DAOS Beyond Persistent Memory: Architecture and Initial Performance Results

Michael Hennecke[1](✉), Jeff Olivier[2], Tom Nabarro[3], Liang Zhen[4], Yawei Niu[4], Shilong Wang[4], and Xuezhao Liu[4]

[1] Intel Deutschland GmbH, Am Campeon 10, 85579 Neubiberg, Germany
michael.hennecke@intel.com
[2] Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95054-1549, USA
jeffrey.v.olivier@intel.com
[3] Intel Corporation (UK) Ltd., Pipers Way, Swindon SN3 1RJ, UK
tom.nabarro@intel.com
[4] Intel China Ltd., GTC, No. 36 3Rd Ring Road, Beijing, China
{liang.zhen,yawei.niu,shilong.wang,xuezhao.liu}@intel.com

Abstract. The Distributed Asynchronous Object Storage (DAOS) is an open source scale-out storage system that is designed from the ground up to support Storage Class Memory (SCM) and NVMe storage in user space. Until now, the DAOS storage stack has been based on Intel Optane Persistent Memory (PMem) and the Persistent Memory Development Kit (PMDK). With the discontinuation of Optane PMem, and no persistent CXL.mem devices in the market yet, DAOS continues to support PMem-based servers but now also supports server configurations where its Versioning Object Store (VOS) is held in DRAM. In this case, the VOS data structures are persisted through a synchronous Write-Ahead-Log (WAL) combined with asynchronous checkpointing to NVMe SSDs. This paper describes the new non-PMem DAOS architecture, and reports first performance results based on a DAOS 2.4 technology preview.

Keyword: DAOS · Storage Class Memory · Optane · CXL.mem · Versioning Object Store · Parallel Filesystem · mdtest

1 Introduction

The Distributed Asynchronous Object Storage (DAOS) [1–3] is an open source software-defined object store designed from the ground up for massively distributed Non-Volatile Memory (NVM) and Storage Class Memory (SCM). It presents a key-value storage interface and provides features such as transactional non-blocking I/O, a versioned data model, and global snapshots – all completely in user space. Its design eliminates many of the bottlenecks of traditional parallel filesystems, as demonstrated by leadership performance rankings in HPC storage benchmarks like the IO500 [4].

As shown in Fig. 1, the DAOS storage engine uses two types of backend storage devices. Storage Class Memory is used to store DAOS internal metadata, application/middleware key/index data, and latency sensitive small I/O. Up until today, DAOS

SCM has been implemented with Intel Optane Persistent Memory (PMem) devices [9] accessed through the Persistent Memory Development Toolkit (PMDK) [5–7]. DAOS creates a *DAX-enabled* ext4 filesystem on the PMem devices (configured in AppDirect mode). Space for the DAOS Versioning Object Store (VOS) is allocated as files in this DAX-enabled filesystem (one file per DAOS pool and per DAOS storage *target*). DAOS memory-maps these files when the engines are up and running, bypassing the Linux page cache. It can then directly access persistent memory in user space by memory instructions like load and store, instead of going through a thick storage stack.

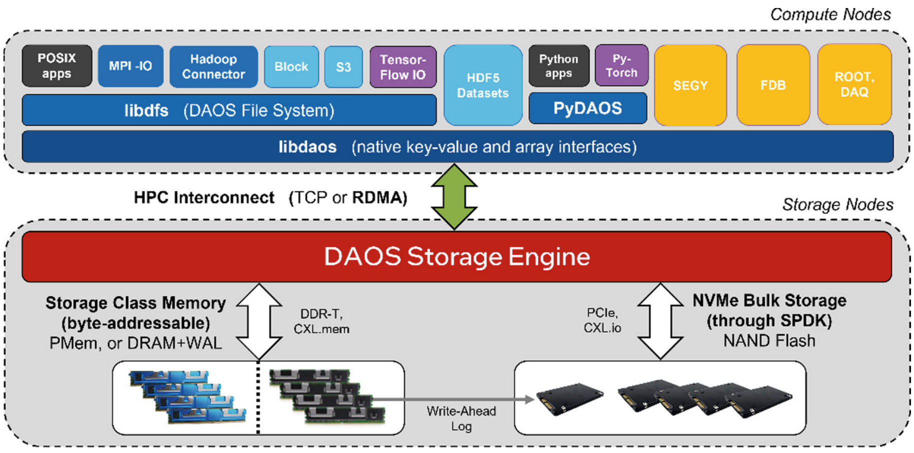

Compute Nodes
POSIX apps
MPI -IO
Hadoop Connector
Block
S3
Tensor-Flow IO
HDF5 Datasets
Python apps
Py-Torch
SEGY
FDB
ROOT, DAQ
libdfs (DAOS File System)
PyDAOS
libdaos (native key-value and array interfaces)
HPC Interconnect (TCP or RDMA)
Storage Nodes
DAOS Storage Engine
Storage Class Memory (byte-addressable) PMem, or DRAM+WAL
DDR-T, CXL.mem
PCIe, CXL.io
NVMe Bulk Storage (through SPDK) NAND Flash
Write-Ahead Log

Fig. 1. DAOS Software Architecture.

