Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Project Report

Evaluation of DAOS for existing scientific software

submitted by

Tronje Krabbe
Ruben Felgenhauer

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Abstract

This paper will give a brief introduction to the goals of DAOS, the Distributed Asynchronous Object Storage, and the issues with traditional I/O systems that DAOS aims to overcome. The DAOS project is part of a joint effort of the US Department of Energy among multiple US national labs and industry partners like Intel and the HDF group to create the HPC storage stack of the future.

We will give insights into the inner workings of DAOS by presenting DAOS' storage, transactional, and fault models, as well as some miscellaneous information, like its scalability. Furthermore, there will be an excursion into the topic of user space I/O, which is one aspect of DAOS that makes it interesting in the high-performance computing context. As a part of this excursion, there will be a brief look at SPDK, the Storage Performance Development Kit, as well as the Linux kernel drivers that make user space I/O possible.

We will also give an insight into the process of adapting a scientific application to DAOS and compare the alternatives to do so in terms of their practicability.

Contents

1. Introduction

In this chapter, we will describe critical aspects, problems, and technical limitations of traditional storage solutions that are commonly used in current scientific HPC applications, as well as how the developers of the "Fast Forward Storage and IO" project, more specifically the DAOS project, are trying to address these issues.

1.1. Challenges in I/O technology

1.1.1. Current situation

Regarding scientific applications in High Performance Computing, one will almost inevitably encounter highly parallelised applications written for large Super Computers with numbers of cores ranking up to the millions. For their highly performance-oriented nature, performing I/O on these large-scale systems has become an art of its own. For many HPC applications, the available storage space of the individual node is kept small, often featuring fast media like SSD storage, while a centralized storage array running a specialized parallel file system is used to retain the gross of the data.

There is a multitude of parallel file systems that are commonly used like the at least partly POSIX-compliant Lustre, or the non-POSIX-compliant OrangeFS which all offer an (optional) POSIX interface. This comes with a number of problems which are further elaborated in Section 1.1.3 and Section 1.1.4. According to Lofstead et al. [32], these traditional storage solutions are unlikely to meet the requirements for extreme-scale science applications. Therefore, the US department of energy among other stakeholders has started the "Fast Forward Storage and IO" project (FFSIO) to overcome the current issues.

1.1.2. Centralized versus distributed storage

As opposed to the large centralized storage arrays that are widely used in HPC clusters, the traditional storage approach for most Big Data applications is a distributed storage system like the Hadoop Distributed File System (HDFS) which is used with the popular

Apache Hadoop stack. This approach allows to distribute queries to the known locations of the needed data to improve the read performance of tasks following the MapReduce paradigm [41].

However, distributed storage can also have advantages with scientific usage of HPC whenever the compute nodes of the cluster are mostly reading and writing a small and disjoint portion of the entire data set and the data that is needed by all nodes is relatively small. With Big Data and HPC applications closing ranks increasingly, this gains even more importance, and a storage solution that fulfils the requirements for scientific computing will probably need to support both fields with a unified approach.

1.1.3. POSIX interfaces

When talking about POSIX file systems, one has to distinguish between file systems that only offer a POSIX-style interface to the user and file systems that are also POSIX-compliant. A POSIX interface is defined by a set of operations like file creation, deletion, write, or read access. Offering a POSIX interface has many advantages for a file system: Since most clusters and workstations running Linux are running some sort of POSIX file system, applications are easy to develop and port to big cluster computers and developers can assume a certain set of supported operations on the target machine to be supported.

When performing an I/O operation (IOP) in Linux, the query is handled by the Virtual File System (VFS) module of the Linux Kernel. Since typical end-user software is usually running in User Space, this makes a context switch necessary. The VFS tries to schedule operations in a number of ways, like combining smaller operations to neighbouring regions in a storage device to fewer larger ones, and other strategies that are beneficial to the performance on average. Since not all types of operations can be automatically aggregated to reduce the number of context switches, the reality can be a big number which can significantly degrade a system's performance even with today's hardware. With the perspective on future technologies like NVRAM and NVMe SSDs, context switches are ought to become the bottlenecks of storage systems. A comparison of their latencies can be found in Table 1.1.

File systems like OrangeFS are tackling this problem by running entirely in User Space. Other advantages of this model are easier debugging and less disastrous consequences on a system's stability upon a malfunction. However, the optional POSIX interface is realized via FUSE, a Kernel module and library that makes running file systems in User Space possible, with the trade-off that each IOP causes two context switches: One for accessing the VFS module, and one for the Kernel to access the fuse library running in User-Space.

Table 1.1.: Latencies of different storage devices in comparison to a typical context switch latency (order of µs) [20][4][46][45].

Storage device	**≈ Latency**	
L1-Cache	≈ 1 ns	
L2-Cache	≈ 5 ns	
L3-Cache	≈ 10 ns	
RAM	≈ 100 ns	
NVRAM	≈ 1000 ns	← ≈ Context switch latency
SSD (NVMe)	≈ 10 000 ns	
SSD (SATA)	≈ 100 000 ns	
HDD	≈ 10 000 000 ns	

1.1.4. POSIX-compliant file systems

Besides the guarantees that a file system is offering a certain set of operations, the POSIX standard is also making a lot of requirements to a file systems for it to be called POSIX-compliant. For example, "POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data. Note that not all filesystems are POSIX conforming" [48].

Together with features like the access time (`atime`) entry that saves the time when a file has been last accessed into the file's meta data, more specifically into the `inode`, this makes writing parallel file systems very complicated, since the information that a file has been written to or read from would have to be synchronized between all clients that are trying to access the file. For this performance-limiting requirement, a lot of file systems offer the option to restrict or turn off features like access time, or to not guarantee that write accesses will be synchronized between clients instantly or at all, thus making them partly non-POSIX-compliant.

1.1.5. Files and directories

Another challenge naturally arising in scientific computing is properly saving and documenting the results. The traditional semantics with storing files in directories is a very generalized way of achieving this. File systems usually offer a predefined set of metadata that will be partially automatically created and stored alongside the actual user-data like the dates of creation, last modification and last access, or the user-name of the owner.

However, such selected amount of metadata usually doesn't meet the requirements to properly organize and manage scientific data. For example, the original author, contact details, file-revision or project description are unknown to the file system.

Furthermore, the actual structure of the data itself is also not taken into account when data is stored, read or modified which prevents optimization to the I/O patterns from the file system's side. Since it is probable that after the first values of data set have been read, the rest of the data set would follow afterwards, a deeper knowledge of the storage system would allow for more efficient caching strategies.

In scientific computing, container formats like HDF5 or NetCDF are already offering a standardized way of determining detailed information about a data set's metadata and data structure. Since these container formats have a widespread use in the scientific community, the idea to optimize the storage solution with this given information arises naturally.

1.2. Related work

As described in Section 1.1, there exists a multitude of challenges in scientific I/O, especially in the field of high performance computing. Even in a primarily computation-driven (as opposed to data-driven) environments like scientific simulation there are already many different competing strategies to approach these challenges.

On the file system level, the partly POSIX-compliant file system Lustre [5] has become the de-facto standard for supercomputing centres. Lustre runs completely in Kernel space, is usually accessed via its POSIX interface like a local file system and relies on centralised arrays of Meta Data Servers and Object Storage Servers. Another popular file system is OrangeFS which was previously developed as PVFS [7]. While OrangeFS runs completely in user space which makes debugging easier, its POSIX interface is realised through FUSE.

The HDF5 [15] library offers a popular model to store structured data by saving them in a standardizes container format that can be seen as a file system inside a file system which ensures data portability and consistency. The related format NetCDF is built upon HDF5 in version 4 [40] and is especially popular among researchers of earth system models. As an intermediate layer between POSIX interfaces and container libraries or directly from an application perspective, the MPI standard specifies an I/O model which is called MPI-IO [44]. The MPI-IO implementation of the MPI implementation MPICH is called ROMIO [43].

On the other hand, there are several storage systems that do not offer a POSIX interface and abstract from the underlying file structure through an I/O library. Ceph [47] is a distributed object store and file system that offers both an object interface and a

POSIX interface. JULEA [29] is another storage framework that runs completely in user space and offers support for object storage among others. In primarily data-driven environments, HDFS [41] is popular for highly distributed storage of data.

In their 2016 paper [32], Lofstead et al. talk about the design aspects behind a proposed storage stack of the future and introduce the DAOS project. The specific implementation of HDF5 support into DAOS is described by Breitenfeld et al. in their 2017 paper [6]. A comparison of different storage systems including DAOS and Ceph with different HPC applications can be found in a 2018 paper [31] by Liu et al.

1.3. The DAOS project

The "DOE Extreme-Scale Technology Acceleration Fast Forward Storage and IO Stack project", usually abbreviated with "Fast Forward Storage and IO" project (FFSIO) was started by Lawrence Livermore National Laboratory and the US Department of Energy to tackle the problems described in Section 1.1, and is additionally developed by Sandia National Laboratories, Los Alamos National Lab, Oak Ridge National Laboratory, Pacific Northwest National Laboratory, Lawrence Berkeley National Laboratory, Argonne National Laboratory, Intel, and the HDF Group. The FFSIO project describes an architecture that consists of multiple layers which range from applications that perform I/O through their respective I/O libraries to the software stack of the storage system. Part of the latter is the "Distributed Application Object Storage" (DAOS). The project was partitioned into multiple phases and while the first phase which completed in 2014 and focused on the basic functionality originally made a list of requirements to the storage system's hardware like NVRAM and NVMe SSDs, the second phase focused more on fault recovery and other additional features and weakened many of the initial hardware requirements because of limited availability and very high cost of these technologies [32].

The developers of DAOS aim to tackle the problems of traditional storage solutions by working with a distributed storage setup that works with the concept of objects and containers rather than files and directories which they argue will allow for a significant performance increase, while the integration of support for already widely-used container formats like HDF5 promises minimal end-user code changes since this layer is abstracting from the POSIX-interface strongly enough that it's already not strictly needed today. Furthermore, the design-goals of DAOS are "high availability, byte-granular multi-version concurrency control" [32], efficiently realized by a Copy-on-write approach. The problem of context switches described in Section 1.1.3 is tackled by the usage of a user space driver for NVMe SSDs provided by the "Storage Performance Development Kit" (SPDK).

