CoSS: Proposing a Contract-Based Storage System for HPC

Matthieu Dorier, Matthieu Dreher, Tom Peterka, Robert Ross
Argonne National Laboratory
Lemont, IL
{mdorier,mdreher,tpeterka,rross}@anl.gov

ABSTRACT

Data management is a critical component of high-performance computing, with storage as a cornerstone. Yet the traditional model of parallel file systems fails to meet users' needs, in terms of both performance and features. In this paper, we propose CoSS, a new storage model based on contracts. Contracts encapsulate in the same entity the *data model* (type, dimensions, units, etc.) and the *intended uses* of the data. They enable the storage system to work with much more knowledge about the input and output expected from an application and how it should be exposed to the user. This knowledge enables CoSS to optimize data formatting and placement to best fit user's requirements, storage space, and performance. This concept paper introduces the idea of contract-based storage systems and presents some of the opportunities it offers, in order to motivate further research in this direction.

CCS CONCEPTS

• **Information systems → Data management systems**;

KEYWORDS

HPC, Storage, I/O, Metadata, Data Model, Contract, CoSS

ACM Reference Format:
Matthieu Dorier, Matthieu Dreher, Tom Peterka, Robert Ross. 2017. CoSS: Proposing a Contract-Based Storage System for HPC. In *PDSWη-DISCS'17: PDSWη-DISCS'17: Second Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3149393.3149396

1 INTRODUCTION

High-performance computing has inherited the traditional file-centric storage approach in the form of parallel file systems. With an ever increasing gap between computational performance and the performance of such storage systems, however, the community has long recognized that the POSIX file system interface was hardly suitable for data management on supercomputers [27].

Nevertheless numerous researchers have provided optimizations to this I/O stack: better I/O algorithms in MPI [23], different data organizations on backend storage [1], and richer data formats [16, 21]. The fact that such optimizations cannot leverage information from higher levels in the stack is evidence of the inadequacy of the file-centric approach to HPC data management. As an example, the file system cannot know that a particular file contains a 2*D* array of double-precision values, yet such information could be useful to infer future accesses to such a file.

The traditional HPC storage stack must manage various forms of metadata, which we categorize in four levels and which are spread across multiple components of the system.

Level 4: Data model. The data model is the part of the metadata that allows one to make sense of the data objects from a scientific point of view. It gives a data object its type (from basic types such as integers and floats to more elaborate, user-defined datatypes), its dimensions, its name and description, its unit, and its relation to other data objects, for example within a hierarchy of groups. The data model remains the same whether the data is in memory or is stored in a file. It is made explicit through a data format, or implicitly via community standard practice.

Level 3: Data format. The data format describes the mapping between the data model and the storage media, which is typically a file in the context of a file system but could be a row within a relational database or any other such mapping. It provides the layout of the data (row/column-major order, endianness, compression, chunking, etc.) and the organization of associated metadata (presence of headers and footers).

Level 2: File metadata. The file metadata comprises the file's standard POSIX attributes such as location within a hierarchy of directories, the permissions, and the date of creation / modification. It may include other attributes added by a user, for example through extended attributes (xattr).

Level 1: Distribution. The file's distribution represents how the file is split and distributed across multiple storage servers and, within these servers, across storage devices. It also describes whether data is replicated or protected using erasure coding. Such distribution can be controlled in some cases; for example OrangeFS exposes it to users.[1]

This file-centric organization is not adequate for users of HPC platforms. **A file, while presenting a convenient way of storing data, is not the central concept to scientific computing. Rather, data objects and the data models that describe them are the key concepts.** Hence a proper storage system for HPC should be object-centric and encapsulate all levels of metadata, in order to provide the user with the data models describing these objects.

Object-centrism is one of the reasons for the success of in situ analysis and visualization, a technique that bypasses the file system to directly communicate objects from simulations to analysis applications. The design of an object-centric storage system could thus benefit from recent advances in this area.

In this paper, we present the concept behind CoSS, of a **contract-based, object-centric storage system** with augmented knowledge about the data semantics. A contract-based storage system gathers the capabilities of an object storage system (e.g., RADOS [25]), with the high-level data models found in scientific data formats (e.g., HDF5, NetCDF), and the notion of contract currently found in

[1] http://dev.orangefs.org/trac/orangefs/wiki/Distributions

PDSW-DISCS'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5134-8/17/11…$15.00
https://doi.org/10.1145/3149393.3149396

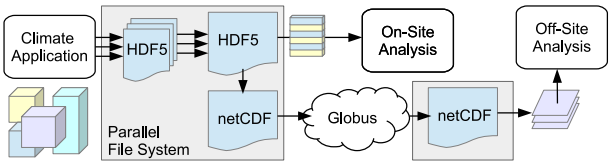

Figure 1: Example of file-centric data flow from a simulation to various analysis tools on- and off-site.

some in situ analysis frameworks [20] and visualization tools [4]. The key features of such a storage system can be summarized as follows.