DAOS uses NVMe SSDs and the Storage Performance Development Kit (SPDK) [8] software to support large storage capacities and large I/O requests. SPDK provides direct, zero-copy data transfer to and from NVMe SSDs. The DAOS engine can submit multiple I/O requests to the NVMe SSDs, using SPDK queue pairs in an asynchronous manner, fully from user space. On completion of the SPDK I/O, DAOS creates indexes in SCM for the data stored on NVMe SSDs.

Intel has discontinued its Optane Persistent Memory product line [9–11]. And while it is possible that future CXL.mem attached storage devices will provide persistent, Byte-addressable storage with performance characteristics that are suitable for DAOS SCM usage, no such devices currently exist in the industry.

DAOS therefore needs an alternative code path for its Storage Class Memory functionality that does not depend on Optane PMem or PMDK. This new functionality has been developed in the DAOS *vos_on_blob* feature branch [12], also known as "*MD-on-SSD*" [13, 14], and is available as a technology preview in the DAOS 2.4 release. This paper explains the new DAOS SCM backend and presents first performance results based on an early *vos_on_blob* development snapshot.

2 DAOS Persistent Memory and Volatile Memory Code Paths

Figure 2 shows the traditional DAOS backend with persistent memory. The DAOS Versioning Object Store (VOS) data structures are stored in memory-mapped files, which reside on Optane PMem devices and are accessed through PMDK. Small (<4kiB) writes are also stored on PMem. Pointers to large *data blobs* that are stored on NVMe SSDs are maintained in the VOS trees.

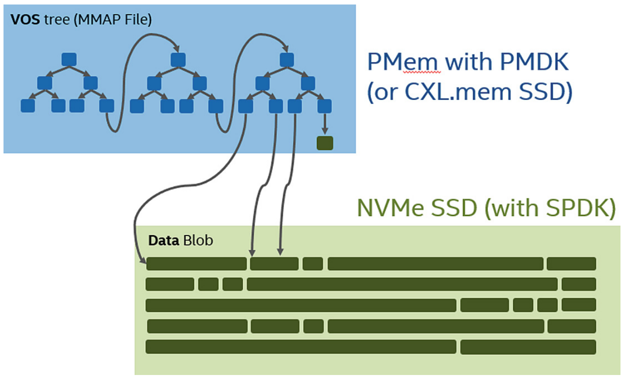


Fig. 2. DAOS Backend using Persistent Memory.

The new DAOS backend is shown in Fig. 3. Here, the VOS data structures are held in DRAM, using VOS files in a *tmpfs* filesystem that is not persistent. Small writes are also stored in DRAM. To persist the data in this volatile tmpfs filesystem, two additional functionalities are added to the code path:

1. A Write-Ahead Log (WAL) [14] performs *synchronous* commits of all write operations to a *WAL blob* that resides on NVMe SSDs. The WAL data structures are very compact to guarantee low latency. Instead of a commit block, the WAL implementation relies on checksums to ensure consistency. Depending on the operation, it is possible to perform parallel commits to the WAL.
2. The complete VOS trees are *asynchronously* checkpointed to a *meta blob* on NVMe SSDs (one per DAOS storage target). The VOS checkpointing is triggered by space pressure in the WAL blob, combined with a timed mechanism that initiates a VOS checkpoint at regular intervals even in the absence of space pressure. The detailed behaviour of the checkpointing mechanism can be fine-tuned through DAOS pool properties.

Compared to the PMem-based DAOS backend, this design has two advantages. Firstly, all reads are served from DRAM, so read performance should be higher compared to reads from Optane PMem. Secondly, by eliminating the Optane PMem hardware dependency it will now be possible to run DAOS servers on additional server hardware platforms, broadening the DAOS ecosystem.

The biggest drawback of the non-PMem design is its limited metadata capacity. The DRAM capacity per DAOS server is significantly smaller than what can be achieved with

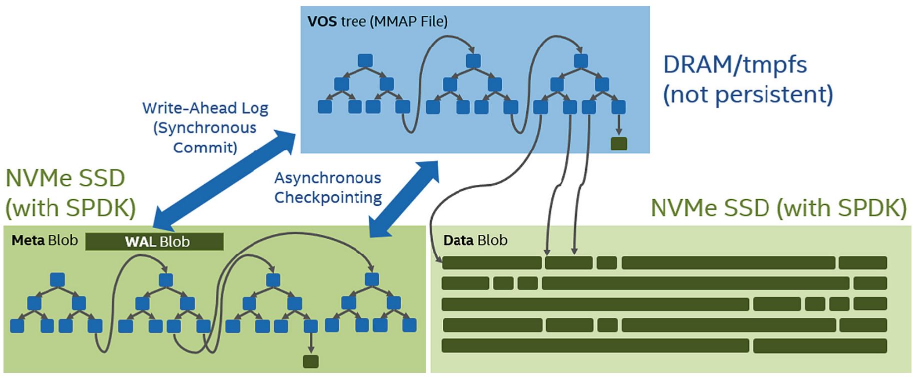

VOS tree (MMAP File)
DRAM/tmpfs (not persistent)
Write-Ahead Log (Synchronous Commit)
Asynchronous Checkpointing
NVMe SSD (with SPDK)
Meta Blob
WAL Blob
NVMe SSD (with SPDK)
Data Blob

Fig. 3. DAOS Backend using Volatile Memory.

the 128GB, 256GB or 512GB Optane PMem DIMMs. In the *phase 1* design described here, metadata is never migrated to the data blobs, so the DRAM capacity limits the total metadata capacity.