At the time of writing, the current version of DAOS is version 1.0.1 [3] which was released in July 2020 and is still not yet intended for production use. Version 1.0.0 [2] was released in June 2020. In this paper, we are using version 0.9.

1.4. Goals of this paper

In Chapter 2, we will describe in detail the principles and internal structure of DAOS in contrast to a classic POSIX file system like its storage model, transactional model and fault models. Additionally, we will take a brief look at the functionality of the SPDK user space driver. In Chapter 3, we will offer a comprehensive guide on how to perform the necessary steps to modify an existing scientific application for usage together with DAOS. In Chapter 4, we will conclude this paper with a summary and a rundown of what to look out for in future developments, and we will review the practicality of the individual strategies described in Chapter 3 and the claim of the DAOS project that the modification requires minimal code change.

2. DAOS internals

In this chapter, we will introduce DAOS and its core concepts and give an overview over its storage model, transactional model and fault model. Additionally we will briefly explain how I/O from user space can be performed on a Linux system on the example of SPDK.

DAOS is an open-source[1] object store which is primarily designed for Non Volatile Memory [25]. It has a key-value interface, and provides features such as "transactional, non-blocking I/O, advanced data protection with self healing on top of commodity hardware, end-to-end data integrity, fine grained data control and elastic storage to optimize performance and cost." It should be pointed out, that "commodity hardware" primarily refers to NVRAM and NVMe SSD storage, both of which are perhaps not specialty hardware, but still expensive and perhaps not what most would consider to be commodity hardware.

DAOS aims to tackle the problems brought on by more and more data-intensive applications which stretch the limits of existing I/O models. The DAOS developers claim that modern I/O workloads feature "an increasing proportion of metadata", as well as "misaligned and fragmented data", which existing storage solutions are ill-equipped for [21]. The goal is to create a stack that supports massively distributed storage "for which failure will be the norm", while offering low latency and high bandwidth.

DAOS is written primarily in C, and thus offers a native C API, although Python and Go bindings are available as well. A management client is included and written in Go. To give an idea of the scope of the project, DAOS is comprised of about 210,000 lines of C code at the time of writing. It is licensed under the Apache License Version 2.0 [22]. Targeting, among others, the scientific computing context, DAOS offers HDF5 integration via a VOL plugin [18].

The following sections will give insights about the inner workings of DAOS, and the concepts it uses. Specifically, Section 2.1 will cover the storage model, Section 2.2 the transactional model, and Section 2.3 the fault model. Section 2.4 will be an excursion into the topic of user space I/O, which DAOS uses to manage NVMe SSD storage.

[1] `https://github.com/daos-stack/daos`

2.1. Storage Model

2.1.1. Overview

Figure 2.1 gives an example of a DAOS configuration across four storage nodes. The memory on the nodes is divided into a number of Targets, which are the basic unit of storage. They correspond to a fixed-size partition of the node's accessible storage. Those Targets are assigned – uniquely – to different Pools. A Pool is thus a collection of Targets, distributed across different storage nodes. Within those Pools, Containers govern access to the Targets. Each Container is a private address space for Objects, which are not shown in this figure.

Figure 2.2 shows the hierarchy of the DAOS storage model. Pools contain Containers, which in turn contain Objects. Objects can have different types – like byte array or key-value store – and they finally contain the actual data saved within DAOS.

Thus, the storage model can be broken down into these four key concepts:

- Targets
- Pools
- Containers
- Objects

The DAOS documentation gives an overview of the targeted level of scalability, shown in Table 2.1.

The following sections will examine the components of the storage model in greater detail, starting with the Target in Section 2.1.2

Table 2.1.: Targeted Level of Scalability for each DAOS Concept [10].

DAOS Concept	**Order of Magnitude**
System	10^5 servers and 10^2 pools
Server	10^1 targets
Pool	10^2 containers
Container	10^9 objects

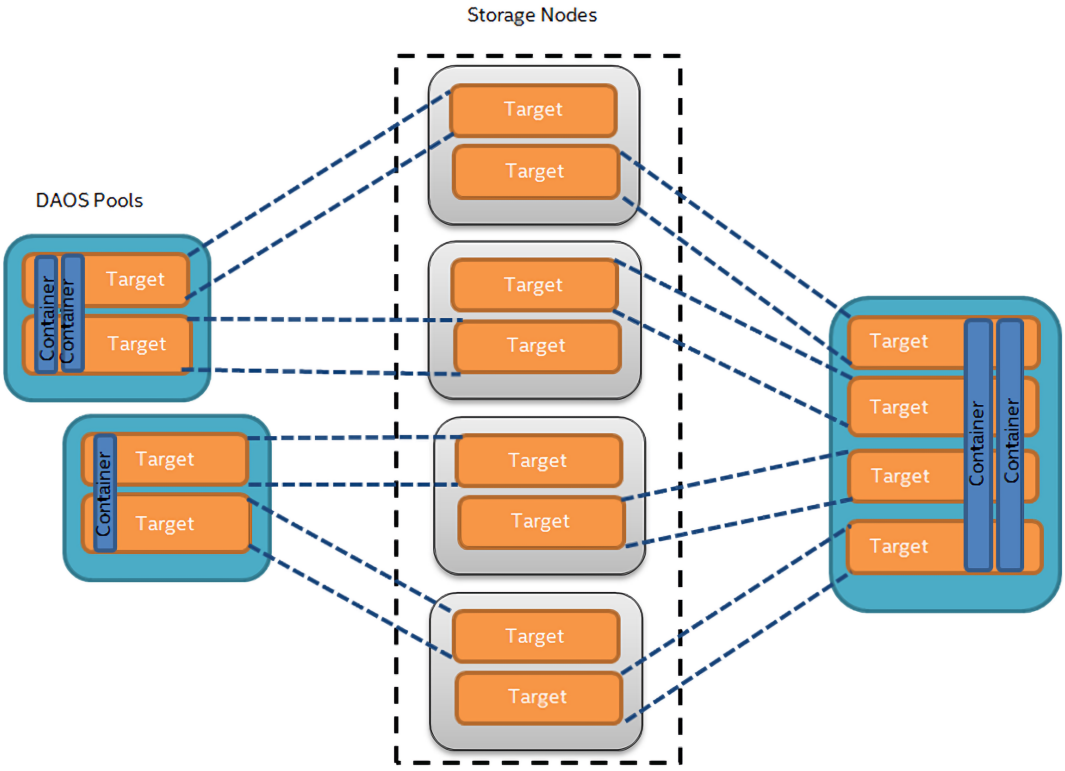

Figure 2.1.: Example of four Storage Nodes, eight DAOS Targets and three DAOS Pools [10].

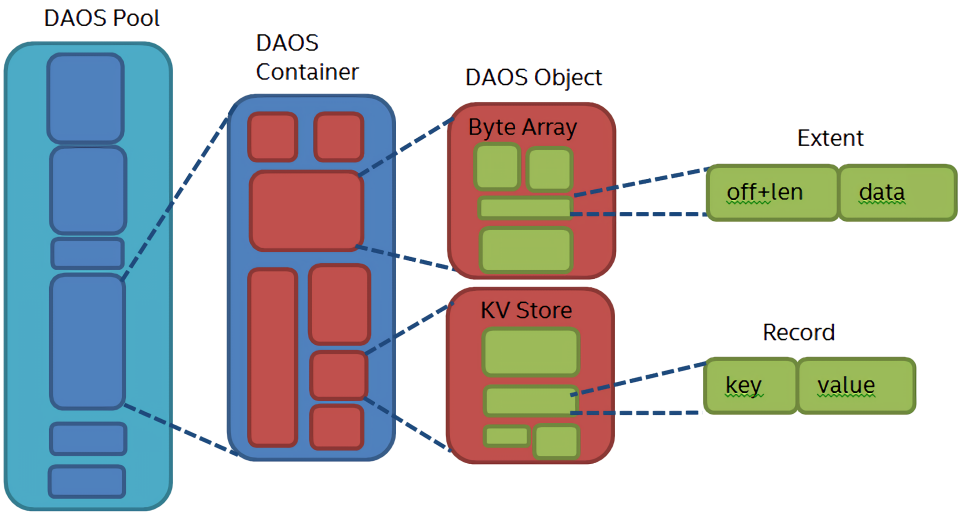

Figure 2.2.: Architecture of DAOS Storage Model [10].

2.1.2. Target

As mentioned previously, the Target is the basic unit of storage within DAOS. It is directly associated with a reservation of memory, as well as – optionally – block-based storage. Regarding memory and block-based storage, DAOS specifically targets NVRAM and NVMe SSDs, respectively.

Since a Target only exists on a single node – as in, it does not govern memory from multiple nodes, but only a single one – it is assumed to have no failover capability when a storage node fails. It is thus assumed to be a single point of failure. It also does not necessarily implement any kind of redundancy internally. It can, however, detect and report corruption via checksums. The higher-order elements in the DAOS storage model are responsible for ensuring redundancy by mirroring data across targets that belong to different nodes.

A Target is further the basic unit of performance and concurrency, since the associated hardware has limited capabilities. Data cannot be written to the Target faster than the underlying memory allows. Targets can report their performance parameters – such as bandwidth and latency – to the upper layers of the storage hierarchy for optimal utilization.

Having now covered the Target, Section 2.1.3 will showcase the first element shown in Figure 2.2, the Pool.

2.1.3. Pool

As shown in Figure 2.1, a Pool is, essentially, a set of Targets spread across different storage nodes. Data and metadata are distributed across these Targets for horizontal scalability, and/or replicated or erasure-coded to ensure durability and availability. Erasure codes are computed using Intel's own Intelligent Storage Acceleration Library[2] [24], which implements Reed-Solomon type erasure codes [39].

As shown in Figure 2.1, each Target is associated with a unique pool, and this membership is "definitive and consistent" [10]. The member-Targets of the Pool are recorded in the so-called Pool Map. This structure also contains the storage topology of the cluster, represented by a tree. This topology is used to identify Targets which share hardware components. The documentation gives the following example of the different levels of such a topology tree, displayed in Table 2.2. This solution represents hierarchical fault domains, and makes it possible to avoid placing redundant data on Targets which are likely to fail together. This and related concepts are further showcased in Section 2.3.

