

Maintaining Lustre Write Performance on NVMe RAID with TRIM

Daniel Landau – Solution Architect, Xinnor



RINNOR

Read-intensive
drives for any
task?



The write speed problem

INNOR

- We see, that NVMe market is flooded with **read-optimized** high-volume NVMe drives
- Such drives are often used even at the projects, where write speed is important

As a result:

- A RAID write speed becomes unstable
- A RAID write speed degrades because of the drive performance degradation. A drive speed becomes in 2-3 times lower, than the speed at the drive specification
- A RAID write speed is unpredictable – sizing is very difficult



How do end users evaluate NVMe write performance?

INNOR

1. NVMe Specification

(2) Sequential Performance is based on FIO on Linux, 512KB data size, with QD=32, 1 job.

(3) Random Performance is based on FIO on Linux, 4KB data size, QD=128, 8 jobs.

Performance Specifications		PM1715412, PM171540V
Sequential Bandwidth - 100% Read (up to)		14500
Sequential Bandwidth - 100% Write (up to)		9300
Random Read (100% Span)		2750000 IOPS (4K Blocks)

2. Endurance rating based on DWPD using 4KB 100% random write and JEDEC 2.19 workload over 5 years.

3. Based on internal testing. Performance will vary by capacity point, or with the changes in useable capacity. Consult product manual for further details. All performance measurements are in full sustained mode and are peak values. Subject to change.

4. Average random read latency at 4KiB, QD=1.

2. Self-testing

- Following SNIA specification, in the sustained mode (the best case)
- But how often do you see sustain mode in production?

3. Testing under real application.

- It happens very rarely.

A real life example

INNOR

NVMe specification write performance: 2400MB/s

128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
------	------	------	------	------	------	------	------	------	------

Full sequential writing with bs=128K:

```
[global]

rw=write
bs=128K
iodepth=32
direct=1
ioengine=libaio
```

```
[job1]
filename=/dev/nvme17n1
```

Result: stable 2400MB/s following the specification

128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
------	------	------	------	------	------	------	------	------	------

Write two blocks, skip two blocks bs=128K

```
[global]
rw=write:512K
rw_sequencer=identical
bs=128K
iodepth=32
direct=1
ioengine=libaio
```

```
[job1]
filename=/dev/nvme18n1
io_size=360GiB
[job2]
filename=/dev/nvme18n1
io_size=360GiB
offset=128K
```

Result: performance degraded to 760MB/s after half of the drive rewriting

The write speed solution

INNOR

- Each NVMe drive has a mechanism may helping sustain or improve the write speed: TRIM/deallocate/discard.
- This mechanism helps OS to inform an NVMe which blocks of data are no longer considered to be "in use" and therefore can be erased internally.
- Informing an NVMe about such blocks does not mean, that they are become erased immediately by NVMe. It's done by NVMe controller in the background.
- Anyway, availability of such erased blocks helps NVMe to operate faster and maintain high writing speed.

Solution:

Support TRIM at the RAID level and convert TRIM operations to RAID to TRIM operations to underlying drives.

NUSE control

nvme list

Node	SN	Model	Namespace	Usage	Format	FW Rev
/dev/nvme0n1	91L0033P	PASCARI XX20DH033T84P314T0910	1	322.65 GB / 773.09 GB	4 KiB + 0 B	X2WM40N1
/dev/nvme0n2	91L0033P	PASCARI XX20DH033T84P314T0910	2	322.40 GB / 773.09 GB	4 KiB + 0 B	X2WM40N1
/dev/nvme0n3	91L0033P	PASCARI XX20DH033T84P314T0910	3	322.35 GB / 773.09 GB	4 KiB + 0 B	X2WM40N1
...						
/dev/nvme10n1	91L0033T	PASCARI XX20DH033T84P314T0910	1	586.37 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme10n2	91L0033T	PASCARI XX20DH033T84P314T0910	2	586.37 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme11n1	91L00305	PASCARI XX20DH033T84P314T0910	1	586.37 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme11n2	91L00305	PASCARI XX20DH033T84P314T0910	2	586.38 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme12n1	91L0035Q	PASCARI XX20DH033T84P314T0910	1	586.38 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme12n2	91L0035Q	PASCARI XX20DH033T84P314T0910	2	586.38 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
/dev/nvme13n1	91L0034A	PASCARI XX20DH033T84P314T0910	1	586.37 GB / 1.50 TB	4 KiB + 0 B	X2WM40N1
...						

NUSE stands for Namespace Utilization: it indicates the data volume actually allocated and in use within a given namespace on the drive.

RINNOR

RAIDs & TRIM

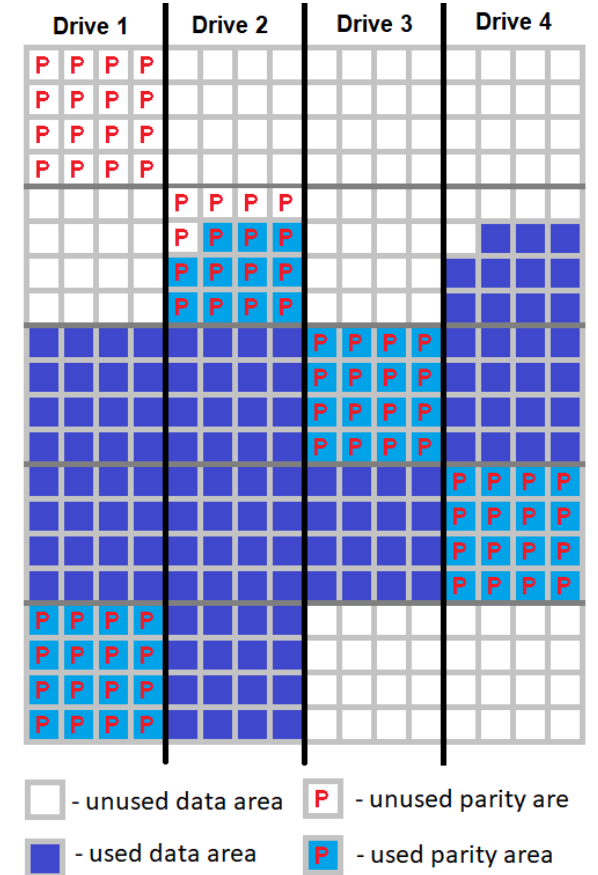
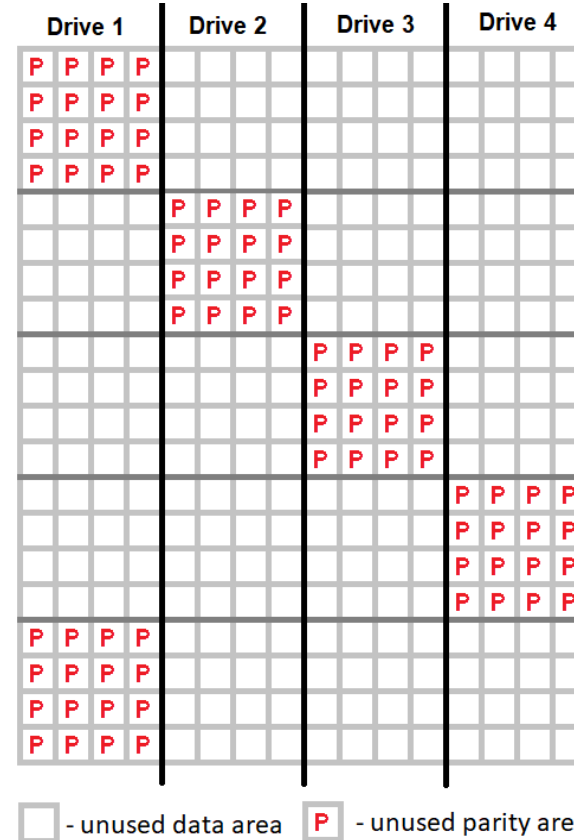