This metadata capacity limitation is addressed in two ways: Firstly, the design has been reviewed to optimize the metadata footprint wherever possible. This allows to reduce the ratio of SCM (PMem or DRAM) capacity to NVMe capacity. Secondly, a *phase 2* roadmap item for the MD-on-SSD feature will implement mechanisms that will allow parts of the metadata to be serialized and migrated to the data blobs. This design will optimize the cost effectiveness of the DAOS servers, at the expense of an increase in latency when sections of the metadata that have been migrated to data blobs need to be loaded back from NVMe SSDs into DRAM. The design of this phase 2 functionality is beyond the scope of this paper.

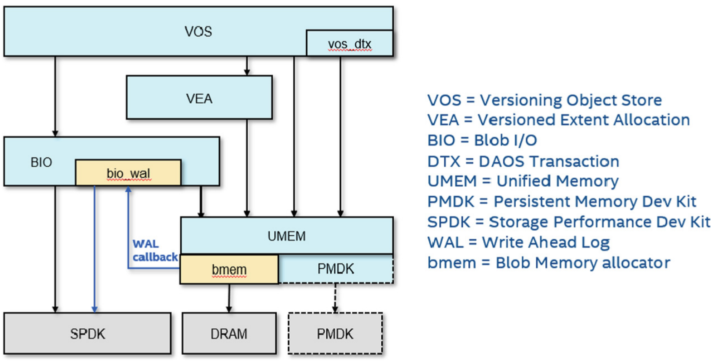

VOS
vos_dtx
VEA
BIO
bio_wal
WAL callback
UMEM
bmem
PMDK
SPDK
DRAM
PMDK
VOS = Versioning Object Store
VEA = Versioned Extent Allocation
BIO = Blob I/O
DTX = DAOS Transaction
UMEM = Unified Memory
PMDK = Persistent Memory Dev Kit
SPDK = Storage Performance Dev Kit
WAL = Write Ahead Log
bmem = Blob Memory allocator

Fig. 4. New DAOS Backend Stack Layering.

It is important to note that the changes to enable DAOS metadata on volatile memory (plus NVMe SSDs) are isolated to a few layers of the DAOS server code stack, with no changes to the DAOS API or client-side software. Figure 4 shows the code paths for the traditional PMem-based DAOS design (in blue), and the added *bmem* blob allocator and *bio_wal* module (in yellow).

3 DAOS Server Design Considerations

In Sect. 2 we have described the DAOS software architecture for DAOS servers with and without Optane Persistent Memory. In this section we are combining that software-centric view with the hardware architecture of the DAOS servers, and we discuss some of the design choices to configure a balanced DAOS server.

The performance evolution of DAOS servers across three hardware generations is described in detail in [15]. DAOS servers with Intel 2nd Gen Intel Xeon SP (formerly codenamed Cascade Lake) and 3rd Gen Intel Xeon SP (formerly codenamed Ice Lake) support Optane Persistent Memory. Configuring these servers is straightforward: DAOS starts one DAOS engine on each CPU socket, there is one SCM tier configured on the Optane PMem DIMMs that are attached to that socket, and one NVMe tier that comprises all the NVMe SSDs that are managed by that engine. Each engine uses a single HPC network port to connect to the DAOS clients.

When configuring DAOS servers with 4th Gen Intel Xeon SP (formerly codenamed Sapphire Rapids), the main design change is the fact that Optane PMem is not available on 4th Gen Intel Xeon SP. The new DAOS code path for volatile memory must be used, and the NVMe SSDs must store the Write-Ahead Log and the asynchronous VOS checkpoints in addition to the end user data.

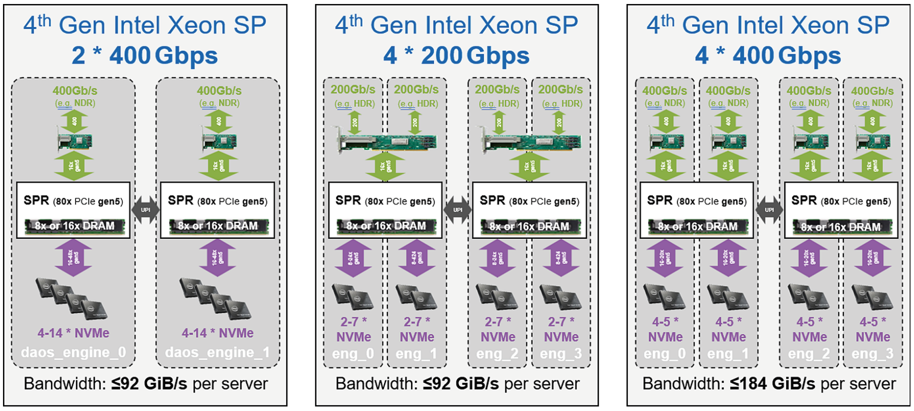


Fig. 5. DAOS Server Design Options for 4th Gen Xeon SP (from [15]).

