







DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory

Zhen Liang¹, Johann Lombardi², Mohamad Chaarawi³,
and Michael Hennecke⁴

¹ Intel China Ltd., GTC, No. 36 3rd Ring Road, Beijing, China
liang.zhen@intel.com

² Intel Corporation SAS, 2 rue de Paris, 92196 Meudon Cedex, France
johann.lombardi@intel.com

³ Intel Corporation, 1300 S MoPac Expy, Austin, TX 78746, USA
mohamad.chaarawi@intel.com

⁴ Lenovo Global Technology Germany GmbH, Am Zehnthof 77,
45307 Essen, Germany
mhennecke@lenovo.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. Its advanced storage API enables the native support of structured, semi-structured and unstructured data models, overcoming the limitations of traditional POSIX based parallel filesystem. For HPC workloads, DAOS provides direct MPI-IO and HDF5 support as well as POSIX access for legacy applications. In this paper we present the architecture of the DAOS storage engine and its high-level application interfaces. We also describe initial performance results of DAOS for IO500 benchmarks.

Keywords: DAOS · SCM · Persistent memory · NVMe · Distributed storage system · Parallel filesystem · SWIM · RAFT

1 Introduction

The emergence of data-intensive applications in business, government and academia stretches the existing I/O models beyond limits. Modern I/O workloads feature an increasing proportion of metadata combined with misaligned and fragmented data. Conventional storage stacks deliver poor performance for these workloads by adding a lot of latency and introducing alignment constraints. The advent of affordable large-capacity persistent memory combined with a high-speed fabric offers a unique opportunity to redefine the storage paradigm and support modern I/O workloads efficiently.

This revolution requires a radical rethinking of the complete storage stack. To unleash the full potential of these new technologies, the new stack must embrace a byte-granular shared-nothing interface from the ground up. It also has to be able to

support massively distributed storage for which failure will be the norm, while preserving low latency and high bandwidth access over the fabric.

DAOS is a complete I/O architecture that aggregates SCM and NVMe storage distributed across the fabric into globally accessible object address spaces, providing consistency, availability and resiliency guarantees without compromising performance.

Section 2 of this paper describes the challenges that SCM and NVMe storage pose to traditional I/O stacks. Section 3 introduces the architecture of DAOS and explains how it integrates with new storage technologies. Section 4 gives an overview of the data model and I/O interfaces of DAOS, and Sect. 5 presents the first IO500 performance results of DAOS.

2 Constraints of Using Traditional Parallel Filesystems

Conventional parallel filesystems are built on top of block devices. They submit I/O through the OS kernel block I/O interface, which is optimized for disk drives. This includes using an I/O scheduler to optimize disk seeking, aggregating and coalescing writes to modify the characteristics of the workloads, then sending large streaming data to the disk drive to achieve the high bandwidth. However, with the emergence of new storage technologies like 3D-XPoint that can offer several orders of magnitude lower latency comparing with traditional storage, software layers built for spinning disk become pure overhead for those new storage technologies.

Moreover, most parallel filesystems can use RDMA capable network as a fast transport layer, in order to reduce data copying between layers. For example, transfer data from the page cache of a client to the buffer cache of a server, then persist it to block devices. However, because of lacking unified polling or progress mechanisms for both block I/O and network events in the traditional storage stack, I/O request handling heavily relies on interrupts and multi-threading for concurrent RPC processing. Therefore, context switches during I/O processing will significantly limit the advantage of the low latency network.

With all the thick stack layers of traditional parallel filesystem, including caches and distributed locking, user can still use 3D NAND, 3D-XPoint storage and high speed fabrics to gain some better performance, but will also lose most benefits of those technologies because of overheads imposed by the software stack.

3 DAOS, a Storage Stack Built for SCM and NVMe Storage

The **D**istributed **A**synchronous **O**bject **S**torage (DAOS) is an open source software-defined object store designed from the ground up for massively distributed Non Volatile Memory (NVM). It presents a key-value storage interface and provides features such as transactional non-blocking I/O, a versioned data model, and global snapshots.

This section introduces the architecture of DAOS, discusses a few core components of DAOS and explains why DAOS can be a storage system with both high performance and resilience.

3.1 DAOS System Architecture

DAOS is a storage system that takes advantage of next generation NVM technology like Storage Class Memory (SCM) and NVM express (NVMe). It bypasses all Linux kernel I/O, it runs end-to-end in user space and does not do any system call during I/O.