[2] `https://github.com/intel/isa-l`

Table 2.2.: Storage Topology Tree Example [10].

Level	Represented Items
1	Targets which share the same motherboard
2	Motherboards which share the same rack
3	Racks which share the same cage

The storage topology tree must be supplied to DAOS [10], and is managed by the Pool map[11]. Information about the format of the Pool map, or guidelines on how to create it, are currently not given in the DAOS documentation.

When a Target fails, data redundancy inside the Pool is restored online. When Targets are added to a Pool, data is migrated to the new Targets so that storage usage is distributed equally among all Pool members.

Regarding security, Pools are only accessible to authenticated and authorized applications/clients. The authorization scheme uses a simple access control list scheme, derived from NFSv4 [9].

Since a Pool stores a number of different persistent metadata – the Pool Map, a list of containers, authentication and authorization information, etc. – its metadata requires a very high level of robustness. It is therefore replicated on several nodes – in the order of tens – of distinct fault domains. With a limited number of nodes responsible for the metadata, DAOS relies on a Raft-based[3] consensus algorithm to guarantee consistency should a fault occur, as well as to avoid "split-brain" [13][26].

Section 2.1.4 will continue with the Container, the second item in the hierarchy displayed in Figure 2.2.

2.1.4. Container

Getting closer within the storage model to actual data, the Container represents an Object address space inside a Pool. We will cover the DAOS Object in Section 2.1.5.

A Container is "the basic unit of atomicity and versioning" within DAOS. That means that all operations on Objects must be tagged – by the caller – with an identifier – the so-called epoch. The epoch concept dictates that there shall be a so-called Highest Committed Epoch (HCE), with all epochs less than or equal to the HCE being immutable, and all newer ones mutable and unreliable.

[3] `https://raft.github.io/`

Epochs that have been committed to a Container may periodically be aggregated to free space utilized by overlapping writes and to reduce metadata complexity. Epochs can also be snapshot, which means that a permanent reference is placed on a committed epoch to prevent this aggregation. The topic of epochs will be covered more in-depth in Section 2.3.

One more notable detail about Containers is that their metadata is either also managed by the parent-Pool's consensus Raft service, or by a Container-owned Raft instance. This can be chosen on Container creation. A particular use-case or reasons for or against using one approach or the other are not given. We assume that using one or the other has subtle performance and redundancy implications that can be fine-tuned for a complicated DAOS setup.

2.1.5. Object

The final, lowest member of the storage hierarchy (recall Figure 2.2) is the Object. An Object can have one of three types:

- byte array
- key-value store
- document store

Of note is that for key-value store Objects, values cannot be partially updated, and are overwritten on write. However, document store Objects, which are special variations of the key-value stores, allow both atomic values, which have the same behaviour as described previously, as well as byte array values, which do support arbitrary extent read and (over)write. Document stores further implement a locality feature, allowing values to be guaranteed to be collocated on the same Target, should this be desired. All of the above Object types may, in the future, be unified under just the document store.

Objects are kept very simple, with almost no metadata being provided by default. In particular, the system does not maintain time, size, owner, or permission attributes. This is done in order to avoid scaling problems as well as overhead common to traditional storage stacks. Additionally, all operations on Objects are idempotent and can be processed multiple times, always yielding the same results, which guarantees that operations can be repeated until they are either successful or abandoned.

This concludes the segment about the DAOS storage model. Section 2.2 will continue with the transactional model.

2.2. Transactional Model

The stated goal of the transactional model is to provide a high degree of both concurrency and control over the durability of data and metadata. The data set can be updated safely and in-place, and it can also roll back to a known and consistent state upon failure. The key to this is the epoch concept, previously mentioned in Section 2.1.4.

To reiterate, the epoch is a caller-selected identifier, which must be provided by the caller for each I/O operation. As this means that transactions are exported to the top-level API of DAOS, this approach is quite distinct from conventional storage systems [10]. This also means that accessing data in DAOS is not as simple as implicitly opening the newest version of a data object – the epoch identifier must be explicitly stated for *every* operation.

Figure 2.3 shows a simple example of how the state of a container evolves as a series of epochs. Recall that a snapshot is a named, permanent epoch – non-snapshot epochs are aggregated periodically to save space. The epochs up to and including the highest committed epoch denote globally-consistent, immutable Container versions. This guarantees that any consumers can see consistent data, while producers can continually update the state of a Container.

Epochs greater than the HCE correspond to transactions that have not yet been globally committed, where new writes are still taking place. Interestingly, these epochs are guaranteed to be applied in epoch-order, not execution-order, allowing applications not only to model concurrent and distributed transactions, but also distributed execution. Uncommitted epochs offer no guarantees regarding their consistency, but are readable all the same.

DAOS provides no mechanisms to detect or resolve conflicts among different transactions, or within one, making the caller responsible for implementing its own concurrency control strategy. DAOS does, however, provide some functionality for developing conflict detection and resolution implementations. All changes submitted against any epoch – so long as it has not yet been aggregated – can be enumerated. This allows an "optimistic" approach, where conflict detection is delayed until all operations have concluded, without blocking them. Then, all operations can be examined and the transaction aborted, should certain rules or conditions not be satisfied.

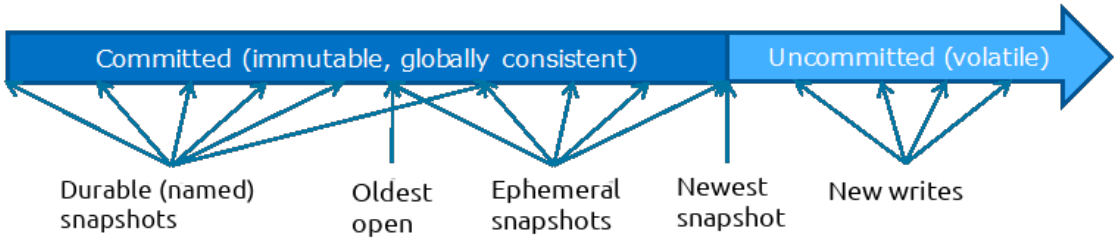

Figure 2.3.: Epoch in a Container [10].

2.3. Fault Model

Recall that DAOS relies of massively distributed storage, and imposes no requirements regarding failover capabilities on the underlying storage. As mentioned in Section 2.1.2, each Target is treated as a single point of failure, and so DAOS achieves reliable redundancy – and thereby availability and durability – by replicating data and metadata across Targets in different fault domains.

2.3.1. Hierarchical Fault Domains

Hierarchical fault domains have been described briefly in Section 2.1.3. DAOS assumes that all fault domains are, in fact, hierarchical, and also do not overlap. The actual hierarchy should be supplied to DAOS by means of "an external database", in an unspecified format [8]. It is worth noting here that a simple one-server, RAM-only DAOS setup, such as the one we used, does not seem to require this, which is, of course, reasonable. It is also possible that the above is a misinterpretation of the documentation – the general layout of a DAOS system is provided in YAML files, and it is possible that the fault domains are computed from the server specifications. DAOS does include example configuration files in `utils/config/examples` (relative to the source code repository root), but includes no examples of a hierarchical fault domain configuration.

Recall also that Pool metadata are replicated on multiple nodes from different high-level fault domains, as also explained in Section 2.1.3. Object data, however, is replicated or erasure-coded across a variable number of fault domains, depending on user requirements [10].

2.3.2. Fault Detection and Diagnosis

DAOS delegates detection of node failure to a Reliability, Availability, and Serviceability (RAS) service, which delivers "authoritative notifications" [10]. This service receives input from a number of sources, among them baseboard management controllers, the network fabric, and the distributed software running on the cluster. If a DAOS client experiences time-outs, it is able to report a problem to this service. The documentation goes on to assert that "all this data is carefully analysed and correlated by the RAS system that then makes authoritative and unilateral decision on node eviction" [10]. Unfortunately, what kind of algorithm or heuristic method is behind the RAS service does not seem to be documented – only that it "can" use a consensus algorithm to assure durability and availability.

The Pool Map is the highest authority – this means that while it listens to the RAS and excludes nodes when the RAS asks it to, it can also decide by itself to exclude nodes

that are not performing satisfactorily, or that the administrator wants removed. It would be interesting to know whether the Pool Map can choose to ignore the RAS and not exclude a node that was deemed failed by the RAS, though this is not explicitly stated. It seems sensible, though, that the RAS has the final say in such a case.

2.3.3. Fault Isolation

Fault isolation is straight-forward – a faulty node must be excluded, and this will be done automatically and quietly so long as the Pool Map has the resources for sufficient redundancy to redistribute the data. Has data been lost, applications will be informed via I/O errors on access, and the administrator will also be notified.

The updated Pool Map is, apparently, pushed eagerly to all Targets. Why Targets must know about the Pool Map is unclear, and the rest of the documentation reads as though Targets do not know about much, other than their own properties. It is likely that, in this case, the documentation means storage nodes rather than targets are provided eagerly with the newest Pool Map.

In any case, client (non-storage) nodes are informed lazily – *not* eagerly – via a simple mechanism: the Pool Map version is included in the RPC protocol, and when a new version becomes available, nodes can fetch the new map. Thus, the exclusion of Targets will eventually have propagated through the whole cluster lazily.

2.3.4. Fault Recovery

Fault recovery is actually performed by all remaining Targets. Each Target creates a list of local Objects that are impacted by the absence of the failed Target(s). This can be achieved because each Target, or its "underlying storage layer", seems to keep a local "object table". Then, for each impacted Object, its shards are redistributed across the remaining, intact Targets, and the recovery is finished. The extent or level of completeness of the object tables that each Target seems to keep is not specified. Intuitively, it would make sense for a higher-level component to be responsible for redistribution of data, such as the containing Pool or some metadata service, but this is not stated explicitly.

In any case, to summarize, the documentation states that the Targets restore redundancy, causing the system to recover from the Target failure. The restoration is executed online, so that applications can still access and update Objects.