TRIM in non-parity RAIDs

- RAID0, RAID1, RAID10
- Easy to implement
- Deterministic read from a TRIM region is not so important
- Effective, since write speed on such RAIDs is easily scalable and high

TRIM in parity RAIDs

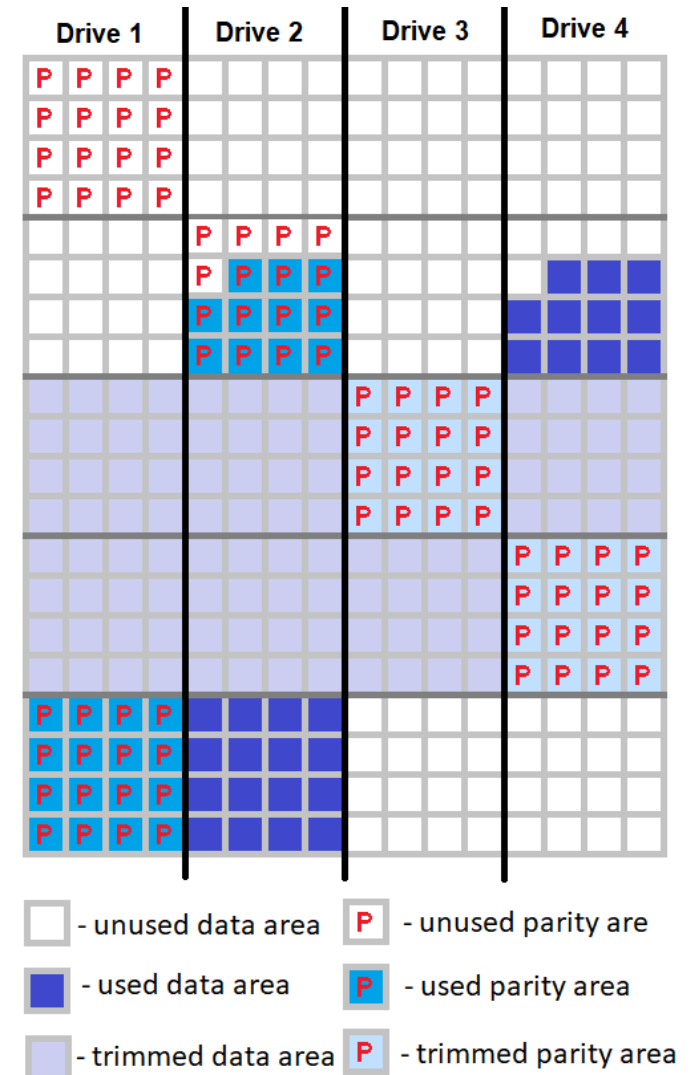
- RAID5, 6, 7, 50, 60, 70
- Hard to implement, since internal stripe consistency required
- Deterministic read from a TRIM region is very important for the realization
- Can be not so effective in alternative realizations, since write speed in parity RAIDs is limited by the syndrome calculations anyway



RAID TRIM. Stripe consistency maintaining

- All data strips in any stripe have to be consistent with the parity stored at the parity strip of the same stripe.
- If this consistency is broken, then the RAID data will be corrupted in case of a drive loss.
- If all data strips of a stripe are 0, then consistent parity strip value is also 0.
- RZAT drives – drives which **R**eturn **Z**ero **A**fter **T**rim.

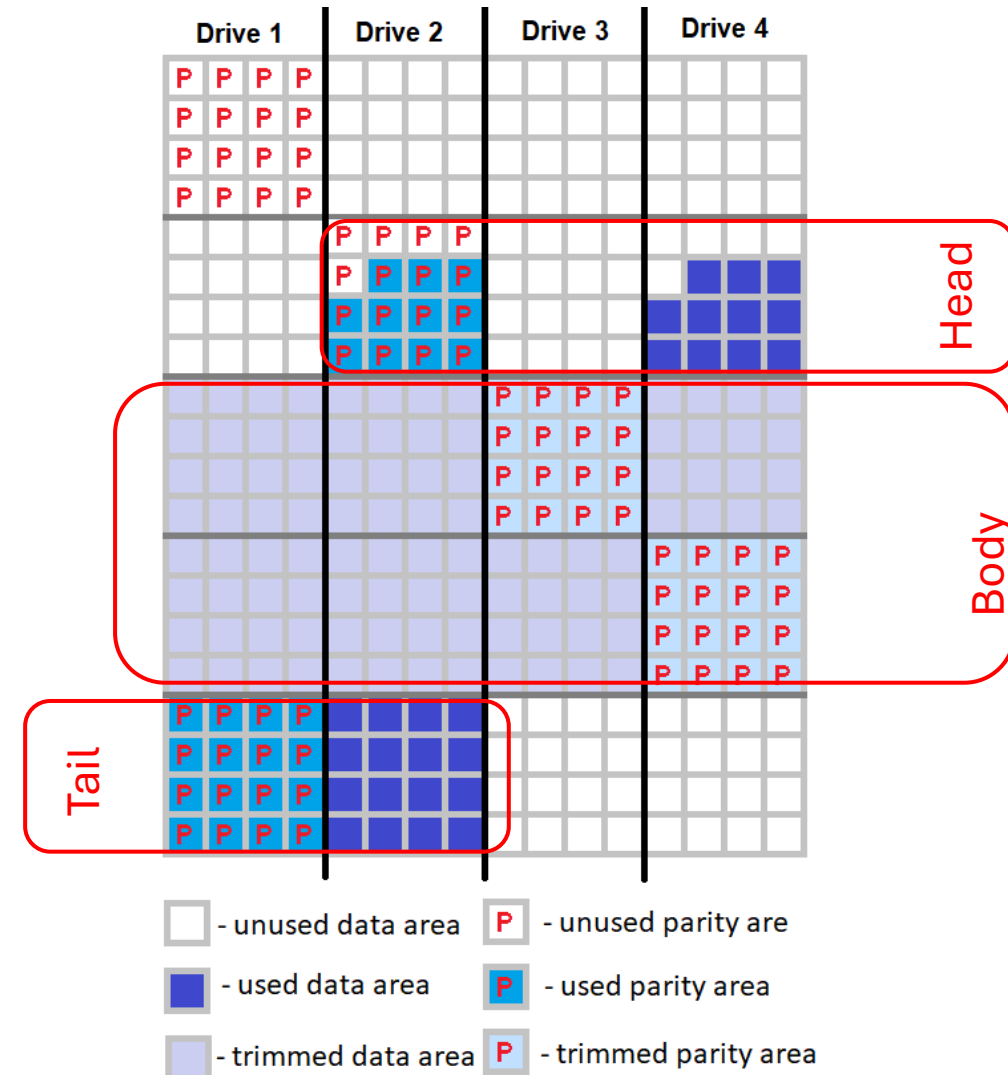
So if RAID consists of RZAT drives only and all strips of a stripes were trimmed, then this stripe will still be consistent, since all data and parity strips value will be 0.