As shown in Fig. 1, DAOS is built over three building blocks. The first one is persistent memory and the Persistent Memory Development Toolkit (PMDK) [2]. DAOS uses it to store all internal metadata, application/middleware key index and latency sensitive small I/O. During starting of the system, DAOS uses system calls to initialize the access of persistent memory. For example, it maps the persistent memory file of DAX-enabled filesystem to virtual memory address space. When the system is up and running, DAOS can directly access persistent memory in user space by memory instructions like load and store, instead of going through a thick storage stack.

Persistent memory is fast but has low capacity and low cost effectiveness, so it is effectively impossible to create a large capacity storage tier with persistent memory only. DAOS leverages the second building block, NVMe SSDs and the Storage Performance Development Kit (SPDK) [7] software, to support large I/O as well as higher latency small I/O. SPDK provides a C library that may be linked into a storage server that can provide direct, zero-copy data transfer to and from NVMe SSDs. The DAOS service can submit multiple I/O requests via SPDK queue pairs in an asynchronous manner fully from user space, and later creates indexes for data stored in SSDs in persistent memory on completion of the SPDK I/O.

Libfabric [8] and an underlying high performance fabric such as Omni-Path Architecture or InfiniBand (or a standard TCP network), is the third build block for DAOS. Libfabric is a library that defines the user space API of OFI, and exports fabric communication services to application or storage services. The transport layer of DAOS is built on top of Mercury [9] with a libfabric/OFI plugin. It provides a callback based asynchronous API for message and data transfer, and a thread-less polling API for progressing network activities. A DAOS service thread can actively poll network events from Mercury/libfabric as notification of asynchronous network operations, instead of using interrupts that have a negative performance impact because of context switches.

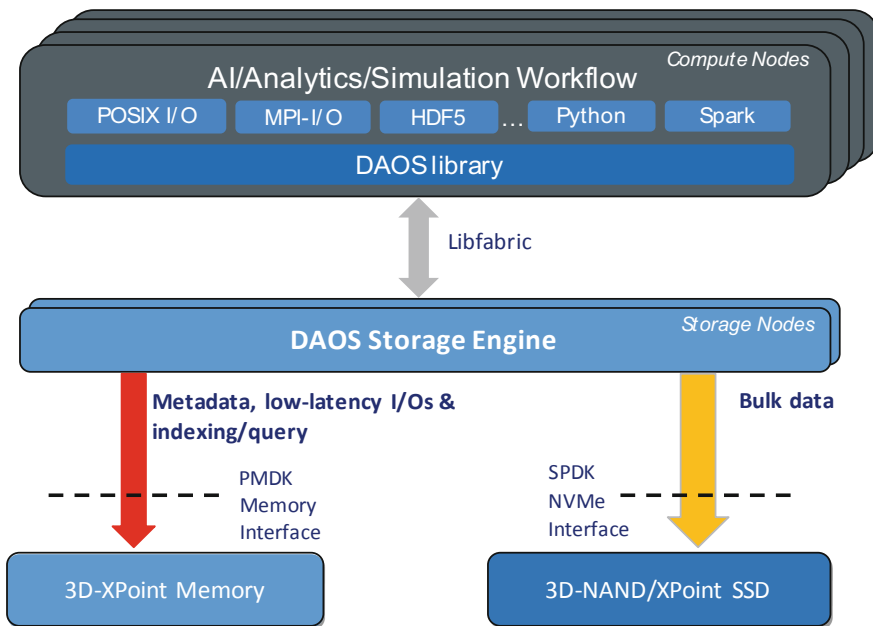


Fig. 1. DAOS system architecture

As a summary, DAOS is built on top of new storage and network technologies and operates fully in user space, bypassing all the Linux kernel code. Because it is architected specifically for SCM and NVMe, it cannot support disk based storage. Traditional storage system like Lustre [11], Spectrum Scale [12], or CephFS [10] can be used for disk-based storage, and it is possible to move data between DAOS and such external file systems.

3.2 DAOS I/O Service

From the perspective of stack layering, DAOS is a distributed storage system with a client-server model. The DAOS client is a library that is integrated with the application, and it runs in the same address space as the application. The data model exposed by the DAOS library is directly integrated with all the traditional data formats and middleware libraries that will be introduced in Sect. 4.