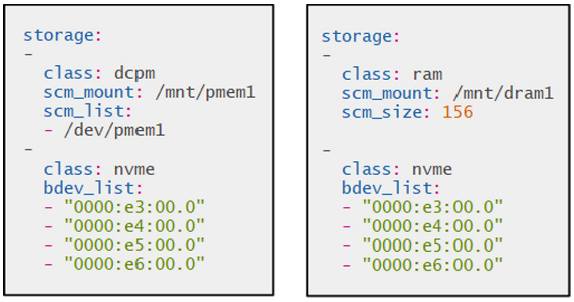
Figure 5 outlines three typical DAOS server designs with 4th Gen Intel Xeon SP. They differ primarily in the number and speed of HPC network ports, and the corresponding number of DAOS engines. As the bandwidth of an HPC network port needs to be “fed” by a corresponding NVMe SSD bandwidth, these differences also imply a different minimum and maximum number of NVMe SSDs per DAOS engine. These boundaries are relevant for the placement of WAL, VOS checkpoints and user data.

- The continuation of the original DAOS server design of a 2-socket server with one DAOS engine per CPU socket is shown on the left of Fig. 5. To saturate a 400 Gbps HPC network link (e.g., NDR InfiniBand), at least *four* PCIe gen5 NVMe SSDs per DAOS engine are needed. The maximum of 14 NVMe SSDs per DAOS engine shown in the figure is determined by the total number of 80 PCIe gen5 lanes per CPU socket. (Some PCIe lanes are used for other functions, for example a management LAN card, so using 16 NVMe SSDs per DAOS engine is not possible).
- The middle of Fig. 5 shows a server design to support 200 Gbps HPC fabrics, by using PCIe gen5 NICs with two 200 Gbps ports (e.g., dual HDR InfiniBand) instead of a single 400 Gbps port. Because DAOS requires exactly one HPC network port per engine, this configuration requires to run two DAOS engines per CPU socket with 2–7 NVMe SSDs per engine.
- Due to the higher number of PCIe lanes of 4th Gen Intel Xeon SP compared to previous generations, it is now also possible to populate a 2-socket DAOS server with *four* HPC network cards, doubling the peak bandwidth to $\leq$184 GiB/s. This is shown on the right of Fig. 5, together with the resulting 4–5 PCIe gen5 NVMe SSDs per engine to feed those network ports.

The DAOS server configuration with four 400 Gbps network links and 4–5 PCIe gen5 NVMe SSDs provides the best bandwidth. It is the standard configuration for traditional supercomputing environments that use a 400 Gbps HPC fabric. For this reason, the remainder of this study focuses on a DAOS configuration with *four* NVMe SSDs per DAOS engine. The other configurations in Fig. 5 are more suitable for capacity-oriented solutions, or in environments that only operate 200 Gbps (or 100 Gbps) fabrics and cannot fully utilize the PCIe gen5 networking capabilities of a Sapphire Rapids based DAOS server.

The discussion of these different hardware designs highlights a critical question that needs to be answered when using the non-PMem DAOS code path: Because the NVMe SSDs now hold *three* different types of data (WAL, VOS checkpoints, and user data), it must be decided which of the NVMe SSDs should be used for which purpose.

Figure 6 shows the traditional storage configuration options in the DAOS servers’ configuration file: There is one SCM tier of class “dcpm” using Optane PMem, and one NVMe tier for the user data. For testing and for ephemeral storage use cases, persistent memory can be emulated by an SCM tier of class “ram”. But such a configuration will not survive failures or engine restarts and does *not* use the MD-on-SSD code path.

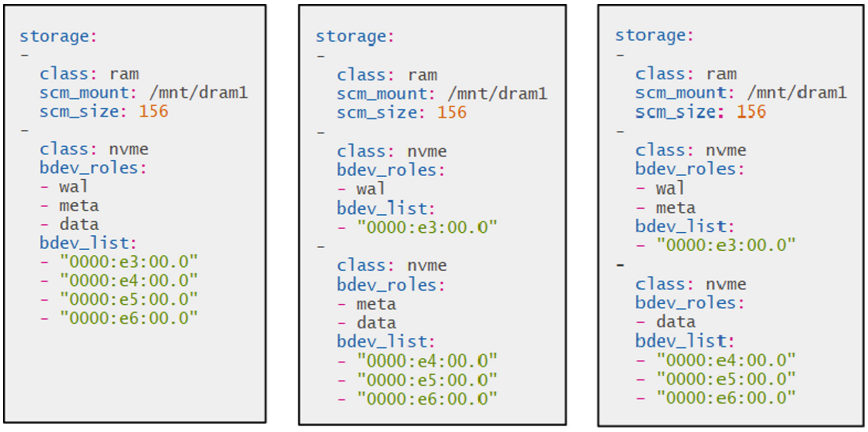


```
storage:
-
  class: dcpm
  scm_mount: /mnt/pmem1
  scm_list:
  - /dev/pmem1
-
  class: nvme
  bdev_list:
  - "0000:e3:00.0"
  - "0000:e4:00.0"
  - "0000:e5:00.0"
  - "0000:e6:00.0"
```

```
storage:
-
  class: ram
  scm_mount: /mnt/dram1
  scm_size: 156
-
  class: nvme
  bdev_list:
  - "0000:e3:00.0"
  - "0000:e4:00.0"
  - "0000:e5:00.0"
  - "0000:e6:00.0"
```

Fig. 6. Traditional Storage Configuration Options in daos_server.yml: Optane PMem-based configuration (left); Ephemeral DRAM-based configuration (right).