2.4. User Space I/O

Normally, writing to and reading from block devices require kernel interaction, so-called context switches. Context switches are generally computationally expensive – in the Linux kernel, it involves switching registers, the stack pointer, and the program pointer [34][17].

So, to speed up performance, one strategy is to manage block devices from user space. The process to achieve this is thus [42]:

- Manually unbind the device driver from the device [28].
- Rebind the driver to one of two special drivers that are part of Linux – `uio`[4] or `vfio`[5].
- At this point, Linux has no control over the device – for example, it will have disappeared from the `devtmpfs` at `/dev`.
- Furthermore, the entire kernel block storage software stack is entirely uninvolved now.
- SPDK provides its own implementation, which is now used to control the device from user space.

The mentioned Linux drivers are beyond the scope of this document. Essentially, they only act as dummy drivers – so that the kernel does not attempt to rebind the device to a fitting driver – and do not provide much additional functionality.

Besides context switches, SPDK also eliminates interrupts, opting to poll devices for completions instead of waiting for interrupts, which is claimed to further improve performance. Apparently, "routing an interrupt to a handler in a user space process just isn't feasible for most hardware designs"[42], and using interrupts would re-introduce the otherwise eliminated context-switches, which are, again, expensive.

[4] `https://www.kernel.org/doc/html/v4.18/driver-api/uio-howto.html`
[5] `https://www.kernel.org/doc/Documentation/vfio.txt`

3. DAOS with a scientific application

In this chapter, we will illustrate the necessary steps to port a specific scientific application that uses a POSIX-style interface to DAOS and evaluate the available alternatives to do so.

For our tests, we used the machine "`abu2`" which is part of the development cluster of the research group "Scientific Computing" (Wissenschaftliches Rechnen, WR) of the Universität Hamburg, which is located at the German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ). `abu2` has 120 GB of RAM and four "AMD Opteron(tm) Processor 6344" CPUs with six cores each and hyperthreading (48 logical cores in total) at 2.9 GHz and runs Ubuntu 18.04. Since this was the only machine available for us and `abu2` does not feature an NVMe SSD, we started DAOS on a single server with only RAM taking the place of NVRAM.

The scientific application we are using is an earth system model that was developed at the Universität Hamburg which is called ECOHAM5 (Ecosystem Model Hamburg, Version 5). ECOHAM simulates the ecology of the North Sea and is used to research the influence of carbon flows in the context of climate change [33][37] and the effects of increasing levels of nutrients [30]. ECOHAM is written in Fortran, parallelised with MPI, and uses NetCDF-4 (Network Common Data Format) for I/O and is therefore a prime example for a typical scientific HPC application. While parallel I/O has been implemented [27], in this paper we are only considering the default version where the master MPI process does all of the (serial) I/O, which makes the utilisation of DAOS easier. However, we did slightly modify ECOHAM for our purposes so that we're able to run it without a batch scheduling system.

By default, ECOHAM's storage stack is structured as follows: The gross of its data is saved in a single NetCDF-4 file. By construction, NetCDF-4 files are also HDF5 (Hierarchical Data Format) files and therefore, the `libnetcdf4` will also internally use the `libhdf5`. Beneath the HDF-5, MPI-IO gets used which finally writes onto a classic POSIX file system.

This offers several different potential contact points to enable ECOHAM to write into DAOS which is also graphically shown in Figure 3.1: The most top-level approach is embracing DAOS with a forked version of NetCDF-4 which has been created by Breitenfeld et al. in 2017 [6][14], however, at the time of writing, a NetCDF-4 version that is compatible with current versions of DAOS could not be found and therefore we won't

consider this option in this paper. Alternatively, a plugin for HDF5 which is referred to as "DAOS-VOL" or "DAOS/HDF5V" can be used [18] that uses DAOS' built-in HDF5 support. The third option is connecting at the MPI-IO layer and using MPICH in a version of at least `v3.4a3` which features a ROMIO ADIO driver for DAOS [36]. Finally, one could also use DAOS' FUSE interface [38] which is referred to as "DFuse" or simply "DFS" and simply configure ECOHAM to write into DFuse's mount point. However, since the usage of FUSE requires two context switches for each I/O operation, as has been already mentioned in Section 1.1.3, this is probably not a good idea in most real-life environments for performance reasons. However, in a testing environment, DFuse can come in handy to quickly examine the data that has been written into DAOS. Additionally, one might also consider a hybrid solution: As described in Section 3.4.1, ECOHAM writes the gross of its data (size-wise) into a single NetCDF-4 file, but creates a lot of metadata in a nested structure of many small files. Since both writing a NetCDF-4 container into a DAOS POSIX container and saving a directory structure in a DAOS HDF5 container is possible but cumbersome, one could simply use DAOS-VOL and FUSE in parallel in situations where write performance for metadata is uncritical.

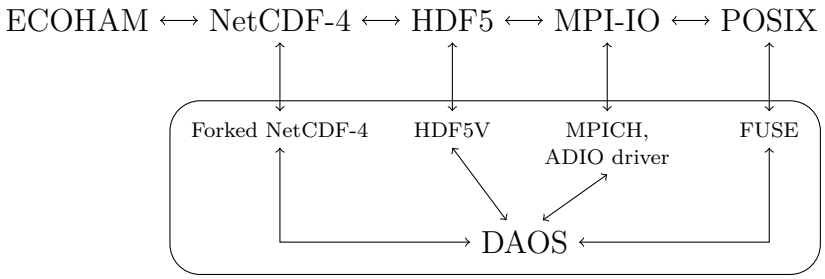

Figure 3.1.: Storage stack of ECOHAM with potential contact points of DAOS using different auxiliary applications or plugins

3.1. Prerequisites

In our setup, we use a common base folder for all applications. This is generally not necessary, but will deal as a way for us to signify that certain applications can or should be built separately and alongside each other. In our case, this path will simply be `$HOME`. We use the variable `$basedir` to guarantee some sort of configurability, which can be set like shown in Listing 3.1.

```
1 $ export basedir="$HOME"
```

Listing 3.1: basedir configuration

Throughout this guide, we will set a number of environment variables which will help us in the later process. Every time we do so, we are expecting the variable to be present

in the later course of this guide. However, most of these variables, especially directory definitions like `$basedir` can be simplified by collecting them in a shell script. An example for such a script has been given in Listing A.2 which can be loaded like shown in Listing 3.2. However, it is advised not to source the script before the installation steps are finished, because some directories that are used will probably not exist up to this point. We will therefore make sure to also include all necessary variable definitions redundantly in their respective steps before Section 3.3. In the following, we assume that a shell is used that supports `bash` syntax and that all commands are executed in the same shell unless otherwise stated.

```
1 $ source load_daos_env.sh
```

Listing 3.2: Automatic environment variable configuration

3.2. Software installation

3.2.1. Installing Spack

For certain aspects of this paper, we use Spack, the supercomputer package manager [16], because its design makes the process of building several different applications in different versions and configurations relatively easy. The process of installing and activating Spack is shown in Listing 3.3.

```
1 $ export spackdir="$basedir"'/spack'
2 $ git clone 'https://github.com/spack/spack.git' "$spackdir"
3 $ source "$spackdir"'/share/spack/setup-env.sh'
```

Listing 3.3: Installing Spack

In the next step we install NetCDF with Fortran support with HDF5 `1.12` as a dependency which is shown in Listing 3.4. This was necessary in our case, because the version of HDF5 that was pre-installed on our system (`1.10`) was too old. The installed packages are a requirement of ECOHAM. We are installing MPICH with an explicitly specified version of `3.3.2` to be able to distinguish it from our custom install as mentioned in Section 3.2.3 which is usually not necessary.

```
1 $ spack install netcdf-fortran%gcc@7.5.0 ^hdf5@1.12.0
    ↪ ^mpich@3.3.2
```

Listing 3.4: Installing NetCDF with HDF5 1.12

3.2.2. Installing DAOS

To build DAOS, we will loosely follow the Administration guide [1], but since we are using the `v0.9` release and the guide is directed towards the `v1.0` release at the time of writing, some things will differ. Alternatively, the document states that a "DAOS RPM packaging is currently available, and DEB packaging is under development and will be available in a future DAOS release. Integration with the Spack package manager is also under consideration." Note that the installation of MPICH as described in Section 3.2.3 would benefit especially if DAOS became available in Spack if one considered also installing MPICH via Spack. However, in the first step, we install DAOS dependencies, clone DAOS' repository, remember its directory in a variable, checkout release `v0.9`, and initialise the submodules. This process is shown in Listing 3.6. Note that the dependency installation steps (line 1–6) are adapted from the Ubuntu 18.04 dockerfile for release `0.9` [12], and that not all steps might be necessary.

```
1  $ sudo apt update
2  $ sudo apt install autoconf bash clang cmake curl doxygen
     ↪ flex gcc git golang-go graphviz libaio-dev
     ↪ libboost-dev libcmocka0 libcmocka-dev libcunit1-dev
     ↪ libevent-dev libibverbs-dev libiscsi-dev libltdl-dev
     ↪ libnuma-dev librdmacm-dev libreadline6-dev libssl-dev
     ↪ libtool-bin libyaml-dev locales make meson nasm
     ↪ ninja-build pandoc patch pylint python-dev python3-dev
     ↪ scons sg3-utils uuid-dev yasm valgrind libhwloc-dev
     ↪ man python-distro software-properties-common
3  $ sudo add-apt-repository ppa:jhli/libsafec
4  $ sudo add-apt-repository ppa:jhli/ipmctl
5  $ sudo apt update
6  $ sudo apt install libsafec-dev libipmctl-dev ndctl ipmctl
7
8  $ export daosdir="$basedir"'/daos'
9  $ git clone 'https://github.com/daos-stack/daos.git'
     ↪ "$daosdir"
10 $ cd "$daosdir"
11 $ git checkout release/0.9
12 $ git submodule init
13 $ git submodule update
```

Listing 3.5: Downloading DAOS v0.9

Next, we'll build DAOS with Scons which is shown in Listing 3.6. We choose to build all dependencies (`--build-deps=yes`), but to use already installed dependencies if available (`USE_INSTALLED=all`). We use 48 cores for compilation (`-j 48`), because our machine features 48 physical cores.

```
1 $ scons -j 48 --config=force --build-deps=yes
    ↪ USE_INSTALLED=all install
```

Listing 3.6: Building DAOS

After this is finished, we need to add DAOS' directory to the `$PATH` and `$CPATH` variables so that we can execute the `daos` binaries from an arbitrary point in the file system. Furthermore, building an application like MPICH with DAOS support requires the `$LD_LIBRARY_PATH` variable to be set accordingly. For `v0.9`, this is shown in Listing 3.7. For `v1.0` and upwards, this may also be done with a provided shell script that can be found inside the DAOS directory[1].

```
1 export CPATH="$daosdir"'/install/include/:'"$CPATH"
2 export PATH="$daosdir"'/install/bin/:\
3 "$daosdir"'/install/sbin/:'"$PATH"
4 export LD_LIBRARY_PATH="$daosdir"'/install/lib/:\
5 "$daosdir"'/install/lib64/:\
6 "$daosdir"'/install/include/:'"$LD_LIBRARY_PATH"
```

Listing 3.7: Setting up DAOS' environment variables

Since we wish to run the `daos_server` binary as a non-root user, we need to employ the `daos_admin` binary which is shown in Listing 3.8. These steps are adapted from the "Pre-deployment Checklist" section of the DAOS Administration Guide [1], where additional information about these commands and other partly optional steps can be found. The process of preparing and starting DAOS can be found in Section 3.3.

