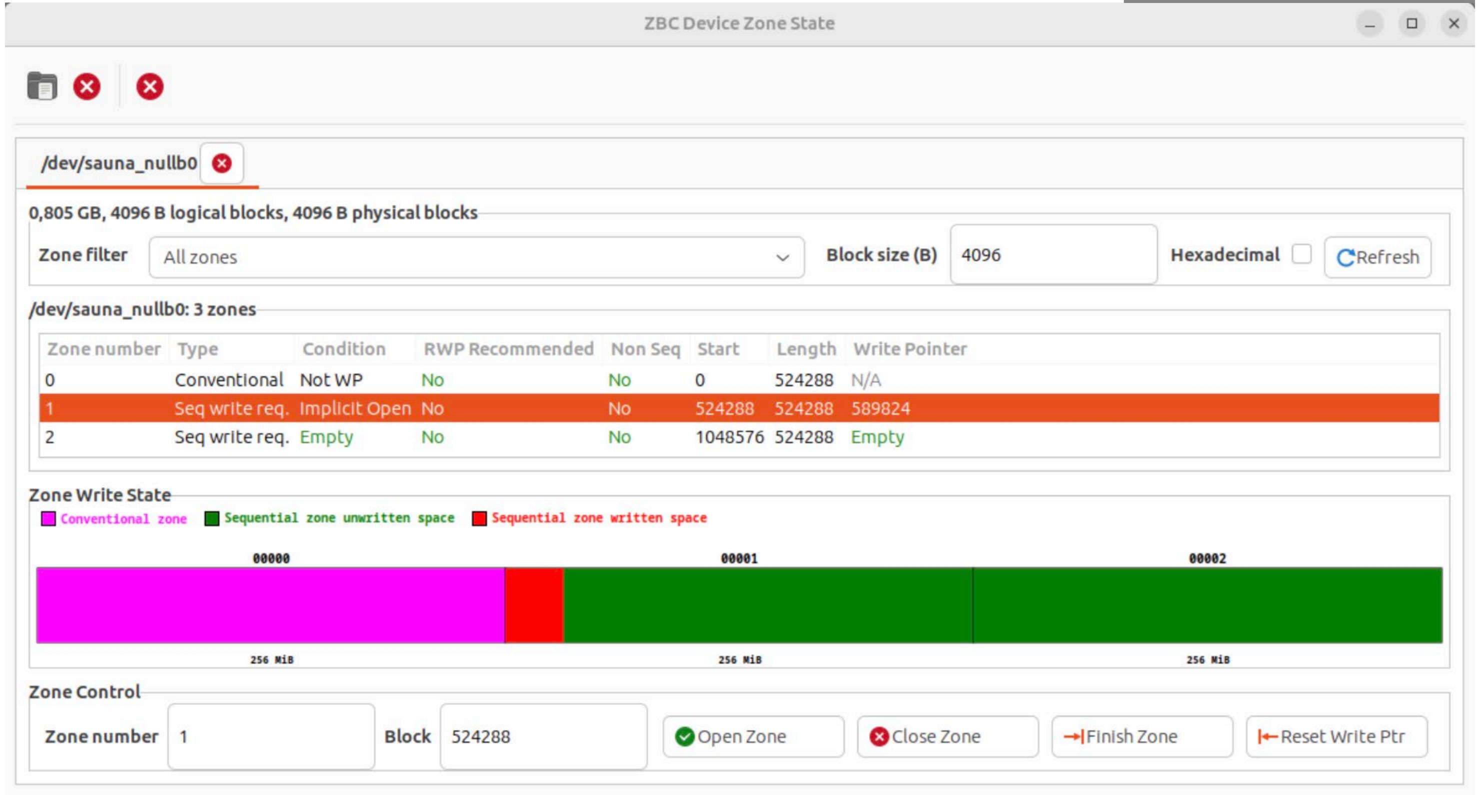
```
fio --name=file01 --
directory=/mnt/saunafs --
size=16M --rw=write --
bs=64K --bs=64K overwrite=1
```

- To deal with the sequential write constraint, the chunk is automatically fragmented.
- The same zone is preferred.
- An overhead of used space is created (16 MiB extra).