In DAOS 2.4, the MD-on-SSD code path will only be used if it is explicitly enabled by specifying the new *bdev_role* property for the NVMe storage tier(s) in the daos_server.yml file. There are three types of *bdev_role: wal, meta,* and *data*. Each role must be assigned to *exactly one* NVMe tier. Depending on the number of NVMe SSDs per DAOS engine there may be one, two or three NVMe tiers with different *bdev_role* assignments.



```
storage:
-
  class: ram
  scm_mount: /mnt/dram1
  scm_size: 156
-
  class: nvme
  bdev_roles:
  - wal
  - meta
  - data
  bdev_list:
  - "0000:e3:00.0"
  - "0000:e4:00.0"
  - "0000:e5:00.0"
  - "0000:e6:00.0"
```

```
storage:
-
  class: ram
  scm_mount: /mnt/dram1
  scm_size: 156
-
  class: nvme
  bdev_roles:
  - wal
  bdev_list:
  - "0000:e3:00.0"
-
  class: nvme
  bdev_roles:
  - meta
  - data
  bdev_list:
  - "0000:e4:00.0"
  - "0000:e5:00.0"
  - "0000:e6:00.0"
```

```
storage:
-
  class: ram
  scm_mount: /mnt/dram1
  scm_size: 156
-
  class: nvme
  bdev_roles:
  - wal
  - meta
  bdev_list:
  - "0000:e3:00.0"
-
  class: nvme
  bdev_roles:
  - data
  bdev_list:
  - "0000:e4:00.0"
  - "0000:e5:00.0"
  - "0000:e6:00.0"
```

Fig. 7. MD-on-SSD Storage Configuration Options in daos_server.yml: Left: One NVMe tier; wal/meta/data shared on four SSDs (*4wmd*). Center: Two NVMe tiers; wal on one SSD, meta/data shared on three SSDs (*1w3md*). Right: Two NVMe tiers; wal/meta shared on one SSD, data on three SSDs (*1wm3d*).

Figure 7 shows three different storage configurations that represent three possible scenarios for a DAOS engine with four NVMe SSDs and MD-on-SSD enabled:

- Figure 7 (left) shows the typical use case for a DAOS server with a small number of NVMe SSDs. With only four or five NVMe SSDs per engine, it is natural to assign all three roles to all NVMe SSDs configured as a *single* NVMe tier.
- Figure 7 (center) presents a configuration where one NVMe SSD is dedicated for the *wal*, while the remaining three NVMe SSDs are assigned to hold the VOS checkpoints (*meta*) and the user *data*. Using two NVMe tiers makes it possible to use a higher endurance and higher performance SSD for the *wal* tier. But note that the performance of a single high-performance SSD may still be lower than the aggregate performance of multiple lower-performance SSDs in the previous scenario.
- Figure 7 (right) also uses two NVMe tiers but co-locates the *wal* and *meta* blobs on the same tier. This may be a better choice than co-locating *meta* and *data* if the endurance of the *data* NVMe SSDs is too low for the relatively frequent VOS checkpointing. But it may also hurt performance by reducing the number of NVMe SSDs available for VOS checkpointing from three to one.

The third option to use two NVMe tiers would be to co-locate *wal* and *data* and dedicate the other tier for *meta*. But the main rationale for a separate *wal* tier is the ability to use a small number of higher-performance SSDs for the synchronous Write-Ahead Log. If *wal* is co-located with *data* then this rationale disappears, and it is better to use a single tier for all three roles. And finally, using three NVMe tiers (one per role) is not reasonable when there are only four NVMe SSDs per engine.

The three MD-on-SSD configurations shown in Fig. 7 will be benchmarked in the next section. They will be compared with the traditional PMem-based configuration shown in Fig. 6 (left), which consist of a single SCM tier and a single NVMe tier with *no* specification of a *bdev_role* for the NVMe tier.

In this study we have fixed the total number of NVMe SSDs per engine to *four* to make performance comparisons easier. But it should be noted that in a production environment the number of NVMe SSDs in the *data* tier should not be reduced below four, because that is typically the minimum number of NVMe SSDs required to saturate the bandwidth of the HPC network link. This implies that for the performance configuration of Fig. 5 (right), a separation into two NVMe tiers is only reasonable if there are *five* NVMe SSDs: Four NVMe SSDs with the *data* role provide sufficient aggregate bandwidth to saturate the engine's HPC network port, and the 5$^{\text{th}}$ NVMe SSD can hold the *wal* blobs.

4 Performance Results

To compare the metadata performance of DAOS servers with Optane PMem to the performance of the new DAOS code path, the mdtest [16] benchmark with the DAOS DFS API [17] has been run on a DAOS server with Intel 8352Y CPUs (32 cores, 2.2 GHz). Only a single DAOS engine has been configured on one CPU, to keep the testing setup simple. All tests have used four Intel/Solidigm D7-P5500 3.84 TB PCIe gen4 NVMe SSDs, see [18] for performance specs. The PMem configuration uses 8x 128GB Optane 200 Series PMem DIMMs [9], and DRAM configurations use a tmpfs of size 156GB configured on the 16x 16GB DDR4 DRAM DIMMs of the server.

For DAOS Servers based on Optane PMem, earlier studies have clearly shown that DAOS metadata rates scale with the number of *targets* per engine, where each target runs on a dedicated physical CPU core. Figure 11 of [19] shows this relationship, varying the number of targets per engine from 4 to 32. The current study uses *24 targets* and *six xs_helper* threads for all benchmarks.