The DAOS I/O server is a multi-tenant daemon that runs either directly on a data storage node or in a container. It can directly access persistent memory and NVMe SSDs, as introduced in the previous section. It stores metadata and small I/O in persistent memory, and stores large I/O in NVMe SSDs. The DAOS server does not rely on spawning pthreads for concurrent handling of I/O. Instead it creates an Argobots [6] User Level Thread (ULT) for each incoming I/O request. An Argobots ULT is a lightweight execution unit associated with an execution stream (xstream), which is mapped to the pthread of the DAOS service. This means that conventional POSIX I/O function calls, pthread locks or synchronous message waiting calls from any ULT can

block progress of all ULTs on an execution stream. However, because all building blocks used by DAOS provide a non-blocking user space interface, a DAOS I/O ULT will never be blocked on system calls. Instead it can actively yield the execution if an I/O or network request is still inflight. The I/O ULT will eventually be rescheduled by a system ULT that is responsible for polling a completion event from the network and SPDK. ULT creation and context switching are very lightweight. Benchmarks show that one xstream can create millions of ULTs per second, and can do over ten million ULT context switches per second. It is therefore a good fit for DAOS server side I/O handling, which is supposed to support micro-second level I/O latency (Fig. 2).

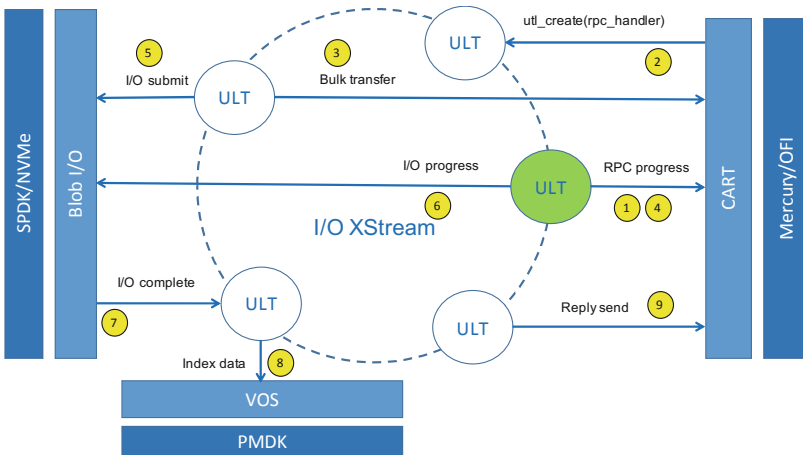


Fig. 2. DAOS server side I/O processing

3.3 Data Protection and Data Recovery

DAOS storage is exposed as objects that allow user access through a key-value or key-array API. In order to avoid scaling problems and the overhead of maintaining per-object metadata (like object layout that describes locality of object data), a DAOS object is only identified by a 128-bit ID that has a few encoded bits to describe data distribution and the protection strategy of the object (replication or erasure code, stripe count, etc.). DAOS can use these bits as hints, and the remaining bits of the object ID as a pseudorandom seed to generate the layout of the object based on the configuration of the DAOS storage pool. This is called algorithmic object placement. It is similar to the data placement technology of Ceph, except DAOS is not using CRUSH [10] as the algorithm.

This paper will only describe the data protection and recovery protocol from a high level view. Detailed placement algorithm and recovery protocol information can be found in the online DAOS design documents [5].

Data Protection

In order to get ultra-low latency I/O, a DAOS storage server stores application data and metadata in SCM connected to the memory bus, and on SSDs connected over PCIe. The DAOS server uses load/store instructions to access memory-mapped persistent memory, and the SPDK API to access NVMe SSDs from user space. If there is an uncorrectable error in persistent memory or an SSD media corruption, applications running over DAOS without additional protection would incur a data/metadata loss. In order to guarantee resilience and prevent data loss, DAOS provides both replication and erasure coding for data protection and recovery.

When data protection is enabled, DAOS objects can be replicated, or chunked into data and parity fragments, and then stored across multiple storage nodes. If there is a storage device failure or storage node failure, DAOS objects are still accessible in degraded mode, and data redundancy is recoverable from replicas or parity data [15].

Replication and Data Recovery

Replication ensures high availability of data because objects are accessible while any replica survives. Replication of DAOS is using a primary-slave protocol for write: The primary replica is responsible for forwarding requests to slave replicas, and progressing distributed transaction status.