SaunaFS monitoring



space		
used	total	used (%)
32 MiB	512 MiB	6.25



ZBC Device Zone State

/dev/sauna_nullb0

0,805 GB, 4096 B logical blocks, 4096 B physical blocks

Zone filter: All zones
Block size (B): 4096
Hexadecimal □

/dev/sauna_nullb0: 3 zones

Zone number	Type	Condition	RWP Recommended	Non Seq	Start	Length	Write Pointer
0	Conventional	Not WP	No	No	0	524288	N/A
1	Seq write req.	Implicit Open	No	No	524288	524288	589824
2	Seq write req.	Empty	No	No	1048576	524288	Empty

Zone Write State

🟪 Conventional zone 🟩 Sequential zone unwritten space 🟥 Sequential zone written space

Zone Control

Zone number: 1
Block: 524288

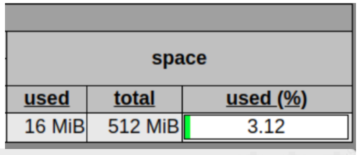
gzbc

Graphical user interface (GUI)

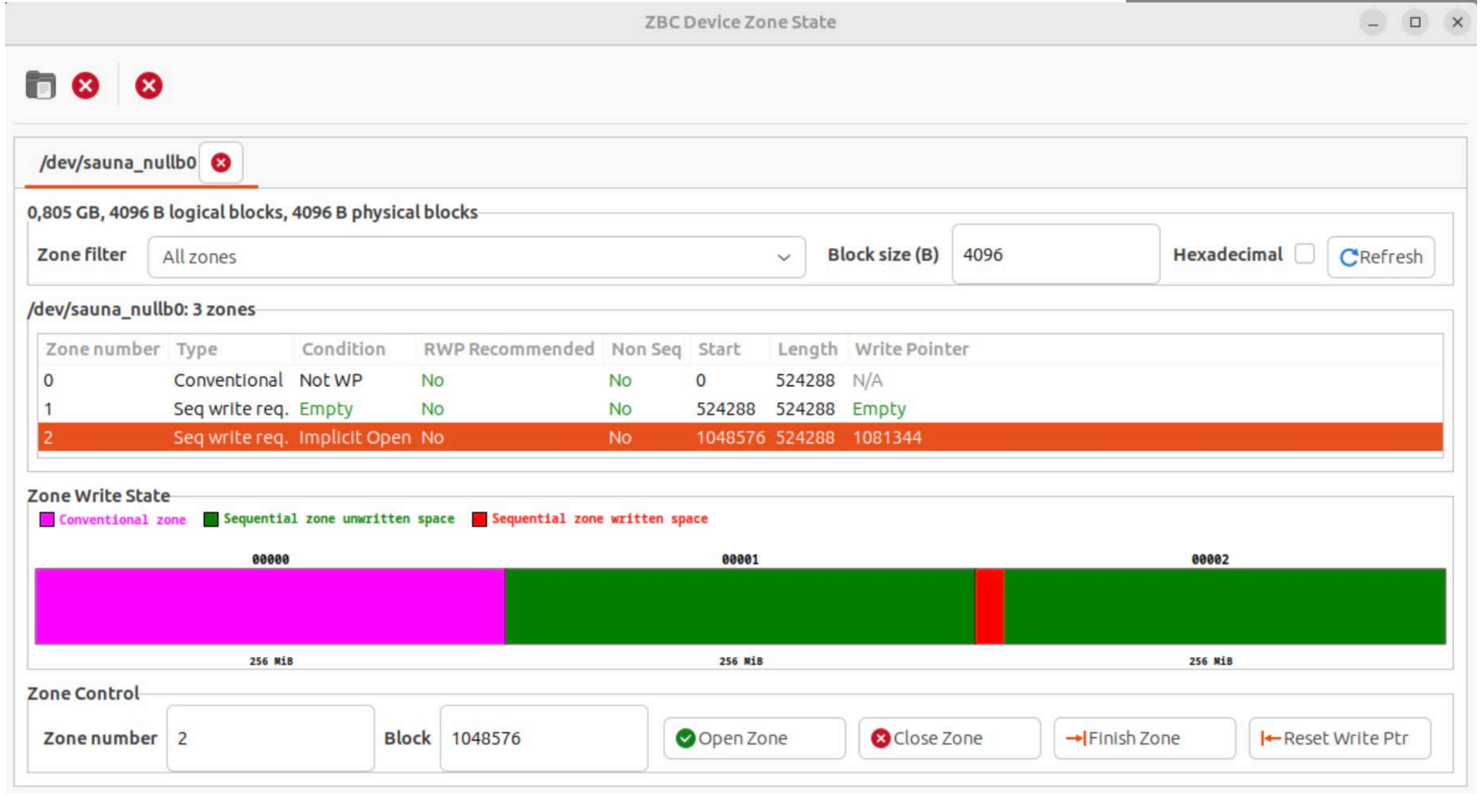
Garbage collection:

- The chunk is defragmented to another zone using the original size.
- The previous zone is marked as dirty.
- As there are no chunks now referencing the previous zone, it can be reset.

SaunaFS monitoring



space		
used	total	used (%)
16 MiB	512 MiB	3.12



ZBC Device Zone State

/dev/sauna_nullb0

0,805 GB, 4096 B logical blocks, 4096 B physical blocks

Zone filter: All zones

Block size (B): 4096

Hexadecimal □

/dev/sauna_nullb0: 3 zones

Zone number	Type	Condition	RWP Recommended	Non Seq	Start	Length	Write Pointer
0	Conventional	Not WP	No	No	0	524288	N/A
1	Seq write req.	Empty	No	No	524288	524288	Empty
2	Seq write req.	Implicit Open	No	No	1048576	524288	1081344

Zone Write State

Refresh
Conventional zone
Sequential zone unwritten space
Sequential zone written space
00000
00001
00002
256 MiB
256 MiB
256 MiB

Zone Control

Zone number: 2

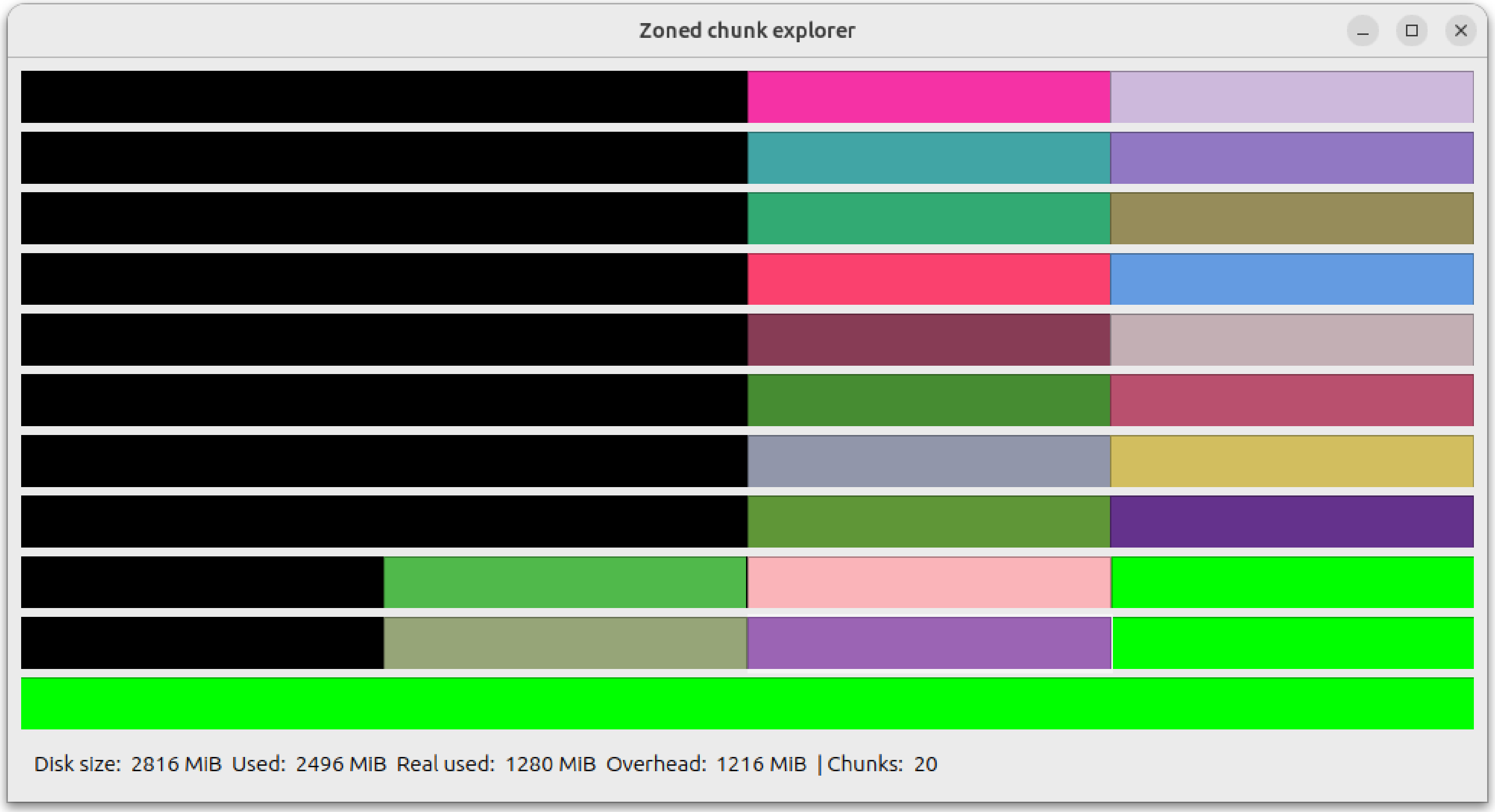
Block: 1048576

gzbc

Graphical user interface (GUI)

Garbage collection:

- Overwrite the files to generate unreferenced space in the Zones.
- SaunaFS graphical tool represents the holes in black color.
- Each different color is a Chunk.
- Free space in the Zone is green.