RAID TRIM. Stripe consistency maintaining

In xiRAID Classic 4.4 a single TRIM region is split in 3 parts:

- Head - first stripe which should be TRIMed partially. It is discarded in RMW way
- Body – set of full stripes to be TRIMed. It is TRIMed at the drive level by large regions, more than a single chunk. It's done to improve the speed.
- Tail - last stripe which should be TRIMed partially. It is discarded in RMW way.



RINNOR

The experiment idea and testbed

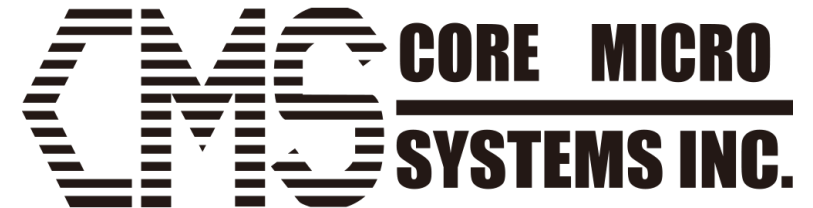


The idea

- Identify the patterns which can be affected by NVMe performance degradation
- Check how using discard could help improving the performance
- Prove that xiRAID Classic is fast enough to get the full performance from the TRIMed RAID drives

To generate the standard workload we decided to use the IO500 test (what can be more standard?)

The testbed for the test was provided by our partner – CoreMicro Systems (Japan)



The testbed configuration

1 x server platform - Viking VS2320

Includes:

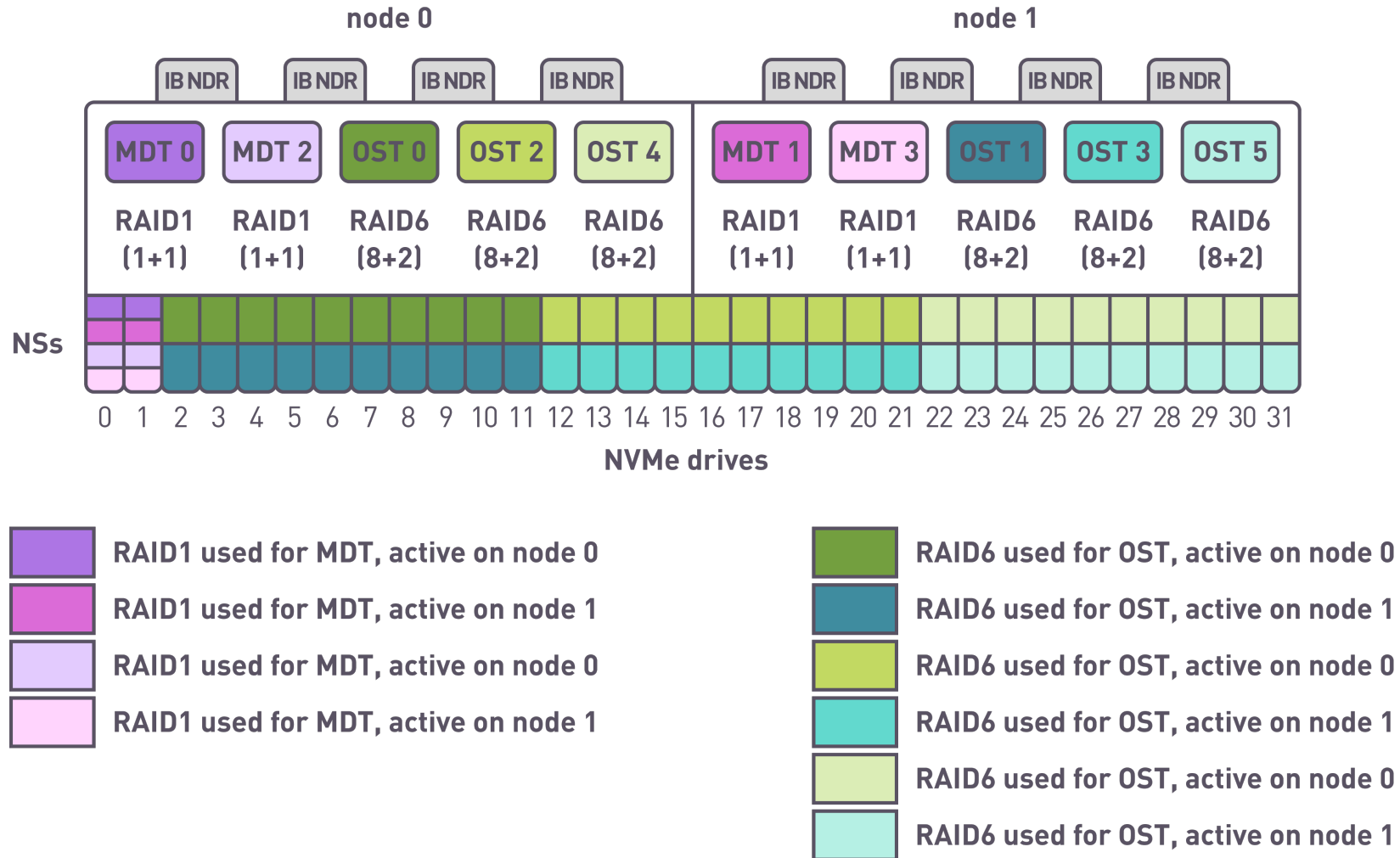
- 2 servers
 - 1 x AMD EPYC 9534 64-Core per server
 - 810GB RAM per server
 - 4 x ConnectX7 IB NDR (400Gb/s)
- 32 shared dual ported drives Phison PASCARI X200P E3.S
 - Size: 3.84TB
 - Seq read by spec: 14800GB/s
 - Seq write by spec: 8600GB/s
 - Endurance: 1DWPD
 - Technology: TLC
- Lustre version: 2.15.8 (current LTS)