[1] at "`utils/sl/utils/setup_local.sh`"

```
1 $ chmod -x "$daosdir"'/install/bin/daos_admin'
2 $ sudo cp "$daosdir"'/install/bin/daos_admin'
    ↪ '/usr/bin/daos_admin'
3 $ sudo chmod 4755 '/usr/bin/daos_admin'
4 $ sudo mkdir -p '/usr/share/daos/control'
5 $ sudo ln -sf
    ↪ "$daosdir"'/install/share/daos/control/setup_spdk.sh'
    ↪ '/usr/share/daos/control'
6 $ sudo mkdir -p '/usr/share/spdk/scripts'
7 $ sudo ln -sf
    ↪ "$daosdir"'/install/share/spdk/scripts/setup.sh'
    ↪ '/usr/share/spdk/scripts'
8 $ sudo ln -sf
    ↪ "$daosdir"'/install/share/spdk/scripts/common.sh'
    ↪ '/usr/share/spdk/scripts'
9 $ sudo ln -s "$daosdir"'/install/include
    ↪ /usr/share/spdk/include'
```

Listing 3.8: Setting up DAOS' elevated privileges

3.2.3. Installing MPICH

There are multiple options to install MPICH with DAOS support. Since our installation of MPICH occurred before the version `v3.4a3` was released, where the code changes for the ROMIO ADIO driver were merged, Listing 3.9 shows the process which uses a version of MPICH which was forked from the DAOS developers which was adapted from [35] and [19]. However, adapting this to the mainline MPICH should be as simple as cloning the official MPICH repository[2] and staying on the `master` branch. An alternative to this approach is installing MPICH via Spack and ensuring that the version is at least `v3.4a3` and that the necessary parameters of the `configure` statement below are given. However, at the time of writing, the default version of MPICH in Spack is `v3.3.2`.

[2] `https://github.com/pmodels/mpich`

```
$ export mpichdir="$basedir"'/mpich'
$ git clone 'https://github.com/daos-stack/mpich'
  ↪ "$mpichdir"
$ cd "$mpichdir"
$ git checkout daos_adio
$ git submodule init
$ git submodule update
$ export MPI_LIB=''
$ ./autogen.sh
$ mkdir 'build'
$ cd build
$ export F90=''
$ export F90FLAGS=''
$ ../configure --prefix="$mpichdir"/install
  ↪ --enable-fortran=all --enable-romio --enable-cxx
  ↪ --enable-g=all --enable-debuginfo
  ↪ --with-device=ch3:sock --with-file-system=ufs+daos
  ↪ --with-daos="$daosdir"'/install'
  ↪ --with-cart="$daosdir"'/install'
$ make -j48
$ make install
```

Listing 3.9: Installing MPICH with DAOS support

Afterwards, we create a new external package for our Spack installation by writing the content of Listing 3.10 into the file "`$HOME/.spack/packages.yaml`".

```
packages:
  mpich:
    paths:
      mpich@3.4a2: /daos-user/mpich/install
    buildable: False
```

Listing 3.10: `packages.yaml` for external MPICH Spack package

This enables us to install `netcdf-fortran` exactly as in Section 3.2.1, but with the new external package as a dependency. This is demonstrated in Listing 3.11.

```
$ spack install netcdf-fortran%gcc@7.5.0 ^hdf5@1.12.0
  ↪ ^mpich@3.4a2
```

Listing 3.11: Installing NetCDF with a custom MPICH

The process of using ECOHAM with this MPICH installation is described in Section 3.4.4.

3.2.4. Installing DAOS-VOL

With an existing installation of DAOS and HDF5, we can build DAOS-VOL as described in Listing 3.12. Note that we use the HDF5 that we installed in Spack as a dependency to `netcdf-fortran` in Section 3.2.1. This gets picked up by `cmake` automatically.

```
$ export daosvoldir="$basedir"'/daos-vol'
$ git clone \
'https://bitbucket.hdfgroup.org/scm/hdf5vol/daos-vol.git' \
"$daosvoldir"
$ cd "$daosvoldir"
$ mkdir build
$ cd build
$ spack load mpich
$ spack load hdf5

$ cmake \
-D CMAKE_INSTALL_PREFIX="$daosvoldir"'/install-custom' \
-D DAOS_LIBRARY="$daosdir"'/install/lib64/libdaos.so' \
-D DAOS_COMMON_LIBRARY="$daosdir"'/install/lib64'\
'/libdaos_common.so' \
-D DAOS_INCLUDE_DIR="$daosdir"'/install/include' \
-D CART_LIBRARY="$daosdir"'/install/lib/libcart.so' \
-D CART_INCLUDE_DIR="$daosdir"'/install/include/cart/' \
".."
$ make
$ make install
```

Listing 3.12: Installing DAOS-VOL

3.3. Using DAOS

This section will show how to configure and start the DAOS server for testing purposes and further subsequent steps. We use an example configuration for a single server setup which uses sockets which we copy from inside `$daospath` to `$basepath` (see Listing 3.13) which should contain two sections at the bottom (see Listing 3.14) that configure a RAM disk ("`scm`") and an NVMe device ("`bdev`"). Since we'll only use RAM, we'll comment out the last three lines and also increase the size of the RAM disk to 50 GB. The bottom file should then look like Listing 3.15.

```
1 $ cd "$daosdir"'/utils/config/examples/'
2 $ cp 'daos_server_local.yml' "$basedir"/
```

Listing 3.13: Acquiring example DAOS config

```
1 scm_mount: /mnt/daos  # map to -s /mnt/daos
2 scm_class: ram
3 scm_size: 4
4
5 bdev_class: file
6 bdev_size: 16
7 bdev_list: [/tmp/daos-bdev]
```

Listing 3.14: Original DAOS config

```
1 scm_mount: /mnt/daos  # map to -s /mnt/daos
2 scm_class: ram
3 scm_size: 50
4
5 # bdev_class: file
6 # bdev_size: 16
7 # bdev_list: [/tmp/daos-bdev]
```

Listing 3.15: Modified DAOS config

We can now start DAOS via the `daos_server` binary as shown in Listing 3.16.

```
1 $ daos_server --debug
    ↪ --config="$basepath"'/daos_server_local.yml' start
```

Listing 3.16: Starting DAOS

Since the call to `daos_server` does not return until DAOS terminates, we will open a second shell where we execute the commands that are given in Listing 3.17: We format the attached storage and create a new pool with 20 GB size. Note that we will need to write about 17.7 GB as will be described in Section 3.4.1. Afterwards, we'll save the pool information which is characterized by a UUID and the number of pool service replicas (which will usually be 0) in the variables `$DAOS_POOL` and `$DAOS_SVCL`. These variables will also be used by the DAOS-VOL plugin (see Section 3.4.3) and MPICH ADIO driver (see Section 3.4.4) to determine the pool to write into. In the following, we assume that we ever only create one pool. Also note that it is now safe to use the `load_daos_env.sh` shell script that has been mentioned in Section 3.1 since all required directories should now be present.

```
1  source "$basepath"'/load_daos_env.sh'
2
3  $ dmg -i network list
4  localhost:10001: connected
5  Supported Providers:
6  localhost:10001:
7   ofi+gni, ofi+psm2, ofi+tcp, ofi+sockets, ofi+verbs, ofi_rxm
8
9  $ dmg --insecure --host-list=localhost:10001 storage format
     ↪ --reformat
10 $ dmg --insecure --host-list=localhost:10001 pool create
     ↪ --scm-size=20G
11
12 $ POOL_INFO="$(dmg --insecure --host-list=localhost:10001
     ↪ pool list)"
13
14 $ echo "$POOL_INFO"
15 localhost:10001: connected
16 Pool UUID                                 Svc Replicas
17 ---------                                 ------------
18 2b5a7f8a-ad58-4cb5-a1cd-397da2792966 0
19
20 $ export DAOS_POOL="$(echo "$POOL_INFO" | awk 'NR==4{print
     ↪ $1}')"
21 $ export DAOS_SVCL="$(echo "$POOL_INFO" | awk 'NR==4{print
     ↪ $2}')"
22
23 $ echo "DAOS_POOL=\"$DAOS_POOL\"; DAOS_SVCL=\"$DAOS_SVCL\""
24 DAOS_POOL="2b5a7f8a-ad58-4cb5-a1cd-397da2792966";
     ↪ DAOS_SVCL="0"
```