Working with high-level metadata. CoSS directly exposes a high-level data models, including the information currently found in data formats such as HDF5, with additional information linking data objects together to build complex structures and to express consistent datasets.

Expressing the intended usage through contracts. CoSS is made aware of the intended usage of the data and can use this knowledge to manage the way data objects are written, stored, transformed, and exposed to readers.

2 MOTIVATING EXAMPLE

The shortcomings of current file-based storage systems and the motivations for a contract-based storage system are best illustrated by an example.

2.1 Example description

Let us consider a climate simulation writing multiple field variables such as temperature, pressure, and wind speed. Such an application periodically writes a set of HDF5 files (in this example one per process) for analysis and visualization, and for later restart.

We will assume that the application needs to keep the data in many small files (say, to ease the restarting process). However, the analysis program requires a single file. Hence, some postprocessing is done to combine all the files of an iteration into a large HDF5 file. From these large files, the user extracts subarrays from two fields and combines them (for example, computing the norm of the wind speed vectors at various altitudes).

Collaborators also need to analyze part of the data produced by the simulation, and for this example we will assume that they use a tool that understands only the NetCDF format. Thus the data has to be converted into NetCDF and sent over the WAN to a remote Globus endpoint. From these files, the collaborators can then extract slices of the temperature field to visualize them.

2.2 Issues posed by the file-centric approach

This file-centric data flow is shown in Figure 1. As the community around an HPC simulation grows, this flow can grow much more complex, with the application itself presenting backends for a number of data formats, along with many handwritten conversion and postprocessing tools.

The first problem of this approach is data redundancy. In our example, we find redundant data in several places. The large HDF5 files generated from gathering the per-process files contains the same or similar data as these small files have, simply organized in a different manner. The same goes for the NetCDF output, which is generated and stored for the sole purpose of being sent to and analyzed by collaborators. Assuming the analysis applications need to know the coordinates of mesh vertices, these coordinates, even though they may remain constant throughout the simulation, are likely to be stored in each and every file to make each file self-contained. This approach again creates redundancy.

These files may also contain too much data compared with what is really needed by reader applications. Suppose a field variable (e.g., pressure) is needed only for restarting the simulation but is never analyzed. The presence of this variable in the output HDF5 files is useful only for the latest iteration of data but pollutes files containing previous iterations. Yet this variable cannot just "disappear" from those files when one notices that it is not going to be useful anymore.

Lossy compression, downsampling, and reduction of precision are operations that reduce the amount of data stored in files. Yet for conservative reasons they may not be applied by the user. Assume that the application writes double-precision data, but that the visualization code can accept single-precision data and afford the loss of accuracy generated by a lossy compressor, while producing the same visual accuracy. The fact that the file may be needed in contexts where a lossless, double-precision format is required (e.g., to restart the application) will prevent the user from considering such reduction techniques even for variables that are not needed in these contexts.

One may argue that all the problems identified above can be solved by agreeing to use a single data format (say, HDF5) for all codes, preferably the format that carries the semantics required by all of them, by writing data objects in separate files to be able to work on them and share them individually, and by applying the best reductions or transformations possible given the constraints imposed on each data object individually. This highlights two key ideas: (1) what is important is not the file or the data format but the *data objects* themselves and their semantics; and (2) a priori knowledge of the data usage can help optimize the storage of such data.

Yet even if all the code-base around a simulation used an agreed-upon format and the question of *which* data to stored was fully answered, the questions of *where* and *when* the data should be transformed remains. Some transformations, such as slicing, are easy to apply to the in-memory data inside the simulation, before writing. Some, however, may be expensive or hard to parallelize (e.g. compression) or may require extra memory that the simulation may not have. Some transformations may be pipelined, such as slicing followed by compression, and even be commutative, such as downsampling followed by reduction from double to single precision. Current parallel file systems have no way to select *where*, *when*, and *how* such transformations should be applied, and cannot apply them within the file system itself.

This example lays the ground for a new type of storage system, which we describe in the next section.

3 TOWARD CONTRACT-BASED STORAGE

Building on the hypothesis that applications and data objects (rather than files) are at the center of HPC workflows, we propose a new design for an HPC storage system named CoSS (Contract-based Storage System). The first goal of CoSS is to collocate the various types of metadata currently scattered across all layers of the storage stack. Its second goal is to provide the user with *views* that are appropriate for the applications that manipulate the data.

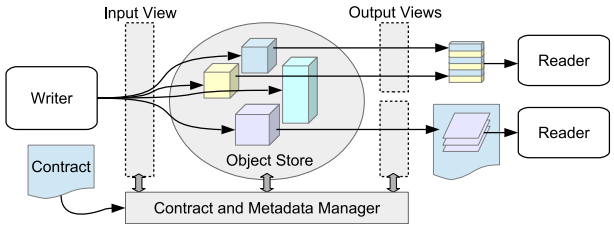

Figure 2: Overview of CoSS, with its object store and contract and metadata manager.