8 x client systems

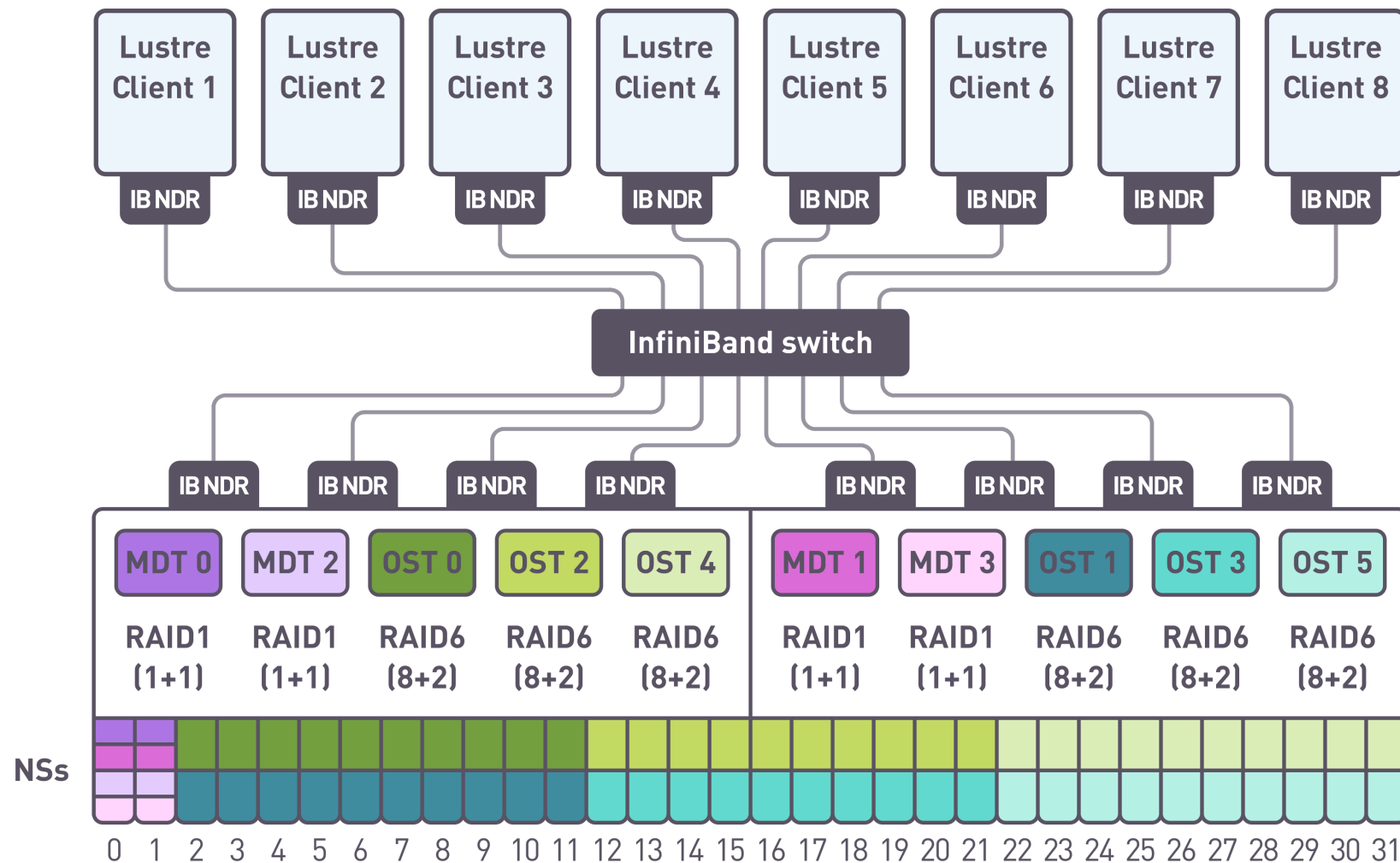
Each includes:

- 2 x AMD EPYC 9654 96-Core
- 810GB RAM per server
- 1 x ConnectX7 IB NDR (400Gb/s)
- Lustre version: 2.15.8 (current LTS)

The testbed Lustre system configuration



The testbed scheme



The Lustre filesystem

```
# lfs df -h
```

UUID	bytes	Used	Available	Use%	Mounted on
lustrel-MDT0000_UUID	424.4G	8.6G	379.8G	3%	/mnt.l1[MDT:0]
lustrel-MDT0001_UUID	424.4G	8.4G	380.0G	3%	/mnt.l1[MDT:1]
lustrel-MDT0002_UUID	424.4G	8.3G	380.0G	3%	/mnt.l1[MDT:2]
lustrel-MDT0003_UUID	424.4G	8.4G	380.0G	3%	/mnt.l1[MDT:3]
lustrel-OST0000_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:0]
lustrel-OST0001_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:1]
lustrel-OST0002_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:2]
lustrel-OST0003_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:3]
lustrel-OST0004_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:4]
lustrel-OST0005_UUID	10.8T	1.7G	10.3T	1%	/mnt.l1[OST:5]

```
filesystem_summary: 65.0T 10.3G 61.7T 1% /mnt.l1
```

The test plan

- We used the full IO500 test
- The time of a single test was below the IO500 required for the official submission (180 seconds instead of 300) by the following reasons:
 - The filesystem size was relatively small to the comparison with the throughput, so the test shows some errors, it's OK
 - We needed to run the test multiple times to check the dynamics, do not need the absolute values so much
- The test sequence consists of 15 test runs without discard and 15 tests with discards with fstrim between them
- Before the sequence start the file system was preconditioned

The filesystem preconditions before a test sequence

Can be done without filesystem going down

1. Rewrite the filesystem two times with large files:

```
[root@client1]# lfs getstripe /mnt.ll/stripe2Mc1/  
/mnt.ll/stripe2Mc1/  
stripe_count: 1 stripe_size: 2097152 pattern: raid0 stripe_offset: -1  
  
[root@client1]# mpirun --allow-run-as-root -np 288 --map-by node --hostfile /root/daniel/ior/hosts_8n  
/mnt.ll/io500/bin/ior -F -t 16M -b 250G -k -w -r -o /mnt.ll/stripe2Mc1/testfile --posix.odirect  
  
[root@client1]# rm -rf /mnt.ll/stripe2Mc1/*
```

2. Run fstrim on the MDS+OSS servers:

```
[root@vikings-ost1 ~]# for i in `mount -t lustre|awk '{ print $3 }'`; do time fstrim -v $i; done  
[root@vikings-ost2 ~]# for i in `mount -t lustre|awk '{ print $3 }'`; do time fstrim -v $i; done
```

3. Wait for an hour to let GC work

An IO500 workload brief

Workload	Pattern / granularity	I/O mode	Primary target
ior-easy R/W	Sequential, file-per-process - 16 MiB	Direct (O_DIRECT)	OST SSDs*
ior-hard R/W	Shared-file, unaligned I/O - 47,008 B (~46 KiB)	Buffered POSIX	OST SSDs*
ior-rnd4K-easy-read	Random reads - 4 KiB	Direct (O_DIRECT)	OST SSDs
mdtest-easy write	File creation, no payload	Metadata only	MDT SSDs
mdtest-easy stat/delete	stat / unlink	Metadata only	MDT SSDs
mdtest-hard write/read	Tiny-file data I/O - 3,901 B	Buffered POSIX	MDT SSDs via DoM**
mdtest-hard stat/delete	Metadata operations on tiny files	Metadata only	MDT SSDs
find	Namespace traversal / lookup	Metadata only	MDT SSDs

* - IOR tests also generate metadata operations on the MDT, but their **dominant data-path load is on the OST NVMe RAIDs**.