Listing 3.17: DAOS storage initialisation and pool creation

Below pool level, it is also necessary to create a container to write into. Containers can have different types like `POSIX` or `HDF5` which are specialized for the kind of data that they are meant to accept. While `DAOS-VOL` is automatically creating a container with type `HDF5` upon write access, it is necessary for both MPICH and DFuse to create a container of type `POSIX` before usage. This is demonstrated in Listing 3.18. The process for a `HDF5` container is analogous.

```
1 $ daos cont create --pool="$DAOS_POOL" --svc="$DAOS_SVCL"
    ↪ --type=POSIX
2 $ export DAOS_CONT="$(daos pool list-cont
    ↪ --pool="$DAOS_POOL" --svc="$DAOS_SVCL")"
```

Listing 3.18: Creating a container with type `POSIX`

Finally, for all applications that communicate with DAOS, it is necessary that a DAOS agent is running on the same machine as the client application. In our case, the server and client machines are identical and we start the `daos_agent` binary with the same configuration file as the `daos_server` which is shown in Listing 3.19. Just as with `daos_server`, this will not return immediately, so in the following, we'll start a third shell for all subsequent tasks.

```
1 $ daos_agent --insecure
    ↪ --config-path="$basepath"'/daos_server_local.yml'
```

Listing 3.19: Starting the DAOS agent

3.4. Adapting ECOHAM to DAOS

In this section we will install ECOHAM and embrace different options to make it write into DAOS. Since neither the original ECOHAM which is maintained by Pätsch and Kühn [37] nor the forked version from the research group "Scientific Computing" of the Universität Hamburg are publicly available online, this mainly deals as an example for other, similar applications. In our tests, we use our own branch that is based on the work of Kostede [27]. By default, ECOHAM will expect to run using a job scheduler like `slurm` and to write into a `lustre` file system. It may also be desirable to run ecoham with an I/O profiler like `scorep`. However, in our example, we need neither of those. In Listing 3.20 we clone the repository and checkout our branch which enables ECOHAM to run without a batch scheduling system, sets the output directory to "`$HOME/ECOHAM_Output`", replaces the `scorep` compiler wrapper with `mpif90`, enables us to specify the number of used MPI processes as a command line argument, and replaces the Spack installation with the one created in Section 3.2.1.

```
$ source "$basedir"'/load_daos_env.sh'
$ git clone 'ecoham.git' "$ecohamdir"
$ cd "$ecohamdir"
$ git checkout daos_evaluation
```

Listing 3.20: Cloning ECOHAM

3.4.1. General usage

Now, to run ECOHAM locally without DAOS, we can use the commands that are shown in Listing 3.21. Originally, ECOHAM takes two parameters: The first argument is the run id which is a generic identifier that can be used to distinguish the output of multiple runs from each other in the context of a batch scheduling system. For our local runs, this is not of special interest for us, so in the following, we always assume that this parameter will be "`0`". The second argument is the run type which can be either "`0`" (compilation only), "`1`" (`0` plus preparation of the job submission) or "`2`" (`1` plus submission of the job). As the compilation is relatively quick, we will always choose "`2`" – compilation, preparation, and submission of the run (which in our case equals starting ECOHAM locally) – so that we can purge the output directory between each two subsequent runs to avoid reuse of any cached output data ("warmstart data") for the best comparability. The third parameter gets added by our patch that was mentioned in Section 3.4 and determines the number of MPI processes that ECOHAM gets started with. In this case, it gets started with 24 processes because this achieved the shortest runtime in our tests (see Section 3.4.2 for a typical value). ECOHAM tends to be delicate in terms of processes that it gets started with: We determined that values from 1 to 27 were safe to run ECOHAM with and that it didn't start with all tested values above 27. ECOHAM's RAM consumption and runtime vary between 3 GB and 19 hours (with one process) and 9 GB and 9 minutes (with 27 processes). Note that these benchmarks were done with a low number of repetitions and that the average value may differ.

```
$ cd "$ecohamdir"'/serielle_io'
$ export ECOHAM_RUNID='0'
$ export ECOHAM_RUN_TYPE='2'
$ export ECOHAM_NUM_PROCS='24'
$ ./InitAndCompile.sh "$ECOHAM_RUNID" "$ECOHAM_RUN_TYPE"
  ↪ "$ECOHAM_NUM_PROCS"
```

Listing 3.21: Running ECOHAM

The shown call will create many files in "`$HOME/ECOHAM_Output/ECOHAM.0`". Before computation, `InitAndCompile.sh` will actually copy ECOHAM's source code to this folder as well. The main space consumer is a NetCDF-4 / HDF5 file located at "`ECOHAM_Output/ECOHAM.0/res.0/0.1977.00/0_3D.nc`" which will be about 17.7 GB. The complete folder will be about 18 GB.

3.4.2. Usage with DFuse

The most straightforward way to utilise any application to write to DAOS is the "DAOS File System" (DFS), otherwise known as "DFuse". As the name suggests, this service provides a POSIX file system interface using FUSE. However, it is presumable that this will not offer a very good performance since the required number of context switches for I/O calls gets increased to 2 (compare POSIX file system: 1 context switch; DAOS goal: 0 context switches), so DFuse defeats one of DAOS' main purposes and should therefore primarily be used in testing environments – a performance evaluation can be found at the end of this section. Nonetheless, it is a very easy way to check if DAOS accepts data and persists it correctly and might be a viable option for small amounts of data with performance-uncritical applications. This can also be applicable in a hybrid solution where DAOS-VOL is used to save NetCDF-4 data, but a nested directory structure containing metadata should be saved in a DAOS `POSIX` container which is described in Section 3.4.5.

DFuse can be started through the commands given in Listing 3.22: We redirect ECOHAM's output simply by symlinking the output directory to the designated DFuse mount point. Finally, we start the `dfuse` binary with the specified pool and container. Afterwards we can simply start ECOHAM like described in Section 3.4.1 in a new shell, since `dfuse` was started with the `--foreground` option and will therefore only return on termination.

```
1 $ rm -rf "$ecohamoutputdir"
2 $ ln -s "$dfsmntdir" "$ecohamoutputdir"
3 $ dfuse -S --mountpoint="$dfsmntdir" --svc="$DAOS_SVCL"
    ↪ --pool="$DAOS_POOL" --container="$DAOS_CONT"
    ↪ --foreground
```

Listing 3.22: Starting DFuse

After ECOHAM has run through, we can stop DFuse via `fusermount` which is shown in Listing 3.23. Afterwards we'll make sure to unlink the directories again and restore the original output directory.

It's worth mentioning that ECOHAM will try to gather information about multiple symlinks that it creates inside the directory "`$HOME/ECOHAM_Output/ECOHAM.0/wrk.0/Input`" which DFuse apparently does not support, because for each file, "`ls: Input/<filename>: Operation not permitted`" will be printed, where "`<filename>`" is a placeholder. Additionally, roughly every second run of ECOHAM will terminate with a segmentation fault prematurely. Complications like these should be a serious consideration if one attempted a hybrid solution like described in Section 3.4.5.

```
1 $ fusermount3 -u "$dfsmntdir"
2 $ rm "$ecohamoutputdir"
3 $ mkdir -p "$ecohamoutputdir"
```

Listing 3.23: Stopping DFuse

Another point to mention is that besides `dfuse`, there also exists a `dfuse_hl` binary that uses the high-level FUSE API and is at its current point a legacy implementation of the DAOS file system that may be removed in the future. The usage is shown in Listing 3.24 and one can simply put this line in place of line 5 of Listing 3.22.

```
1 $ dfuse_hl "$dfsmntdir" -s -f -d -p "$DAOS_POOL" -l
   ↪ "$DAOS_SVCL" -c "$DAOS_CONT"
```

Listing 3.24: Starting DFuse via `dfuse_hl`

While ECOHAM does not show any error messages about unpermitted operations in a successful run with `dfuse_hl` which could imply that the required operations are supported by it. However, a closing statement about the correctness of the output in both cases would require plausibility tests for all files which we did not conduct, and the problems with frequent segmentation faults does persist here as well. As one can find in Table 3.1, there is no difference between the performance of `dfuse` and `dfuse_hl` beyond their standard deviation in our application: "`normal`" denotes a regular run of ECOHAM using a POSIX file system, "`dfuse`" denotes a run using the regular DFuse binary, "`dfuse_hl`" denotes a run using the high level DFuse binary. Listed is the mean runtime of the different variants.

Note that this shows the complete runtime of ECOHAM including compilation and is not a proper I/O benchmark. The deltas between the runtimes can not simply be transferred to an assumption that the I/O takes the respective amount longer or shorter time. Furthermore, this isn't a proper benchmark for DAOS either: Our single-server setup is far from realistic and the results can not be compared to the potential results in a production environment.

Table 3.1.: Performance evaluation of different runs of ECOHAM on a POSIX file system (normal) and using the DAOS FUSE file system.

Variant	#Measurements		Mean runtime[2]		Std. dev.[2]
	Total	Valid	[s]	[m:s]	[s]
normal	10	10	535.7	8:56	19.8
dfuse	10	5	690.1	11:30	9.1
dfuse_hl	10	4	688.1	11:28	32.1

3.4.3. Usage with DAOS-VOL

Running ECOHAM with DAOS-VOL should only require setting two environment variables as shown in Listing 3.25. The plugin will then be automatically picked up by the HDF5 installation that we provisioned inside the Spack directory when we start ECOHAM like described in Section 3.4.1. The pool information will be read from the variables `$DAOS_POOL` and `$DAOS_SVCL` which we set in Section 3.3.

```
1 $ export HDF5_VOL_CONNECTOR=daos
2 $ export HDF5_PLUGIN_PATH="$daosvoldir"'/install-custom/lib'
```

Listing 3.25: Setting up DAOS-VOL

This will automatically create a new container of type `HDF5` inside the specified DAOS Pool. However note, that only the content of ECOHAM's NetCDF-4 file (`0_3D.nc`) will be saved into this container; the rest of directory structure is written into the POSIX file system as usual. Therefore, this method of applying DAOS-VOL is primarily attractive for applications that write all of their data into a single HDF5 file and does not really provide a desirable behaviour for our case. However, in Section 3.4.5, we suggest a way to use both DAOS-VOL and DFuse to tackle this problem.

[2]Only valid runs are considered.