The DAOS clients are 2-socket nodes using the same CPUs, connected through a fully non-blocking InfiniBand HDR fabric with NVIDIA QM8790 HDR switches. All mdtest benchmarks have been performed with a stonewall time of 60 s for the write phases. Three sets of benchmarks have been performed, following the setup of the IO500 [4] with one addition:

- *mdtest-easy*: Metadata operations on 0-Byte files, using separate directories for each MPI task. No user data is written to the files.
- *mdtest-hard*: Metadata operations on small (3901 Byte) files in a shared directory. The size of the user data is below the DAOS cut-off limit of 4kiB, so all writes will go to the metadata in PMem or DRAM (plus WAL and metadata/checkpoints).
- *mdtest-hard2*: Same setup as mdtest-hard but using 2x the file size. Here the user data is bigger than 4kiB and will be written to the data blobs on NVMe SSDs.

All benchmarks have been run on one client node and 16 client nodes, and for each node count the number of MPI tasks per node has been varied from 1 to 64, for a total of 1 to 1024 MPI tasks. All performance graphs in this section show the total number of MPI tasks of a benchmark on the x-axis (in log scale), and the different numbers of tasks per node for a given node count are shown as a line-graph of the same color.

For each mdtest benchmark, the following storage configurations (shown in Fig. 6 and Fig. 7) have been measured, with labels in the graphs indicating the engine configuration:

- *PMEM, 4d*: PMem-based configuration with four NVMe SSDs for data;
- *DRAM, 4wmd*: DRAM-based configuration with one NVMe tier, four NVMe SSDs are shared for all three roles (*wal*, *meta* and *data*);
- *DRAM, 1w3md*: DRAM-based configuration with two NVMe tiers, one NVMe SSD dedicated to *wal* and three NVMe SSDs shared for *meta* and *data*;
- *DRAM, 1wm3d*: DRAM-based configuration with two NVMe tiers, one NVMe SSD shared for *wal* and *meta* and three NVMe SSDs dedicated for *data.*

A DAOS pool with an NVMe size of 8 TB and one POSIX container has been used for all mdtest benchmarks. The SCM size was set to 500 GB for the PMem configuration (a ratio of 1:16 = 6.25%), while an SCM size of 125 GB (a ratio of 1:64 = 1.56%) was used for the DRAM configurations.

Figure 8 shows the performance result for the *mdtest-easy* benchmark. Create rates in Fig. 8 (A) are highest when using PMem, indicating that the writing of the WAL and VOS checkpoints to NVMe does slow down the achievable create rate. However, the difference from the DRAM-based configurations using MD-on-SSD is relatively small. Among those, co-locating the WAL and VOS checkpoints on a single NVMe SSD (*1wm3d*) results in the lowest performance. Assigning all roles to a single NVMe tier with four NVMe SSDs (*4wmd*) results in roughly the same performance as sharing three NVMe SSDs for VOS checkpoints and data (*1w3md*), and both are slightly ahead of the *1wm3d* case. This suggests that a higher number of NVMe SSDs is beneficial for the VOS checkpointing. The stat operations in Fig. 8 (B) are read-only operations, so there is no need for *wal* logging and consequently no noticeable performance difference between the four configurations. When deleting the 0-Byte files, Fig. 8 (D) shows a similar behaviour to the create case, which is expected as this operation generates a similar load for the *wal* and *meta* blobs. Overall, the achievable performance is very comparable for all four server configurations. The noticeable degradation at high task counts (particularly for the stat rates) is present for all PMem- and DRAM-based server configurations. This will be subject to further studies.

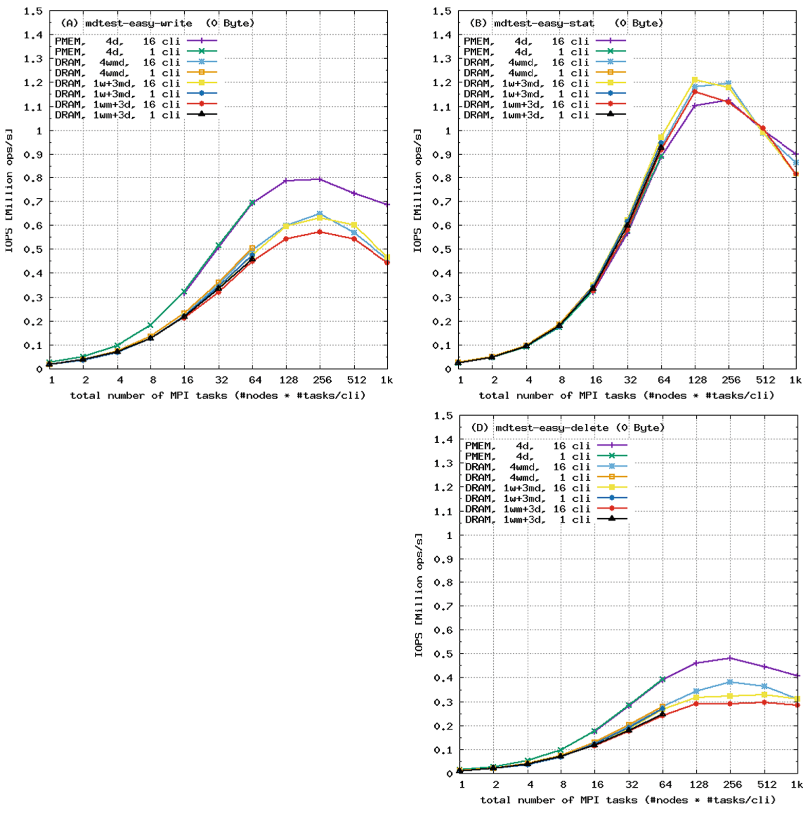


Fig. 8. DAOS Metadata Performance Scaling (mdtest-easy, 0-Byte files): (A) write, (B) stat, no read test (C) for 0-Byte files, (D) delete.