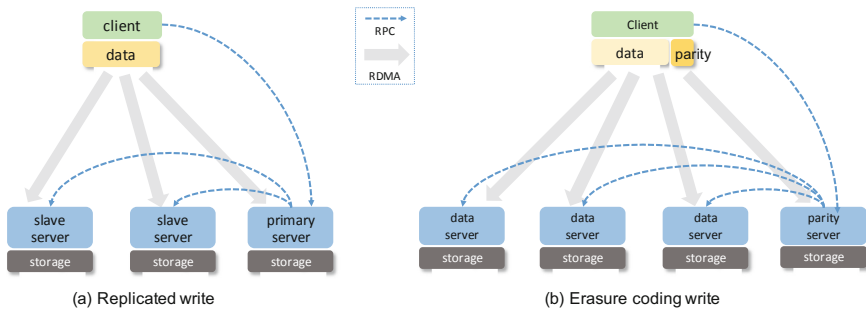


Fig. 3. Message and data flow of replication and erasure coding

The primary-slave model of DAOS is slightly different from a traditional replication model, as shown in Fig. 3a. The primary replica only forwards the RPC to slave replica servers. All replicas will then initiate an RDMA request and get the data directly from the client buffer. DAOS chooses this model because in most HPC environments, the fabric bandwidth between client and server is much higher than the bandwidth between servers (and the bandwidth between servers will be used for data recovery and rebalance). If DAOS is deployed for a non-HPC use case that has higher bandwidth between servers, then the data transfer path of DAOS can be changed to the traditional model.

DAOS uses a variant of two-phase commit protocol to guarantee atomicity of the replicated update: If one replica cannot apply the change, then all replicas should abandon the change as well. This protocol is quite straightforward if there is no failure.

However, if a server handling the replication write failed during the two-phase transaction, DAOS will not follow the traditional two-phase commit protocol that would wait for the recovery of the failed node. Instead it excludes the failed node from the transaction, then algorithmically selects a different node as a replacement, and moves forward the transaction status. If the failed-out node comes back at some point, it ignores its local transaction status and relies on the data recovery protocol to catch up the transaction status.

When the health monitoring system of DAOS detected a failure event of a storage target, it reports the event to the highly replicated RAFT [14] based pool service, which can globally activate the rebuild service on all storage servers in the pool. The rebuild service of a DAOS server can promptly scan object IDs stored in local persistent memory, independently calculates the layout of each object, and then finds out all the impacted objects by checking if the failed target is within their layouts. The rebuild service also sends those impacted object IDs to algorithmically selected fallback storage servers. These fallback servers then reconstruct data for impacted objects by pulling data from the surviving replicas.

In this process, there is no central place to perform data/metadata scans or data reconstruction: The I/O workload of the rebuild service will be fully declustered and parallelized.

Erasur Coding and Data Recovery

DAOS can also support erasure coding (EC) for data protection, which is much more space and bandwidth efficient than replication but requires more computation.

Because the DAOS client is a lightweight library which is linked with the application on compute nodes that have way more compute resource than the DAOS servers, the data encoding is handled by the client on write. The client computes the parity, creates RDMA descriptors for both data and parity fragments, and then sends an RPC request to the leader server of the parity group to coordinate the write. The RPC and data flow of EC is the same as replication: All the participants of an EC write should directly pull data from the client buffer, instead of pulling data from the leader server cache (Fig. 3b). DAOS EC also uses the same two-phase commit protocol as replication to guarantee the atomicity of writes to different servers.

If the write is not aligned with the EC stripe size, most storage systems have to go through a read/encode/write process to guarantee consistency of data and parity. This process is expensive and inefficient, because it will generate much more traffic than the actual I/O size. It also requires distributed locking to guarantee consistency between read and write. With its multi-version data model, DAOS can avoid this expensive process by replicating only the partial write data to the parity server. After a certain amount of time, if the application keeps writing and composes a full stripe eventually, the parity server can simply compute the parity based on all this replicated data. Otherwise, the parity server can coordinate other servers in the parity group to generate a merged view from the partial overwritten data and its old version, then computes parity for it and stores the merged data together with that new parity.

When a failure occurs, a degraded mode read of EC-protected data is more heavy-weight compared to replication: With replication, the DAOS client can simply switch to read from a different replica. But with EC, the client has to fetch the full data stripe and

has to reconstruct the missing data fragment in flight. The processing of degraded mode write of EC-protected data is the same as for replication: The two-phase commit transaction can continue without being blocked by the failed-out server, instead it can immediately proceed as soon as a fallback server is selected for the transaction.

The rebuild protocol of EC is also similar to replication, but it will generate significantly more data movement compared to replication. This is a characteristic of all parity based data protection technologies.

End to End Data Integrity

There are three types of typical failures in DAOS storage system:

- Service crash. DAOS captures this by running the gossip-like protocol SWIM [13].
- NVMe SSD failure. DAOS can detect this type of failure by polling device status via SPDK.
- Data corruption caused by storage media failure. DAOS can detect this by storing and verifying checksums.