** - **DoM = 256 KiB**: the 3,901-byte mdtest-hard files reside entirely on the MDT, including their file data.

A single test run output example

```

IO500 version io500-isc26_v2-26-g02091cf254d8 (standard)
ERROR INVALID (src/phase_dbg.c:15) stonewall-time 180s != 300s
ERROR INVALID (src/main.c:490) Runtime is smaller than expected minimum runtime
[RESULT]          ior-easy-write          121.082861 GiB/s : time 211.767 seconds [INVALID]
ERROR INVALID (src/main.c:490) Runtime is smaller than expected minimum runtime
[RESULT]          mdtest-easy-write        229.918607 kIOPS : time 187.100 seconds [INVALID]
[          ]          timestamp            0.000000 kIOPS : time 0.000 seconds
ERROR INVALID (src/main.c:490) Runtime is smaller than expected minimum runtime
[RESULT]          ior-hard-write           2.849528 GiB/s : time 177.094 seconds [INVALID]
ERROR INVALID (src/main.c:490) Runtime is smaller than expected minimum runtime
[RESULT]          mdtest-hard-write        38.402748 kIOPS : time 184.115 seconds [INVALID]
[RESULT]          find                    1539.365042 kIOPS : time 32.361 seconds
[RESULT]          ior-easy-read            227.601040 GiB/s : time 112.670 seconds
[RESULT]          mdtest-easy-stat          462.700006 kIOPS : time 93.472 seconds
[RESULT]          ior-hard-read            15.273336 GiB/s : time 33.080 seconds
[RESULT]          mdtest-hard-stat          463.028553 kIOPS : time 35.681 seconds
[RESULT]          mdtest-easy-delete        244.514896 kIOPS : time 176.241 seconds
[RESULT]          mdtest-hard-read          389.003623 kIOPS : time 19.105 seconds
[RESULT]          mdtest-hard-delete        30.266287 kIOPS : time 233.295 seconds
[          ]          ior-rnd4K-easy-read   3.483736 GiB/s : time 196.127 seconds
[SCORE ] Bandwidth 33.093366 GiB/s : IOPS 231.966586 kiops : TOTAL 87.615952 [INVALID]

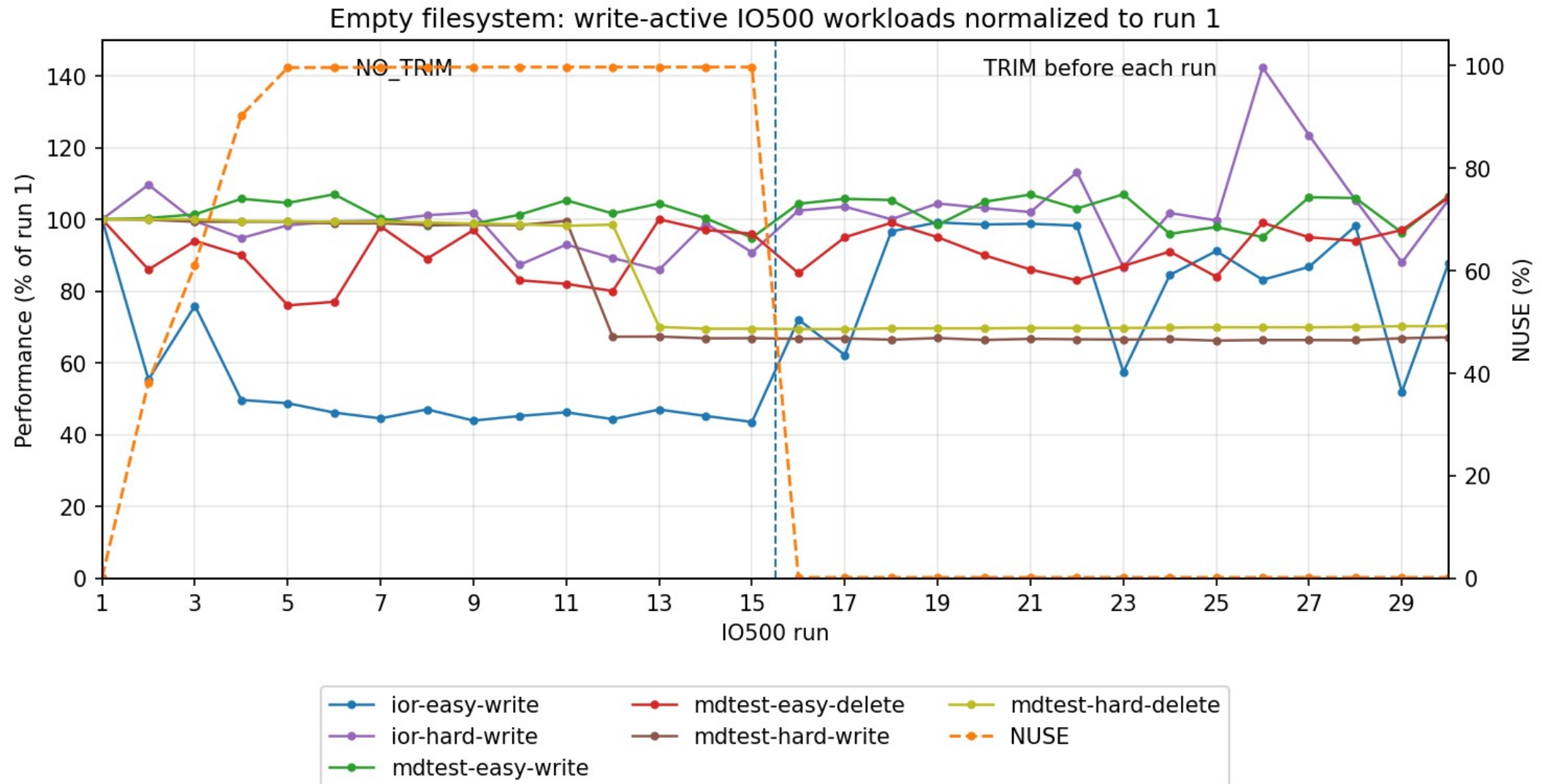
```