Figure 9 reports the performance for the *mdtest-hard* testcase. Absolute values of the metadata rates are lower because actual data is written and read (instead of just creating empty files). But qualitatively the performance graphs are very similar to Fig. 8, with all three MD-on-SSD configurations showing comparable performance and the PMem-based configuration showing slightly better write and delete rates.

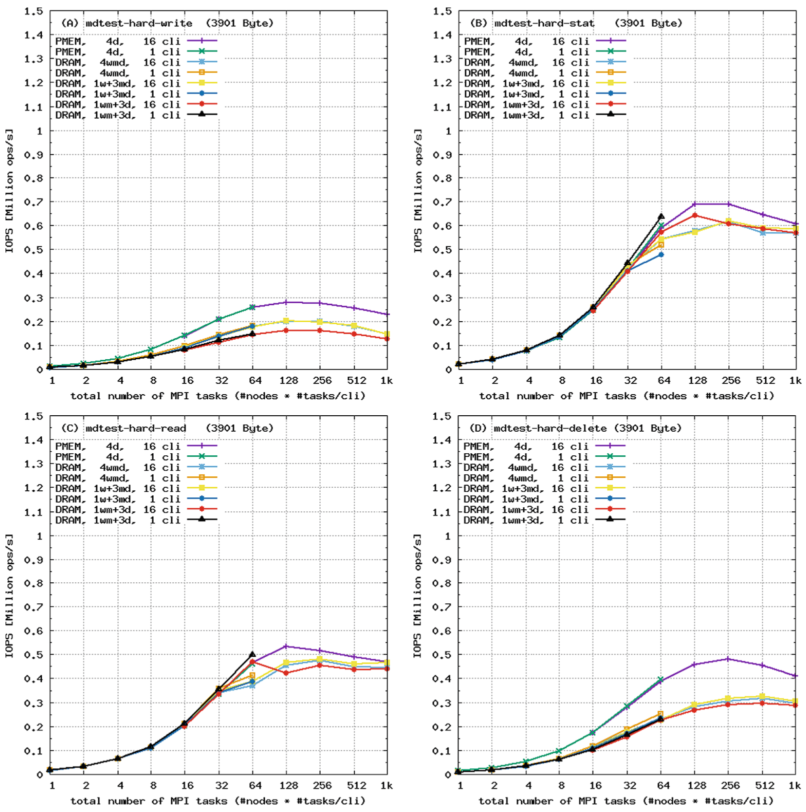


Fig. 9. DAOS Metadata Performance Scaling (mdtest-hard, 3901-Byte files): (A) write, (B) stat, (C) read, (D) delete.

Finally, Fig. 10 repeats the *mdtest-hard* benchmarks but uses an *increased file size* of 7802 Byte to ensure that data is written to the data blobs on NVMe SSDs. The read phase is slower than in Fig. 9, which is expected due to the increased file size. The write, stat and delete performance is almost identical to the performance with the smaller file size. It would be interesting to study this test case with a higher number of NVMe SSDs per engine – four SSDs are probably a too small number to see significant differences between the different configuration options.