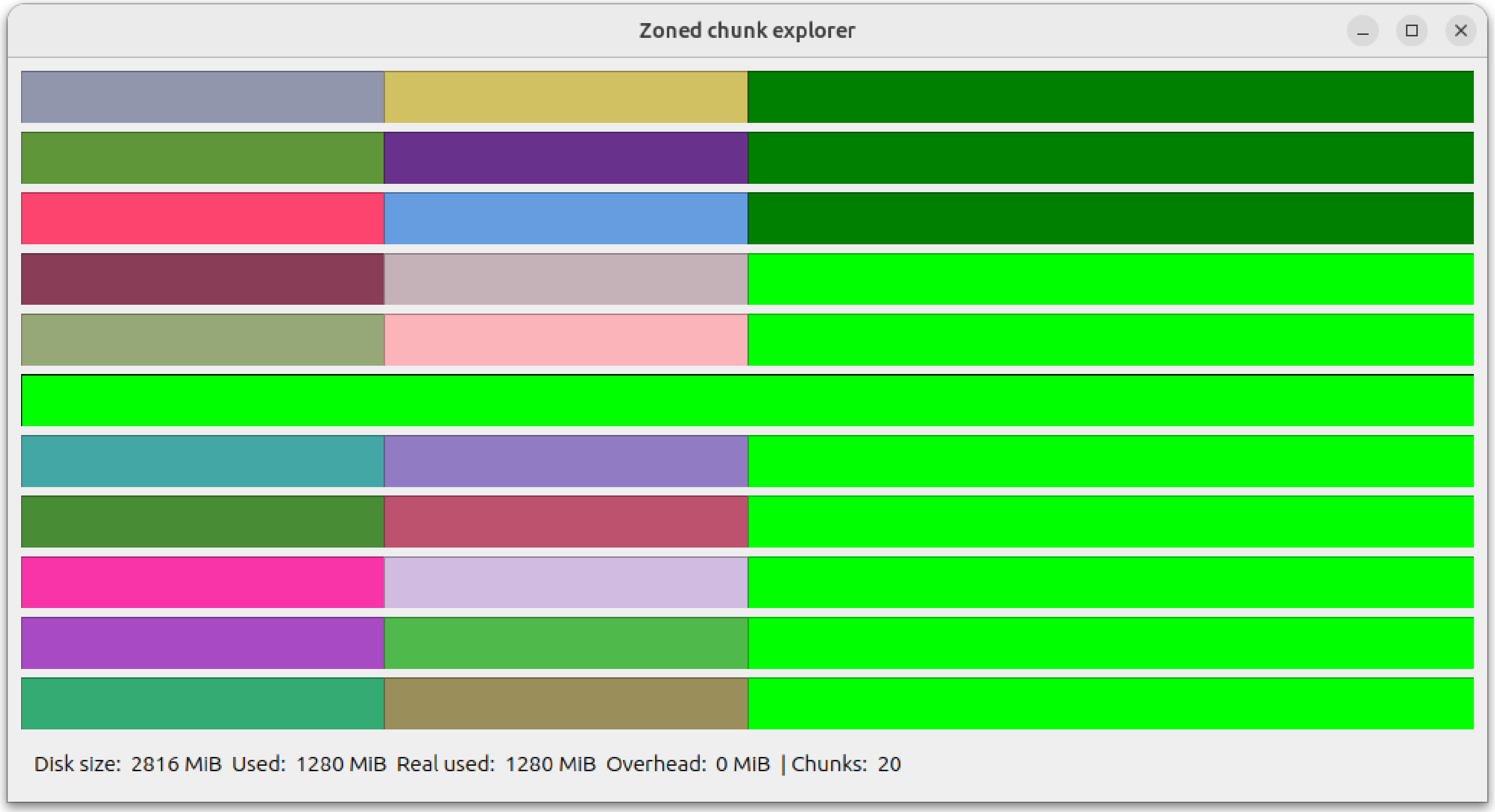


Disk size: 2816 MiB Used: 2496 MiB Real used: 1280 MiB Overhead: 1216 MiB | Chunks: 20

Graphical user interface (GUI)

Garbage collection:

- All chunks in zones are defragmented.
- All dirty (black) space is reclaimed.
- MINIMUM ONE ZONE IS EMPTY.