In order to support end-to-end checksums and detect silent data corruption, before writing the data to server the DAOS client computes checksums for the data being written. When receiving the write, the DAOS server can either verify the checksums, or store the checksums and data directly without verification. The server side verification can be enabled or disabled by the user, based on performance requirements.

When an application reads back the data, if the read is aligned with the original write then server can just return the data and checksum. If the read is not aligned with the original write, the DAOS server verifies the checksums for all involved data extents, then computes the checksum for the part of data being read, and returns both data and checksum to the client. The client then verifies the checksum again before returning data to the application. If the DAOS client detects a checksum error on read, it can enable degraded mode for this particular object, and either switch to another replica for the read, or reconstruct data in flight on the client if it is protected by EC. The client also reports the checksum error back to the server. A DAOS server will collect all checksum errors detected by local verification and scrubbing, as well as errors reported by clients. When the number of errors reaches a threshold, the server requests the pool service to exclude the bad device from the storage system, and triggers data recovery for it.

Checksums of DAOS are stored in persistent memory, and are protected by the ECC of the persistent memory modules. If there is an uncorrectable error in persistent memory, the storage service will be killed by SIGBUS. In this case the pool service will disable the entire storage node, and starts data recovery on the surviving nodes.

4 DAOS Data Model and I/O Interface

This section describes the data model of DAOS, the native API built for this data model, and explains how a POSIX namespace is implemented over this data model.

4.1 DAOS Data Model

The DAOS data model has two different object types: Array objects that allow an application to represent a multi-dimensional array; and key/value store objects that have native support of a regular KV I/O interface and a multi-level KV interface. Both KV and array objects have versioned data, which allows applications to make disruptive change and rollback to an old version of the dataset. A DAOS object always belongs to a domain that is called a DAOS container. Each container is a private object address space which can be modified by transactions independently of the other containers stored in the same DAOS pool [1] (Fig. 4).

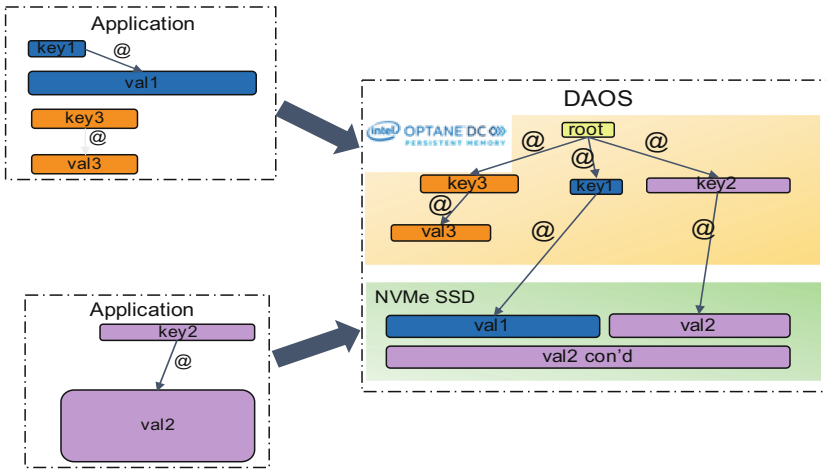


Fig. 4. DAOS data model

DAOS containers will be exposed to applications through several I/O middleware libraries, providing a smooth migration path with minimal (or sometimes no) application changes. Generally, all I/O middleware today runs on top of POSIX and involves serialization of the middleware data model to the POSIX scheme of directories and files (byte arrays). DAOS provides a richer API that provides better and more efficient building blocks for middleware libraries and applications. By treating POSIX as a middleware I/O library that is implemented over DAOS, all libraries that build on top of POSIX are supported. But at the same time, middleware I/O libraries can be ported to work natively over DAOS, bypassing the POSIX serialization step that has several disadvantages that will not be discussed in this document. I/O middleware libraries that have been implemented on top of the DAOS library include POSIX, MPI-I/O, and HDF5. More I/O middleware and frameworks will be ported in the future to directly use the native DAOS storage API.

4.2 DAOS POSIX Support

POSIX is not the foundation of the DAOS storage model. It is built as a library on top of the DAOS backend API, like any other I/O middleware. A POSIX namespace can be encapsulated in a DAOS container and can be mounted by an application into its filesystem tree.