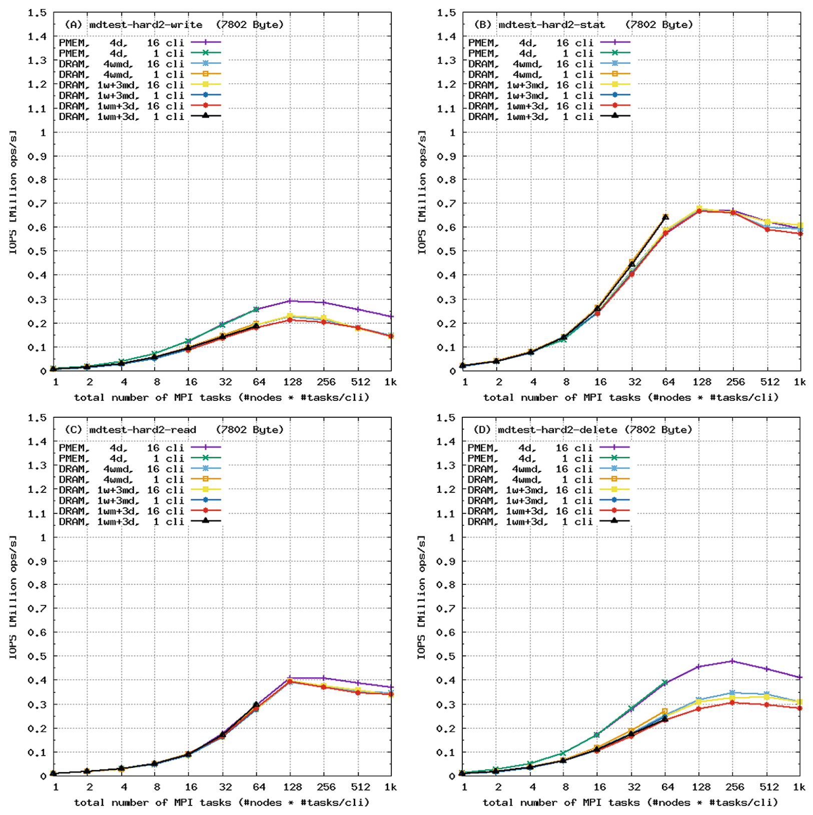


Fig. 10. DAOS Metadata Performance Scaling (mdtest-hard2, 7802-Byte files): (A) write, (B) stat, (C) read, (D) delete.

5 Summary and Conclusions

We have presented the new DAOS code path to support DAOS servers without persistent memory, as an alternative to the existing Optane PMem-based DAOS design that continues to be supported. Code changes are localized to a few code layers on the server side, with no changes to the DAOS APIs or the DAOS clients. Initial studies using a technology preview of the feature are very encouraging. Performance results with an early implementation of the *vos_on_blob* feature show very similar metadata rates across the tested server configurations. Future work will focus on optimizing the implementation for metadata space efficiency, as well as for performance.

The behaviour of VOS checkpointing can be fine-tuned through DAOS pool properties: The *checkpoint* property can be set to either "*timed*" or "*lazy*" mode. The "*timed*" mode is the default and triggers a VOS checkpoint every *checkpoint_freq* seconds – or whenever space pressure in the WAL builds up. In *"lazy"* mode, only space pressure (fill grade of *checkpoint_thresh* percent or higher) will trigger a VOS checkpoint. All benchmarks in this study have been performed with the default ("*timed*" mode with a

5 s interval). More extensive studies are necessary to better understand how to optimize these tunables for specific workloads or NVMe storage media characteristics.

References

1. Liang, Z., Lombardi, J., Chaarawi, M., Hennecke, M.: DAOS: a scale-out high performance storage stack for storage class memory. In: Panda, D.K. (ed.) SCFA 2020. LNCS, vol. 12082, pp. 40–54. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48842-0_3
2. Liang, Z., Fan, Y., Wang, D., Lombardi, J.: Distributed transaction and self-healing system of DAOS. In: Nichols, J., Verastegui, B., Maccabe, A.‘, Hernandez, O., Parete-Koon, S., Ahearn, T. (eds.) SMC 2020. CCIS, vol. 1315, pp. 334–348. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63393-6_22
3. Scot Breitenfeld, M., et al.: DAOS for extreme-scale systems in scientific applications (2017). https://arxiv.org/pdf/1712.00423.pdf
4. IO500. https://io500.org/
5. Rudoff, A.: APIs for persistent memory programming (2018). https://storageconference.us/2018/Presentations/Rudoff.pdf
6. Scargall, S.: Programming Persistent Memory. Apress, Berkeley ISBN 978-1-4842-4931-4 (2020). https://doi.org/10.1007/978-1-4842-4932-1
7. PMDK. https://pmem.io/pmdk/
8. SPDK. https://spdk.io/
9. Intel® Optane™ Persistent Memory 200 Series. https://ark.intel.com/content/www/us/en/ark/products/series/203877/intel-optane-persistent-memory-200-series.html
10. Support for Intel® Optane™ Persistent Memory 200 Series. https://www.intel.com/content/www/us/en/support/products/203877/memory-and-storage/intel-optane-persistent-memory/intel-optane-persistent-memory-200-series.html
11. Customer Support Options for Discontinued Intel Optane Solid-State Drives and Modules. https://www.intel.com/content/www/us/en/support/articles/000024320.html
12. DAOS vos_on_blob Feature Branch (build daos-2.3.106-2.9536.ge3c942ec.el8.x86_64). https://github.com/daos-stack/daos/tree/feature/vos_on_blob
13. Lombardi, J., et al.: Metadata on SSDs design documentation (2022). https://daosio.atlassian.net/wiki/spaces/DC/pages/11196923911/Metadata+on+SSDs
14. Niu, Y.: WAL detailed design (2022). https://daosio.atlassian.net/wiki/spaces/DC/pages/11215339529/WAL+Detailed+Design
15. Hennecke, M.: Performance evolution of DAOS servers (2023). https://www.intel.com/content/www/us/en/high-performance-computing/performance-evolution-of-daos-servers.html
16. IOR and mdtest github repository. https://github.com/hpc/ior
17. The DFS (DAOS File System) driver. https://github.com/hpc/ior/blob/main/README_DAOS
18. Solidigm/Intel SSD D7-P5510 Product Brief. https://www.solidigm.com/content/dam/solidigm/en/site/products/data-center/d7/p5510/documents/d7-p5510-series-product-brief.pdf
19. Hennecke, M.: Understanding DAOS storage performance scalability. In: International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCASIA-WORKSHOP 2023), 27 February–2 March 2023, Raffles Blvd, Singapore (2023). https://doi.org/10.1145/3581576.3581577
20. Migration from Direct-Attached Intel Optane Persistent Memory to CXL-Attached Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory-to-cxl-attached-memory.html