RINNOR

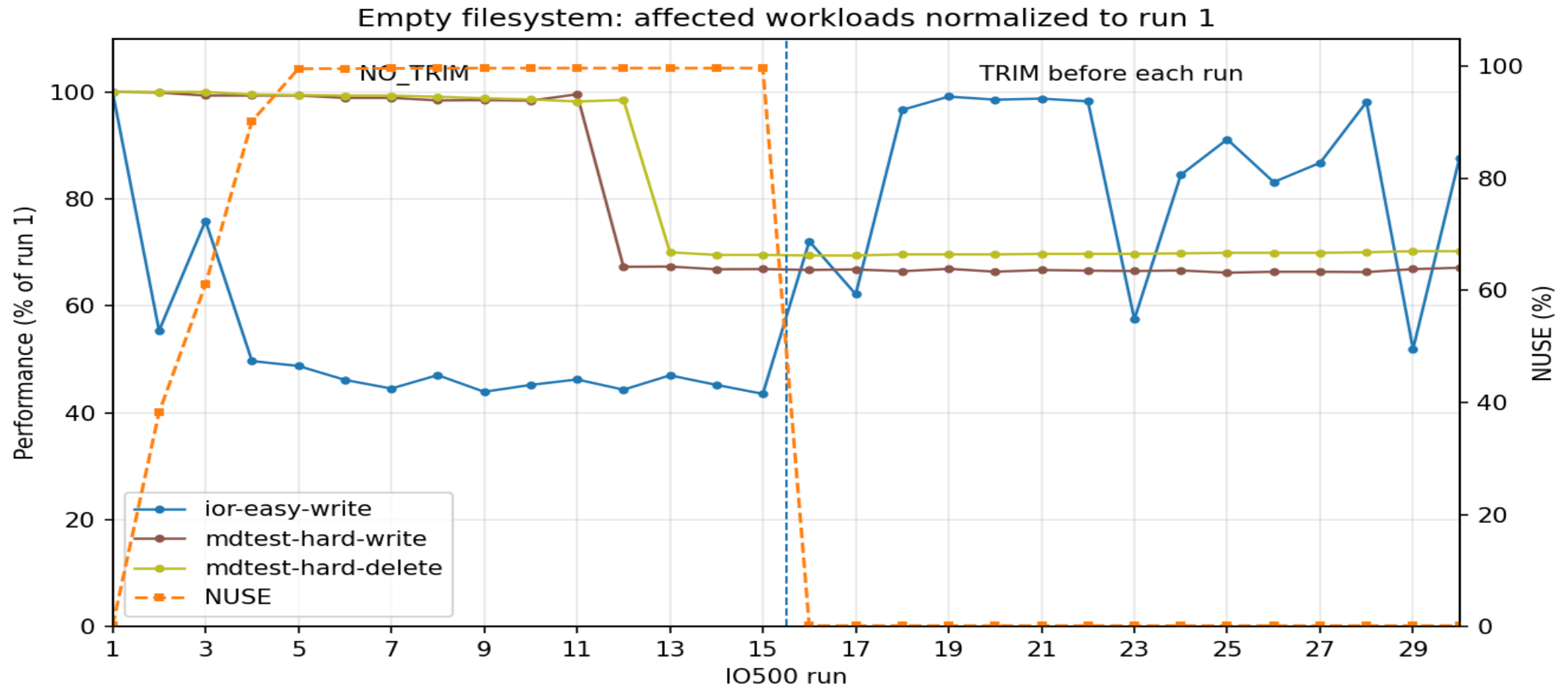
Results and analysys



The results



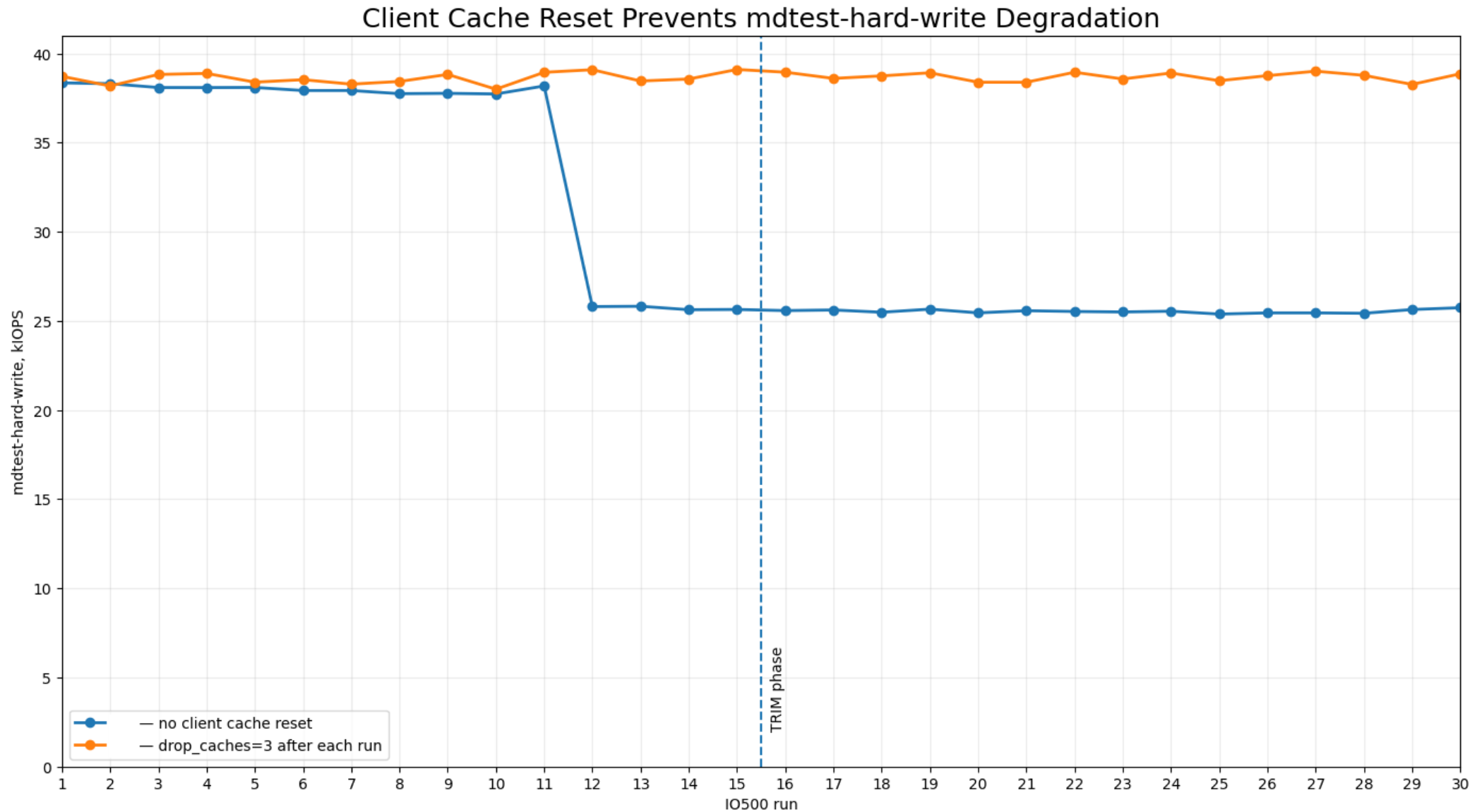
The results - affected tests only



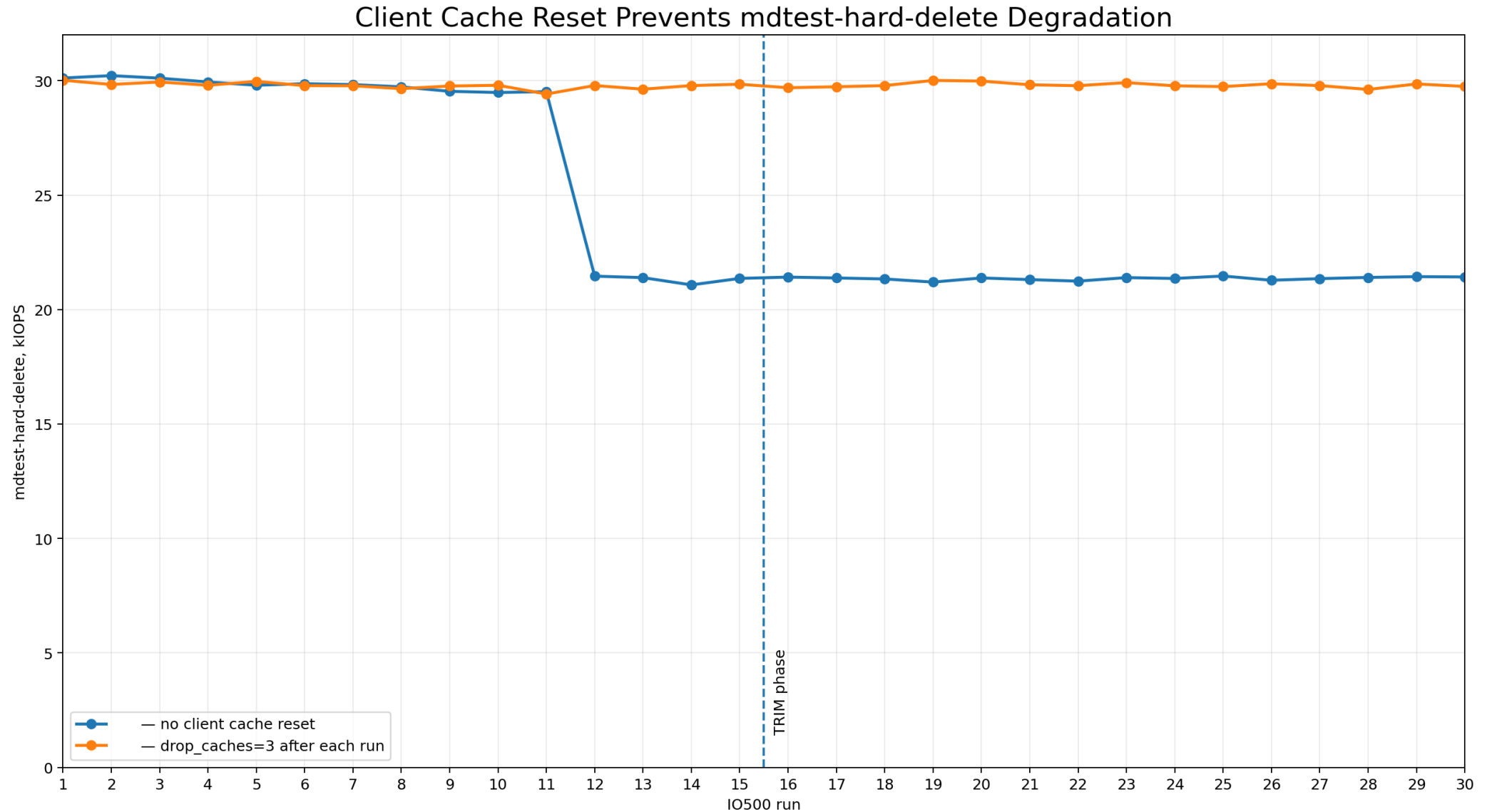
The mdtest-* results explanation

- We had a theory, that the performance drop was caused by the drives write performance degradation.
- Later we found, that the repeated metadata workloads cause the client-side inode/dentry cache to grow to hundreds of