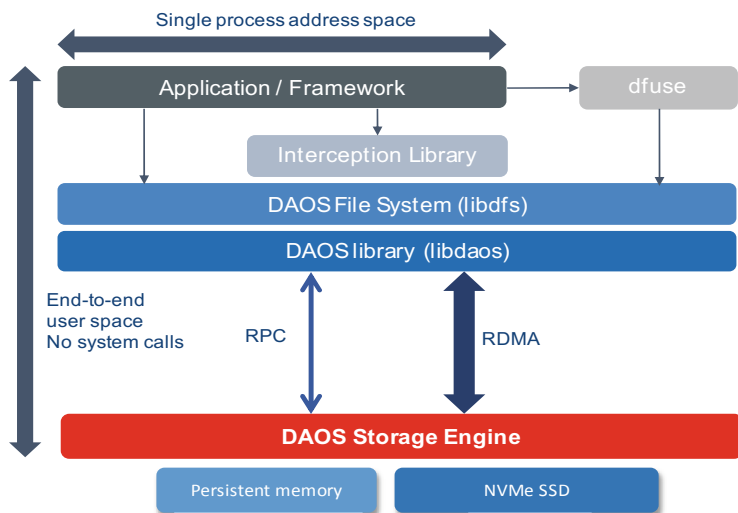


Fig. 5. DAOS POSIX support

Figure 5 shows the software stack of DAOS for POSIX. The POSIX API will be used through a fuse driver using the DAOS Storage Engine API (through `libdaos`) and the DAOS File System API (through `libdfs`). This will inherit the overhead of FUSE in general, including system calls etc. This overhead is acceptable for most file system operations, but I/O operations like read and write can actually incur a significant performance penalty if all of them have to go through system calls. In order to enable OS-bypass for those performance sensitive operations, an interception library has been added to the stack. This library will work in conjunction with `dfuse`, and allows to intercept POSIX `read(2)` and `write(2)` calls in order to issue these I/O operations directly from the application context through `libdaos` (without any application changes).

In Fig. 5, there is a layer between `dfuse`/interception library and `libdaos`, which is called `libdfs`. The `libdfs` layer provides a POSIX like API directly on top of the DAOS API. It provides file and directory abstractions over the native `libdaos` library. In `libdfs`, a POSIX namespace is encapsulated in a container. Both files and directories are mapped to objects within the container. The namespace container can be mounted into the Linux filesystem tree. Both data and metadata of the encapsulated POSIX file system will be fully distributed across all the available storage

of the DAOS pool. The `dfuse` daemon is linked with `libdfs`, and all the calls from FUSE will go through `libdfs` and then `libdaos`, which can access the remote object store exposed by the DAOS servers.

In addition, as mentioned above, `libdfs` can be exposed to end users through several interfaces, including frameworks like SPARK, MPI-IO, and HDF5. Users can directly link applications with `libdfs` when there is a shim layer for it as plugin of I/O middleware. This approach is transparent and requires no change to the application.

5 Performance

The DAOS software stack is still under heavily development. But the performance it can achieve on new storage class memory technologies has already been demonstrated at the ISC19 and SC19 conferences, and first results for the IO500 benchmark suite on DAOS version 0.6 have been recently submitted [16]. IO500 is a community activity to track storage performance and storage technologies of supercomputers, organized by the Virtual Institute for I/O (VI4IO) [17]. The IO500 benchmark suite consists of data and metadata workloads as well as a parallel namespace scanning tool, and calculates a single ranking score for comparison. The workloads include:

- IOR-Easy: Bandwidth for well-formed large sequential I/O patterns
- IOR-Hard: Bandwidth for a strided I/O workload with small unaligned I/O transfers (47001 bytes)
- MDTest-Easy: Metadata operations on 0-byte files, using separate directories for each MPI task
- MDTest-Hard: Metadata operations on small (3901 byte) files in a shared directory
- Find: Finding relevant files through directory traversals

We have adapted the I/O driver used for IOR and MDTEST to work directly over the DFS API described in Sect. 4. The driver was pushed and accepted to the upstream `ior-hpc` repository for reference. Developing a new IO driver is relatively easy since, as mentioned before, the DFS API closely resembles the POSIX API. The following summarizes the steps for implementing a DFS backend for IOR and `mdtest`. The same scheme can also be applied to other applications using the POSIX API:

- Add an initialize callback to connect to the DAOS pool and open the DAOS container that will encapsulate the namespace. A DFS mount is then created over that container.
- Add callbacks for all the required operations, and substitute the POSIX API with the corresponding DFS API. All the POSIX operations used in IOR and `mdtest` have a corresponding DFS API, which makes the mapping easy. For example:
 - change `mkdir()` to `dfs_mkdir()`;
 - change `open64()` to `dfs_open()`;
 - change `write()` to `dfs_write()`;
 - etc.
- Add a finalize callback to unmount the DFS mount and close the pool and container handle.