This approach also is not currently in a working state. ECOHAM does attempt to write to DAOS using the DAOS-VOL connector, but fails with the message "`NetCDF: HDF error`". The DAOS server log, located at "`/tmp/daos_server.log`", contains the following message multiple times: "`Could not prepare akey iterator DER_NONEXIST(-1005)`". We were able to trace this error to "`src/common/btree.c`". It is generated in line 1742 of this file, in a function called "`dbtree_fetch`". We were unable to find out what exactly causes the error. DAOS and DAOS-VOL as such work fine, as demonstrated in Section 3.5, so this is likely somehow caused by specific behavior within ECOHAM or NetCDF code.

3.4.4. Usage with MPICH ADIO driver

Alternatively, one could also connect to DAOS on the MPI-IO level. This can be achieved by swapping the `spack load` directives in ECOHAM from the packages installed in Section 3.2.1 to the packages installed in Section 3.2.3. Additionally, we have to prepend "`daos:`" to the paths of all files that are written with MPI-IO. The necessary patch can be found in Listing A.1. Now, if the environment variables `$DAOS_POOL`, `$DAOS_SVCL` and `$DAOS_CONT` are set, ECOHAM should use the forked MPICH and write into DAOS.

Note that this would simply write ECOHAM's NetCDF-4 file into a container of type `POSIX`. Again, all remaining files of the directory structure that are not written by MPI-IO are still written into the POSIX file system. Additionally, by writing it into a POSIX container, it is also significantly harder to access the written data as NetCDF-4 data from outside of DAOS, since one can not simply directly access it via the HDF5 API, e.g. if one wished to access only a certain record of the file or to use applications like `h5dump` to get a glimpse of the file's structure. Instead, one has to perform additional steps like mounting the POSIX container or opening the container with MPI-IO. Therefore, using the MPICH DAOS driver is not a satisfying solution for usage with ECOHAM either. However, since a container of type `POSIX` is used, it is therefore trivial to incorporate DFuse to read the file from the container. This might prove useful if one wanted to achieve a high writing performance but reading the file for evaluation was not time-critical.

As with DAOS-VOL, this approach did also not work as intended. While ECOHAM runs through without any errors and the `0_3D.nc` is missing from the file system suggesting that either an error occurred or that ROMIO picked it up correctly, it is apparently not written into any DAOS container either. It remains to find out if this problem persists with an updated MPICH installation via Spack.

3.4.5. Usage with DAOS-VOL and FUSE

To approach the problem of DFuse's low write speed and that DAOS-VOL will only write the NetCDF-4 data into DAOS, an obvious solution would be a combination of the two: One could benefit from DAOS-VOL's supposed high write speed and write all the remaining data into DFuse's mount point.

The adaptation should be trivial: One could create a new container of type `POSIX`, mount it using DFuse and symlink ECOHAM's output directory to the mount point as described in Section 3.4.2, and afterwards set the required environment variables as described in Section 3.4.3 before starting ECOHAM. A remaining disadvantage of this method is of course that the NetCDF-4 file will still miss inside the POSIX context, so in scenarios where this is critical, e.g. for immediate post-run analysis with external tools, this might be a problem. Additionally, the lower performance of DFuse could drag our whole application back again.

3.5. Proof-of-concept DAOS-VOL Test Application

To confirm DAOS-VOL as working, we have written a simple test application[3], which includes two target binaries. One writes data into a HDF5 file and then exits, and the other is intended to be run afterwards, and will read the data back and confirm it is, in fact, the data that the first binary has written.

With DAOS running, and when linked with the same HDF5 library as DAOS-VOL, this works fine, as long as the required environment variables are set, as described in Section 3.4.3. An abridged excerpt of the source code is included in Listing A.3. After running the writing portion of the test application, the HDF5 utility `h5dump` also finds the written HDF5 file – which does not exist in the file system, but actually resides within DAOS – and can dump its header and contents. We can conclude that the issues with getting ECOHAM to work with DAOS-VOL likely occur due to some kind of issue with either ECOHAM itself, or NetCDF. In principal, writing data to DAOS using DAOS-VOL seems to work fine, as demonstrated by the test application.

[3] `https://gist.github.com/tronje/5cf6650896f2cb828630b8abb0c29b37`

4. Conclusion and outlook

This report presented some of the inner workings of DAOS. The reader should have a general idea of how data is treated and stored in it, and of the concepts that make DAOS special and, arguably, a promising and future-proof storage system. Many aspects of DAOS were not explored. Some due to thin or difficult to find documentation, and others merely due to the limited scope of this document. A more in-depth look at the Pool Map, and Pool metadata in general would have been possible, as well as the concept of Object Schemata and Classes [23].

The security model which DAOS employed was also largely ignored, as it simply is not very fleshed out yet, and not very interesting or novel in general. It is also, apparently, very much subject to change, as the documentation notes several times that security aspects may change or be improved later.

How well DAOS performs in real-world scenarios remains to be seen. Before DAOS can actually be tested, much work will likely have to be put into the documentation of the project, as similarly stated previously, in Chapter 2.

Besides DAOS, this report briefly showcased how it is possible to perform I/O operations on block devices from user space on a Linux system – one of many novel solutions that DAOS employs – by using special Linux drivers and SPDK. This approach, while complicated, gives considerably I/O performance advantages.

Additionally, we provided a comprehensive guide about how to use DAOS with the scientific application ECOHAM. Judging from the several pitfalls of the individual strategies, we can conclude that the question concerning how much effort is required to adapt an existing scientific application to DAOS is not very easily answered.

On the one hand, the required code changes to ECOHAM were indeed very small. In fact, the variant with MPI-IO was the only one that required any code change at all. On the other hand, this comes with a few downsides: Firstly, the available alternatives and required measures are not very easy to find and to our knowledge, this paper is the first that offers a comparison and guide for this matter, although it was only described for a non-publicly available application, but due to DAOS' rapidly changing nature it is supposable that much of the information presented here will become obsolete rather quickly anyway. Secondly, none of the alternatives proved itself to be a perfect match for ECOHAM's output procedure: The DFuse interface doesn't offer a satisfying

performance, DAOS-VOL doesn't consider the non-NetCDF-4 data, MPI-IO does not do this either and additionally writes NetCDF-4 data into a POSIX container, and with a hybrid approach, we'll end up with data that is split into two containers while the performance issues of DFuse persist.

Therefore, even in in our toy example, a satisfying way to use ECOHAM with DAOS does not exist at the time of writing. It may be true that DAOS might already be a good fit for applications that perform all of their output via HDF5 or MPI-IO, but it is easy to assume that there are many applications that follow a similar way as ECOHAM with mixed output, where an adaptation to DAOS is simply not sensible without major code changes.

We can therefore conclude that at its current state of development, DAOS' fitness in the context of scientific applications varies dramatically from case to case and has to be carefully evaluated for each situation. It remains for future work to do performance and correctness evaluations of the demonstrated alternatives and to set up DAOS in a more natural server environment.

Bibliography

[1] *Administration Guide – DAOS v1.0*. URL: https://daos-stack.github.io/admin/installation/ (visited on 2020-08-14).

[2] *Announcement: DAOS 1.0 is generally available!* URL: https://daos.groups.io/g/daos/message/1052 (visited on 2020-08-17).

[3] *Announcement: DAOS 1.0.1 and pre-built RPMs are available*. URL: https://daos.groups.io/g/daos/message/1107 (visited on 2020-08-17).

[4] Jonas Bonér. *Latency Numbers Every Programmer Should Know*. URL: https://gist.github.com/jboner/2841832 (visited on 2020-08-17).

[5] Peter Braam. "The Lustre Storage Architecture". In: *CoRR* abs/1903.01955 (2019). arXiv: 1903.01955. URL: http://arxiv.org/abs/1903.01955.

[6] M. Scot Breitenfeld et al. "DAOS for Extreme-scale Systems in Scientific Applications". In: *CoRR* abs/1712.00423 (2017). arXiv: 1712.00423. URL: http://arxiv.org/abs/1712.00423.

[7] Philip Carns et al. "PVFS: A parallel file system for Linux clusters". In: *ALS* 4 (2000-11).

[8] *DAOS Fault Model*. URL: https://daos-stack.github.io/overview/fault/ (visited on 2020-08-22).

[9] *DAOS Security Model*. URL: https://daos-stack.github.io/overview/security/ (visited on 2020-08-22).

[10] *DAOS Storage Model*. URL: https://github.com/daos-stack/daos/blob/master/doc/storage_model.md (visited on 2019-07-01).

[11] *DAOS Storage Model*. URL: https://daos-stack.github.io/overview/storage/ (visited on 2020-08-17).

[12] *daos/Dockerfile.ubuntu.18.04 at release/0.9*. URL: https://github.com/daos-stack/daos/blob/release/0.9/utils/docker/Dockerfile.ubuntu.18.04 (visited on 2020-08-14).

[13] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. "Consistency in a Partitioned Network: A Survey". In: *ACM Comput. Surv.* 17.3 (1985-09), pp. 341–370. ISSN: 0360-0300. DOI: 10.1145/5505.5508. URL: https://doi.org/10.1145/5505.5508.

[14] *Exascale FastForward / Unidata NetCDF*. URL: https://bitbucket.hdfgroup.org/projects/FFWD2/repos/netcdf-c/browse (visited on 2020-08-27).

[15] Mike Folk et al. “An overview of the HDF5 technology suite and its applications”. In: 2011-03, pp. 36–47. DOI: 10.1145/1966895.1966900.

[16] T. Gamblin et al. “The Spack package manager: bringing order to HPC software chaos”. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[17] Sibsankar Haldar and Alex Aravind. *Operating Systems*. Pearson Education, 2010. ISBN: 8131730220.

[18] *HDF5 DAOS VOL connector*. URL: https://bitbucket.hdfgroup.org/projects/HDF5VOL/repos/daos-vol/browse (visited on 2020-08-27).

[19] *HPC I/O Middleware – DAOS v1.0*. URL: https://daos-stack.github.io/user/hpc/ (visited on 2020-08-14).

[20] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. “NVRAM-Aware Logging in Transaction Systems”. In: *Proc. VLDB Endow.* 8.4 (2014-12), pp. 389–400. ISSN: 2150-8097. DOI: 10.14778/2735496.2735502. URL: https://doi.org/10.14778/2735496.2735502.