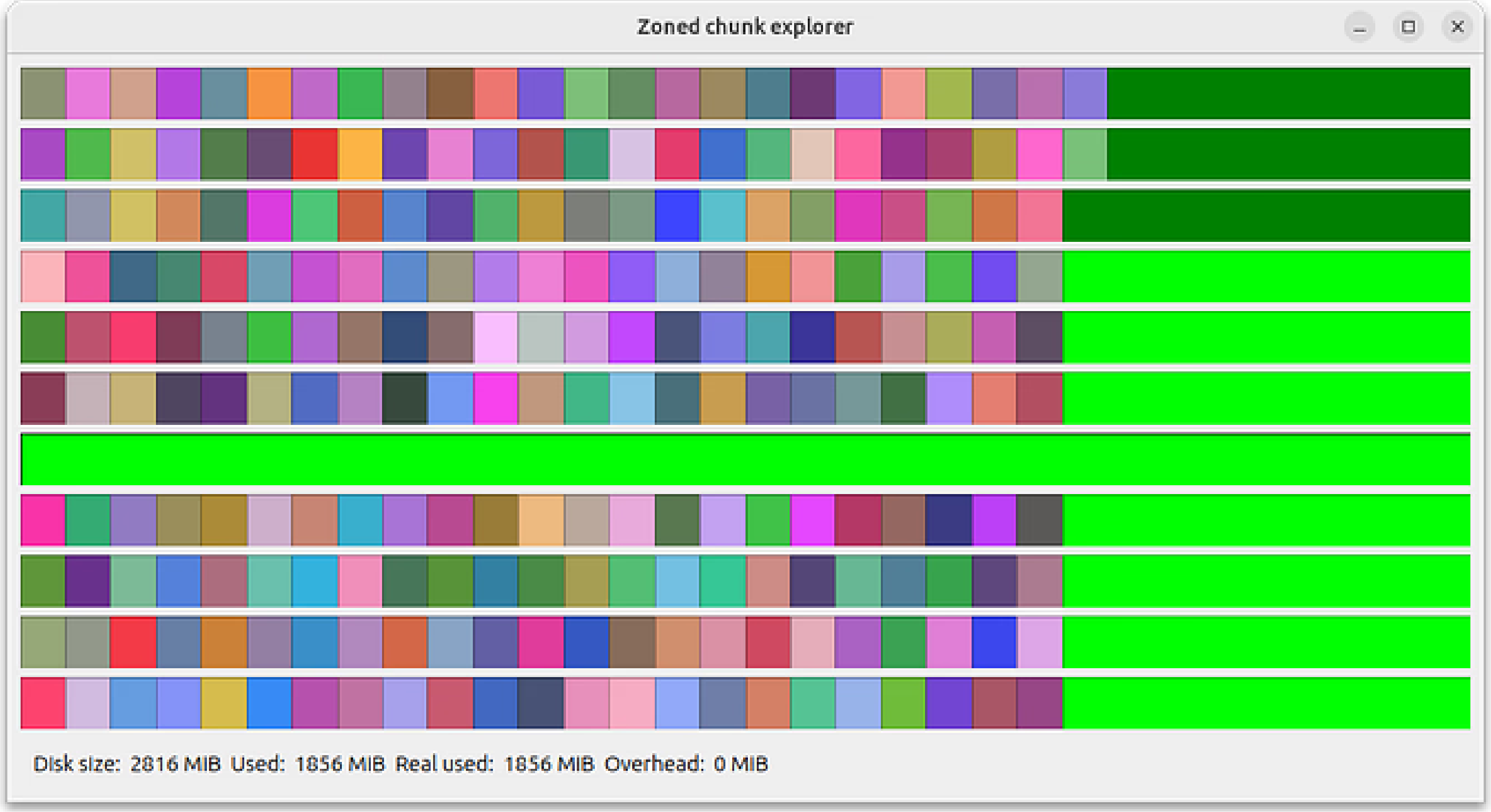


Disk size: 2816 MiB Used: 1280 MiB Real used: 1280 MiB Overhead: 0 MiB | Chunks: 20

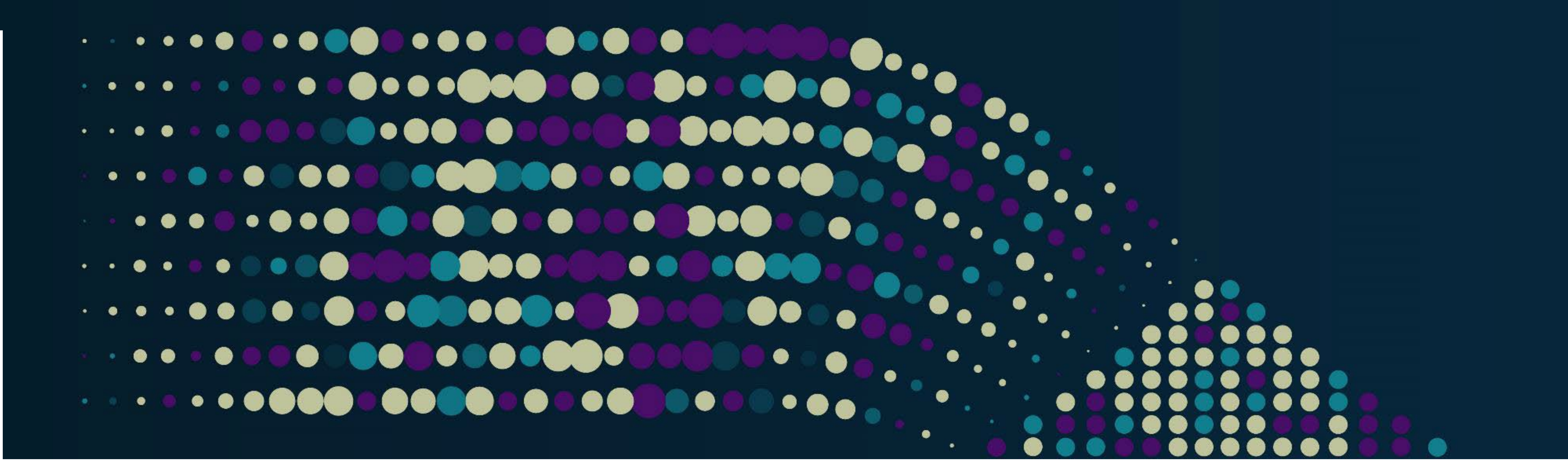
Graphical user interface (GUI)

Garbage collection – with 8MiB chunks:

- All chunks in zones are defragmented.
- All dirty (black) space is reclaimed.



Disk size: 2816 MiB Used: 1856 MiB Real used: 1856 MiB Overhead: 0 MiB



Thank you!

Your feedback is important to us.

Piotr Modrzyk

Principal Architect

pm@leil.io