Two lists of IO500 results are published: The “Full List” or “Ranked List” contains performance results that are achieved on an arbitrary number of client nodes. The “10 Node Challenge” list contains results for exactly 10 client nodes, which provides a standardized basis for comparing those IO500 workloads which scale with the number of client nodes [3]. For both lists, there are no constraints regarding the size of the storage system. Optional data fields may provide information about the number and type of storage devices for data and metadata; when present in the submissions this information can be used to judge the relative efficiency of the storage systems.

For the submission to IO500 at SC19 [16], the IO500 benchmarks have been run on Intel’s DAOS prototype cluster “Wolf”. The eight dual-socket storage nodes of the “Wolf” cluster use Intel Xeon Platinum 8260 processors. Each storage node is equipped with 12 Intel Optane Data Center Persistent Memory Modules (DCPMMs) with a capacity of 512 GiB (3 TiB total per node, configured in app-direct/interleaved mode). The dual-socket client nodes of the “Wolf” cluster use Intel Xeon E5-2699 v4 processors. Both the DAOS storage nodes and the client nodes are equipped with two Intel Omni-Path 100 adapters per node.

Figure 6 shows the IO500 IOR bandwidth of the top four storage systems on the November 2019 edition of the IO500 “10-Node Challenge”. DAOS achieved both the #1 overall rank, as well as the highest “bw” bandwidth score (the geometric mean of the four IOR workloads). Due to its multi-versioned data model, DAOS does not require read-modify-write operations for small or unaligned writes (which generates extra I/O traffic and locking contention in traditional POSIX filesystems). This property of the DAOS storage engine results in very similar DAOS bandwidth for the “hard” and “easy” IOR workloads, and provides predictable performance across many different workloads.

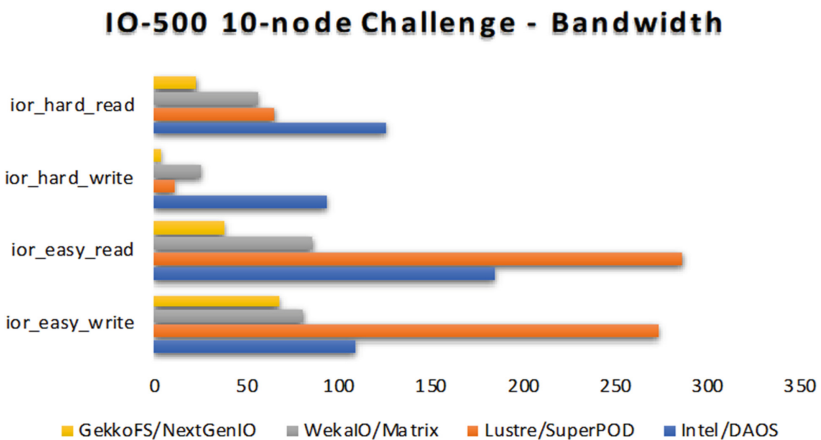


Fig. 6. IO500 10-node challenge – IOR bandwidth in GB/s

Figure 7 shows the mdtest metadata performance of the top four storage systems on the November 2019 edition of the IO500 “10-Node Challenge”. DAOS dominates the overall “md” metadata score (geometric mean of all mdtest workloads), with almost a 3x difference to the nearest contender. This is mainly due to the lightweight end-to-end user space storage stack, combined with an ultra-low latency network and DCPMM storage media. Like the IOR bandwidth results, the DAOS metadata performance is very homogeneous across all the tests, whereas many of the other file systems exhibit large variations between the different metadata workloads.