[21] Intel Corporation. *DAOS Community*. URL: https://wiki.hpdd.intel.com/display/DC/DAOS+Community+Home (visited on 2019-02-24).

[22] Intel Corporation. *DAOS Community*. URL: https://github.com/daos-stack/daos/blob/master/LICENSE (visited on 2020-08-27).

[23] Intel Corporation. *DAOS Object*. URL: https://github.com/daos-stack/daos/blob/master/src/object/README.md (visited on 2019-02-27).

[24] Intel Corporation. *DAOS Quick Start Guide*. URL: https://github.com/daos-stack/daos/blob/master/doc/quickstart.md (visited on 2019-02-24).

[25] Intel Corporation. *DAOS README*. URL: https://github.com/daos-stack/daos/blob/master/README.md (visited on 2019-01-23).

[26] Intel Corporation. *Service Replication*. URL: https://github.com/daos-stack/daos/blob/master/src/rdb/README.md (visited on 2019-02-27).

[27] Simon Kostede. “Performanceanalyse der Ein- / Ausgabe des Ökologiemodells ECOHAM5”. MA thesis. Universität Hamburg, 2016-08.

[28] Greg Kroah-Hartman. *Manual driver binding and unbinding*. URL: https://lwn.net/Articles/143397/ (visited on 2020-08-27).

[29] Michael Kuhn. “JULEA: A Flexible Storage Framework for HPC”. In: *High Performance Computing*. Ed. by Julian M. Kunkel et al. Cham: Springer International Publishing, 2017, pp. 712–723. ISBN: 978-3-319-67630-2.

[30] Hermann-J. Lenhart et al. “Predicting the consequences of nutrient reduction on the eutrophication status of the North Sea”. In: *Journal of Marine Systems* 81.1-2 (2010-04). Symposium on Advances in Marine Ecosystem Modelling Research JUN 23-26, 2008, Plymouth, ENGLAND, pp. 148–170. URL: http://nora.nerc.ac.uk/id/eprint/13939/.

[31] Jialin Liu et al. “Evaluation of HPC Application I/O on Object Storage Systems”. In: (2018-11). URL: http://conferences.computer.org/scw/2018/pdfs/PDSW-DISCS2018-7eAHkUBB2dWhq2ImKsykIN/5DJPALrRZcBywbLvt3wESZ/74djJsykXVavOscu8abhsU.pdf.

[32] Jay F. Lofstead et al. “DAOS and Friends: A Proposal for an Exascale Storage System”. In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), pp. 585–596. DOI: 10.1109/SC.2016.49. URL: http://pages.cs.wisc.edu/~johnbent/Pubs/lofstead_sc16.pdf.

[33] Ina Lorkowski et al. “Interannual variability of carbon fluxes in the North Sea from 1970 to 2006 – Competing effects of abiotic and biotic drivers on the gas-exchange of CO2”. In: *Estuarine, Coastal and Shelf Science* 100 (2012). Recent advances in biogeochemistry of coastal seas and continental shelves, pp. 38–57. ISSN: 0272-7714. DOI: https://doi.org/10.1016/j.ecss.2011.11.037. URL: http://www.sciencedirect.com/science/article/pii/S0272771411004987.

[34] David Mosberger and Stephane Eranian. “IA-64 Linux Kernel: Design and Implementation”. In: (2001-01).

[35] *MPI-IO Code, Build, and Testing*. URL: https://wiki.hpdd.intel.com/display/DC/MPI-IO+Code%2C+Build%2C+and+Testing (visited on 2020-08-27).

[36] *MPICH 3.4a3 released*. URL: https://www.mpich.org/2020/07/06/mpich-3-4a3-released/ (visited on 2020-08-14).

[37] Johannes Pätsch and Wilfried Kühn. “Nitrogen and carbon cycling in the North Sea and exchange with the North Atlantic—A model study. Part I. Nitrogen budget and fluxes”. In: *Continental Shelf Research* 28.6 (2008), pp. 767–787. ISSN: 0278-4343. DOI: https://doi.org/10.1016/j.csr.2007.12.013. URL: http://www.sciencedirect.com/science/article/pii/S0278434307003470.

[38] *POSIX Namespace – DAOS v1.0*. URL: https://daos-stack.github.io/user/posix/ (visited on 2020-08-14).

[39] I. S. Reed and G. Solomon. “Polynomial Codes Over Certain Finite Fields”. In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304. DOI: 10.1137/0108018. eprint: https://doi.org/10.1137/0108018. URL: https://doi.org/10.1137/0108018.

[40] Rew Russell, Edward Hartnett, and John Caron. “NetCDF-4: Software Implementing an Enhanced Data Model for the Geosciences”. In: 2006-01.

[41] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[42] *SPDK User Space Drivers Documentation*. URL: https://spdk.io/doc/userspace.html (visited on 2020-08-27).

[43] R Thakur, E Lusk, and W Gropp. “Users guide for ROMIO: A high-performance, portable MPI-IO implementation”. In: (1997-10). DOI: 10.2172/564273.

[44] Rajeev Thakur, William Gropp, and Ewing Lusk. “On Implementing MPI-IO Portably and with High Performance”. In: *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*. IOPADS ’99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 23–32. ISBN: 1581131232. DOI: 10.1145/301816.301826. URL: https://doi.org/10.1145/301816.301826.

[45] V. M. Weaver. “Self-monitoring overhead of the Linux perf_event performance counter interface”. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 102–111.

[46] Vincent M. Weaver. “Linux perf_event Features and Overhead”. In: (2013). URL: http://web.eece.maine.edu/~vweaver/projects/perf_events/overhead/fastpath2013_perfevents.pdf.

[47] Sage Weil et al. “Ceph: A Scalable, High-Performance Distributed File System.” In: 2006-11, pp. 307–320.

[48] *write(2) Linux User’s Manual*. 2018-02.

Appendices

A. Additional listings

```
diff --git a/serielle_io/InitAndCompile.sh
   ↪ b/serielle_io/InitAndCompile.sh
index 46e256f..20b33a4 100755
--- a/serielle_io/InitAndCompile.sh
+++ b/serielle_io/InitAndCompile.sh
@@ -2,9 +2,9 @@

 . /daos-user/spack/share/spack/setup-env.sh

-spack load netcdf-fortran ^hdf5@1.12.0 ^mpich@3.3.2
-spack load hdf5@1.12.0 ^mpich@3.3.2
-spack load mpich@3.3.2
+spack load netcdf-fortran ^hdf5@1.12.0 ^mpich@3.4a2
+spack load hdf5@1.12.0 ^mpich@3.4a2
+spack load mpich@3.4a2

 if [ -z "$3" ]; then
    export ECOHAM_NO_CORES=1
diff --git a/serielle_io/src/eco_output_3D.f90
   ↪ b/serielle_io/src/eco_output_3D.f90
index eab4033..04d40c6 100644
--- a/serielle_io/src/eco_output_3D.f90
+++ b/serielle_io/src/eco_output_3D.f90
@@ -568,7 +568,7 @@
        endif !if (n_full3D .gt. 0)

 #ifdef NETCDF
-       filename = trim(filename)//'.nc'
+       filename = 'daos:'//trim(filename)//'.nc'
        call write_log_message('   write output to
          ↪ NETCDF-file: '//trim(filename))

        ! allocate and initialize netcdf 2D-output metadata
```

Listing A.1: Patching ECOHAM for usage with MPICH ADIO driver

```
# Setup base path for all applications
export basedir="$HOME"

# Setup application locations
export daosdir="$basedir"'/daos'
export mpichdir="$basedir"'/mpich'
export ecohamdir="$basedir"'/ecoham'
export spackdir="$basedir"'/spack'
export daosvoldir="$basedir"'/daos-vol'

# Prepend daos directories to PATH and CPATH
export CPATH="$daosdir"'/install/include/:'"$CPATH"
export PATH="$daosdir"'/install/bin/:\
"$daosdir"'/install/sbin/:'"$PATH"

# Prepend daos directories to LD_LIBRARY_PATH
export LD_LIBRARY_PATH="$daosdir"'/install/lib/:\
"$daosdir"'/install/lib64/:\
"$daosdir"'/install/include/:'"$LD_LIBRARY_PATH"

# Set network device that DAOS should use
export OFI_INTERFACE=eth0

# Setup Spack
source "$spackdir"'/share/spack/setup-env.sh'

# Setup mount directory for DFS
export dfsmntdir="$basedir"'/dfs_mnt'
mkdir -p "$dfsmntdir"

# Setup ECOHAM output directory
export ecohamoutputdir="$basedir"'/ECOHAM_Output'
mkdir -p "$ecohamoutputdir"

# Setup patch directory
export ecohampatchdir="$basedir"'/ecoham_patches'
```

Listing A.2: `load_daos_env.sh` – Shell script to setup environment variables for DAOS

```c
#include <stdio.h>
#include <stdlib.h>

#include "hdf5.h"
#include "common.h"

int main(void) {
    int rc = EXIT_SUCCESS;
    hid_t file_id;
    hid_t dataset_id;
    hid_t dataspace_id;
    herr_t status;
    int i;
    const hsize_t dims[1] = { DIM_X };

    int *write_data = malloc(DIM_X * sizeof(int));
    int *read_data = malloc(DIM_X * sizeof(int));

    for (i = 0; i < DIM_X; i++) {
        write_data[i] = i % 10;
    }

    file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC,
      ↪ H5P_DEFAULT, H5P_DEFAULT);
    dataspace_id = H5Screate_simple(1, dims, NULL);
    dataset_id = H5Dcreate2(file_id, DATASET,
      ↪ H5T_STD_I32BE, dataspace_id,
                H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
    H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
              H5P_DEFAULT, write_data);
    H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
             H5P_DEFAULT, read_data);
    H5Dclose(dataset_id);
    H5Sclose(dataspace_id);
    H5Fclose(file_id);
    free(write_data);
    free(read_data);
    return rc;
}
```

Listing A.3: DAOS-VOL proof-of-concept, write application (abridged)

List of Figures

List of Listings

List of Tables