GiB, increasing metadata-operation overhead and degrading mdtest-hard create/delete performance until the cache is reclaimed.
- Tried to use recommended to set `llite.*.inode_cache=0` but this parameter does not exist at Lustre 2.15.8 (added in 2.16 with LU-13970)
- So, we just started executing “`echo 3 > /proc/sys/vm/drop_caches`” between the io500.sh runs

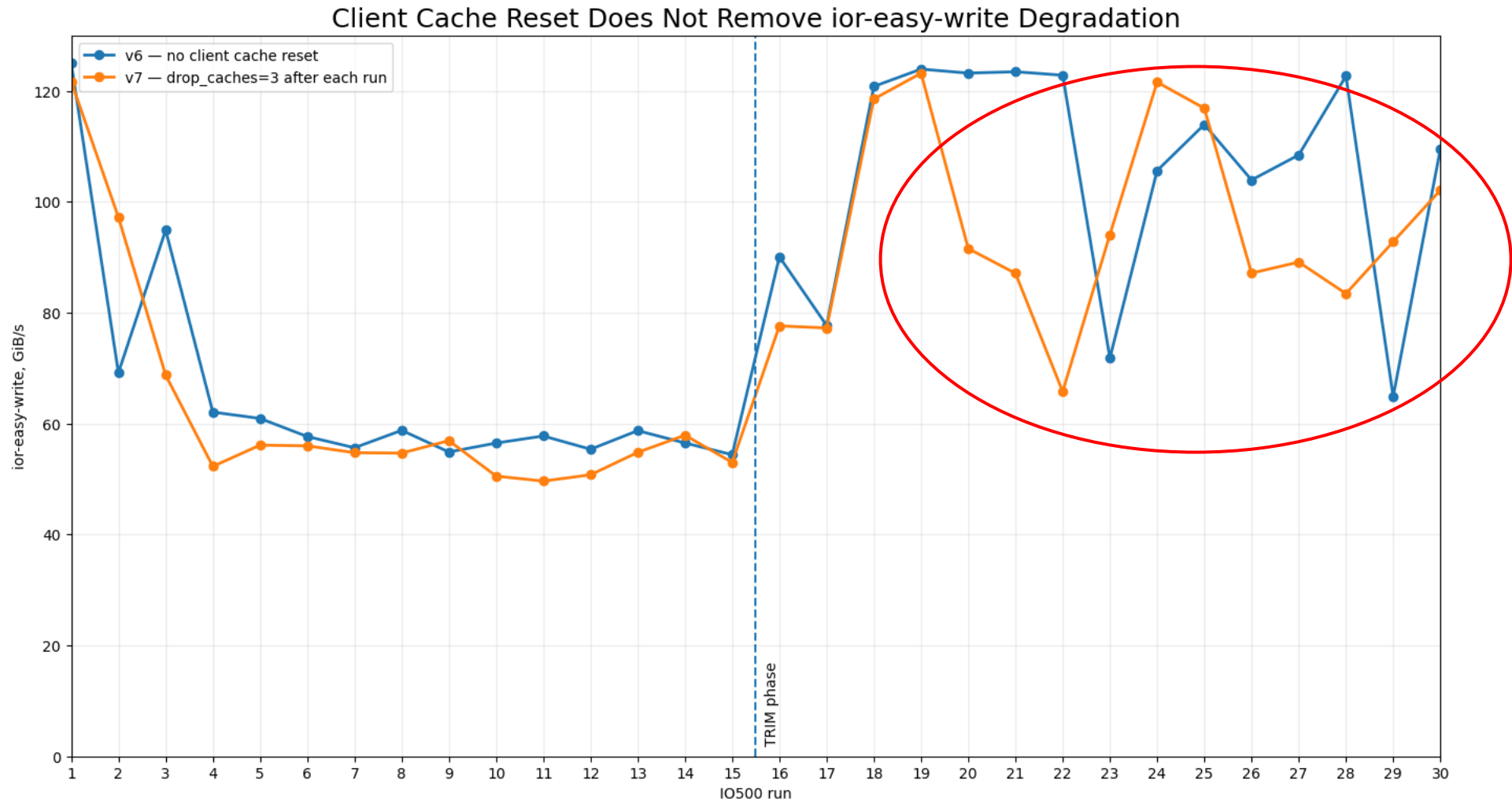
The mdtest-hard-write inode cache problem verification