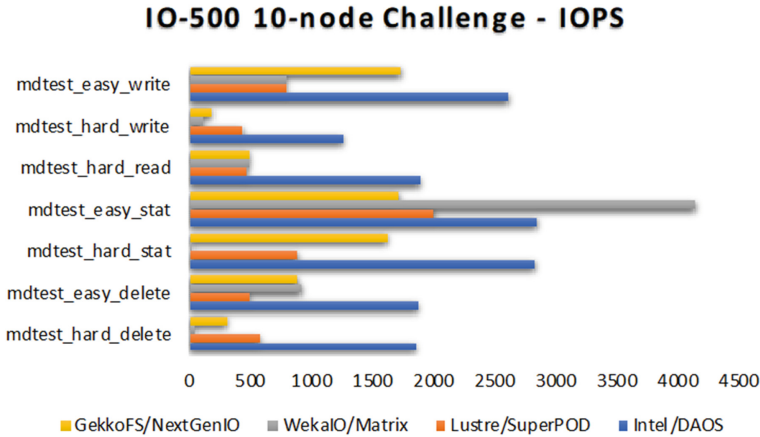


Fig. 7. IO500 10-node challenge – mdtest performance in kIOP/s

DAOS achieved the second rank on the November 2019 “Full List”, using just 26 client nodes. Much better performance can be expected with a larger set of client nodes, especially for those metadata tests that scale with the number of client nodes. So a direct comparison with other storage systems on the “Full List” (some of which were tested with hundreds of client nodes) is not as meaningful as the “10-Node Challenge”.

The full list of IO500 results and a detailed description of the IO500 benchmark suite can be found at Ref. [16].

6 Conclusion

As storage class memory and NVMe storage become more widespread, the software stack overhead factors more and more as part of the overall storage system. It is very difficult for traditional storage systems to take full advantage of these storage hardware devices. This paper presented DAOS as a newly designed software stack for these new

storage technologies, described the technical characteristics of DAOS, and explained how it can achieve both high performance and high resilience.

In the performance section, IO500 benchmark results proved that DAOS can take advantage of the new storage devices and their user space interfaces. More important than the absolute ranking on the IO500 list is the fact that DAOS performance is very homogeneous across the IO500 workflows, whereas other file systems sometimes exhibit orders-of-magnitude performance differences between individual IO500 tests.

This paper only briefly introduced a few core technical components of DAOS and its current POSIX I/O middleware. Other supported I/O libraries like MPI-I/O and HDF5 are not covered by this paper and will be the subject of future studies. Additional I/O middleware plugins based on DAOS/libdfs are still in development. The roadmap, design documents and development status of DAOS can be found on github [5] and the Intel DAOS website [4].

References

1. Breitenfeld, M.S., et al.: DAOS for extreme-scale systems in scientific applications (2017). <https://arxiv.org/pdf/1712.00423.pdf>
2. Rudoff, A.: APIs for persistent memory programming (2018). <https://storageconference.us/2018/Presentations/Rudoff.pdf>
3. Monnier, N., Lofstead, J., Lawson, M., Curry, M.: Profiling platform storage using IO500 and mistral. In: 4th International Parallel Data Systems Workshop, PDSW 2019 (2019). <https://conferences.computer.org/sc19w/2019/pdfs/PDSW2019-6YFSp9XMTx6Zb1FALMAAsH/5PVXONjoBjWD2nQgL1MuB3/6lk0OhJIEPG2bUdbXXPPoq.pdf>
4. DAOS. <https://wiki.hpdd.intel.com/display/DC/DAOS+Community+Home>
5. DAOS github. <https://github.com/daos-stack/daos>
6. Seo, S., et al.: Argobots: a lightweight low-level threading and tasking framework. IEEE Trans. Parallel Distrib. Syst. **29**(3) (2018). <https://doi.org/10.1109/tpds.2017.2766062>
7. SPDK. <https://spdk.io/>
8. Libfabric. <https://ofiwg.github.io/libfabric/>
9. Mercury. <https://mercury-hpc.github.io/documentation/>
10. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: CRUSH: controlled, scalable, decentralized placement of replicated data. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006 (2006). <https://doi.org/10.1109/sc.2006.19>
11. Braam, P.J.: The Lustre storage architecture (2005). <https://arxiv.org/ftp/arxiv/papers/1903/1903.01955.pdf>
12. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the First USENIX Conference on File and Storage Technologies, Monterey, CA, 28–30 January 2002, pp 231–244 (2002). <http://www.usenix.org/publications/library/proceedings/fast02/>
13. Das, A., Gupta, I., Motivala, A.: SWIM: scalable weakly-consistent infection-style process group membership protocol. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, DSN 2002, pp. 303–312 (2002)
14. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm (2014). <https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf>

15. Barton, E.: DAOS: an architecture for extreme storage scale storage (2015). https://www.snia.org/sites/default/files/SDC15_presentations/dist_sys/EricBarton_DAOS_Architecture_Extreme_Scale.pdf
16. IO500 List, November 2019. <https://www.vi4io.org/io500/list/19-11/start>
17. Kunkel, J., et al.: Virtual institute for I/O. <https://www.vi4io.org/start>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