The mdtest-hard-delete inode cache problem verification



The ior-easy-write drops investigation

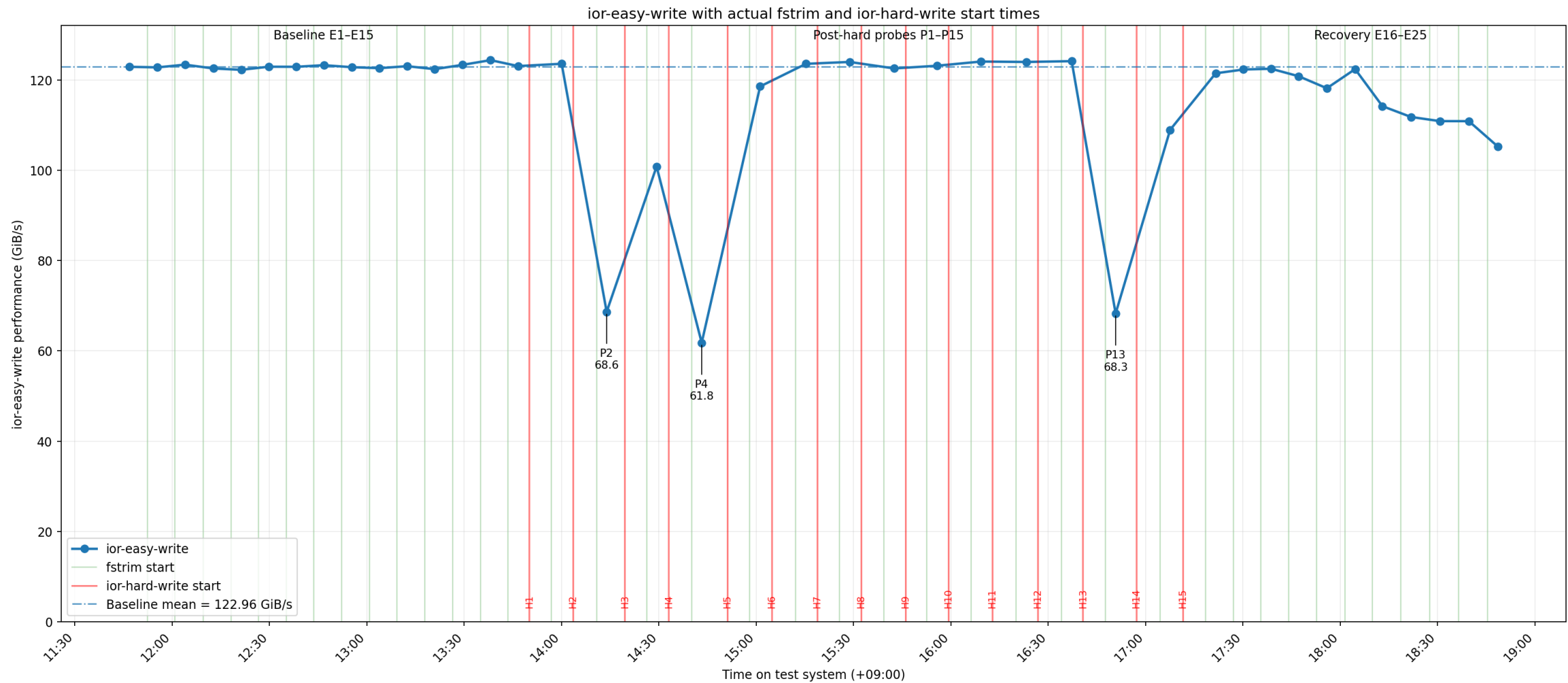


The drops are not caused by the discard efficiency

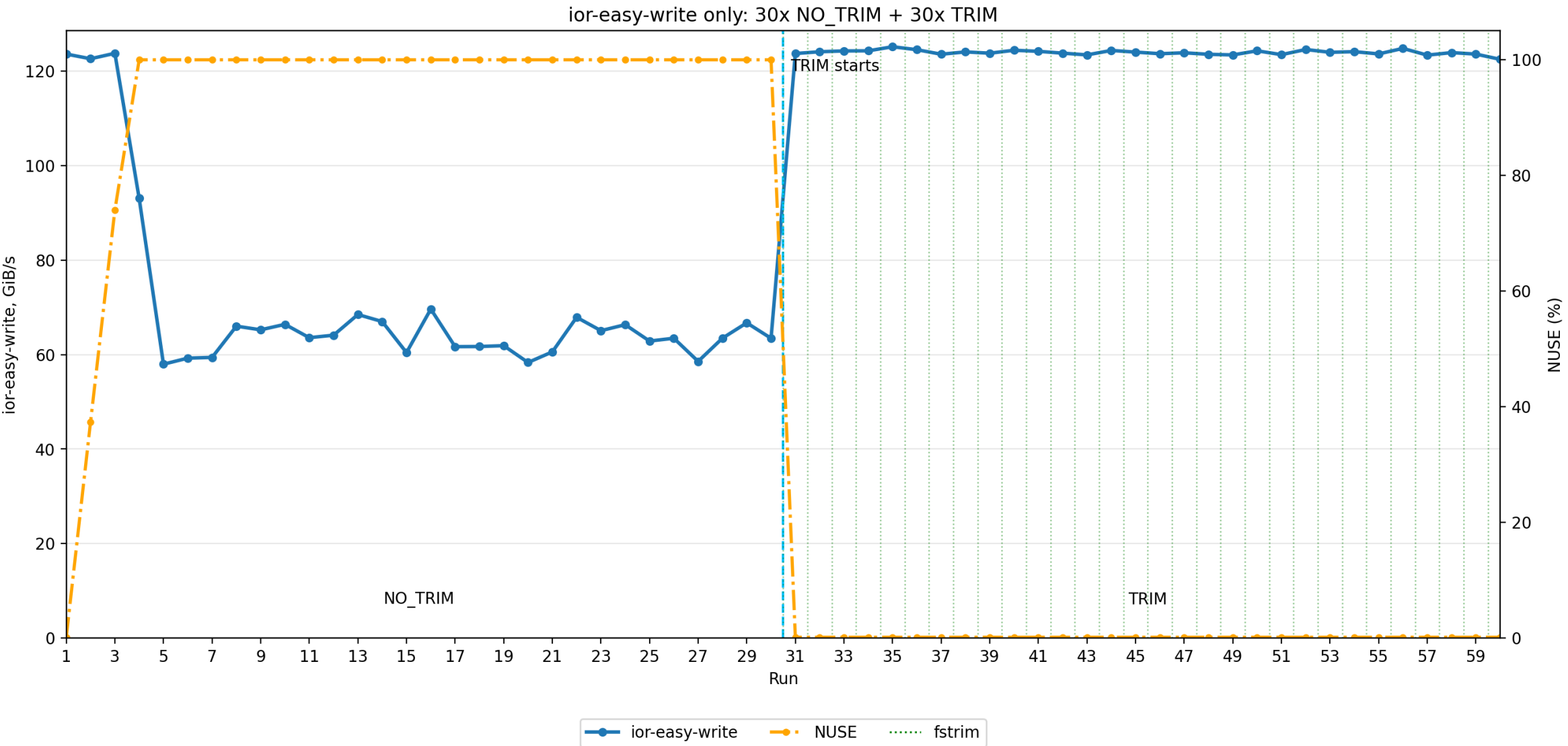
Who also writes to OSTs?

Workload	Pattern / granularity	I/O mode	Primary target
ior-easy R/W	Sequential, file-per-process - 16 MiB	Direct (O_DIRECT)	OST SSDs*
ior-hard R/W	Shared-file, unaligned I/O - 47,008 B (~46 KiB)	Buffered POSIX	OST SSDs*
ior-rnd4K-easy-read	Random reads - 4 KiB	Direct (O_DIRECT)	OST SSDs
mdtest-easy write	File creation, no payload	Metadata only	MDT SSDs
mdtest-easy stat/delete	stat / unlink	Metadata only	MDT SSDs
mdtest-hard write/read	Tiny-file data I/O - 3,901 B	Buffered POSIX	MDT SSDs via DoM**
mdtest-hard stat/delete	Metadata operations on tiny files	Metadata only	MDT SSDs
find	Namespace traversal / lookup	Metadata only	MDT SSDs

ior-hard-write affection check



Reference: ior-easy-write with and without fstrim



Conclusions

- TRIM/deallocation can significantly improve `ior-easy-write`-like pattern performance.
- The TRIM/deallocation improvement can be degraded if `ior-easy-write`-like pattern is alternated with small-block random write pattern on the same OSTs (like `ior-hard-write`).
- A drop in NUSE after `fstrim` does not necessarily mean that full write performance has been restored since logical deallocation is immediate, while physical NAND reclamation is asynchronous.
- More generic: workload history therefore affects both performance and recovery time after TRIM.

Practical recommendations

- Use `fstrim`, instead of `LDISKFS discard` mount option.
- Run `fstrim` regularly, preferably during maintenance windows or periods of low filesystem activity.
- Whenever possible, allow some idle time after large deallocation so that SSD background garbage collection can progress.
- Using `xiRAID Classic` with `TRIM` helps getting maximal write performance 😊

XINNOR

Prove it yourself:

xinnor.io