3.1 Overview of CoSS

Figure 2 shows an overview of CoSS. The actual storage space is managed by an object store, which stores data in the form of objects identified by a unique *id*. The Contract and Metadata Manager is the entity that makes sense of these objects according to user-provided information in the form of a *contract*.

3.1.1 Object store. The object store is write-once-read-many. Stored objects cannot be modified. The community has already recognized that most HPC codes do not actually modify previously written data. Rather, they rely on explicit versioning (such as creating a new file at every iteration of the simulation). However, objects may be written through multiple operations and potentially by multiple writers. We envision objects to be associated with policies specifying how they are populated, for example:

- **Atomic access:** The object will be written in a single operation by a single process. Once written, the object is considered in a consistent state.
- **Chunked access:** The object will be written in multiple operations, potentially by multiple writers, but the definition of that object enables CoSS to know when the object is in a consistent state. For example, a multidimensional array written chunk by chunk is in a consistent state when the union of all chunks make up the entire array.
- **Log-structured access:** The object will be written in multiple operations, potentially by multiple writers that will append data to it. An explicit "commit" operation by a writer will let CoSS know that the object is in a consistent state and should no longer be written to. An example of such an object is a list of particles whose length is unknown a priori.

In a typical file system, partial writes to a file make the file immediately visible to potential readers even though the data inside is incomplete. Enabling the storage system to have a notion of the conditions under which a data object is consistent prevents the system from exposing inconsistent objects to readers, and having a policy-based approach avoids the need for a single mechanism that supports all modalities (e.g., a distributed transaction).

3.1.2 Object management. Object management is done by the contract and metadata manager. Objects are logically gathered into *projects*. A project has several *branches*, each corresponding to an execution of the simulation. A branch groups objects into several *epochs*, which could correspond to iterations of the simulation. Projects and branches have names. Epochs are identified by a positive integer. Whenever the simulation runs, a new branch is created. Whenever the simulation enters an I/O phase, it opens an epoch, writes objects, and then closes the epoch. Upon closing of the epoch, the objects of that epoch become visible to reader applications. Any nonconsistent objects are discarded from the epoch.

The contract and metadata manager also manages level 1 and 2 metadata: it assigns permissions and creation/modification times to projects, branches, and epochs, and manages the underlying data distribution in the object store.

3.1.3 Contracts. A project has an associated *contract*. The contract is defined either by a document (e.g., in XML, JSON, or YAML), or programmatically through the execution of a script (e.g., in Python). The contract contains various information, such as:

- The *data model*, which includes information typically available in data formats like HDF5 and NetCDF, such as the data type, dimensions, layout, and compression. The semantics may also include information required by visualization software (and that an XDMF file would typically provide), namely, relationships between objects allowing complex structures such as meshes to be built, and field variables to be mapped onto these meshes.
- The *views*, which express how the data should be presented to applications that access it. Views are divided into an *input view*, on the writer side, and one or more *output views*, on the reader side.

3.2 Views

The *input view* defines how the data is produced by the writer application. This includes its layout within the memory of the writing application.

Output views place constraints on how the storage system might organize and/or reduce the data in the objects that make up the project. For example, while the input view specifies that the application will write a $3D$ array A of double-precision values, an output view may require to expose it to readers as a $2D$ array B corresponding to the first slice of the array A, in single-precision values.

Alternatively, an output view may specify that a set of objects should be made available in the form of a file, that is, a single object representing a byte stream that can be transferred to a traditional file system and be read as a file (for example in HDF5 format).

Views also provide a requested level of resilience, which is used by CoSS to replicate the data and protect it with adequate error correcting mechanisms.

3.3 Storage system intelligence

Knowing in advance what an application will produce and how the data will be consumed, allows CoSS to perform a number of optimizations. In particular, **the notion of input and output views represents a contract between the producer and consumers on one side, and the storage system on the other side**. It allows the latter to *match* views: not only can CoSS check that the produced data *can* be exposed to the consumers in the required form; it also decides *where*, *when*, and *how* to transform the data such that it satisfies the output views, while optimizing performance and resource usage.

In the example of the $3D$ array A exposed in the output view as a $2D$ array B with reduced precision, CoSS can choose between several places and times to operate the slicing and reduction. These options are illustrated in Figure 3.[2] The decision can be made by

[2]Note that a traditional parallel file system does not enable the in-storage options. Additional tools or scripts are typically written by users to convert data from one representation to another.

CoSS depending on the current load, memory constraints, run time constraints, and concurrency. For example the reduction to single precision can be done on the writer side provided that the writer has enough memory. It can otherwise be done in storage or by the reader itself. CoSS can also chose to keep both versions, knowing that another consumer still needs A as an input.

Regardless of whether the transformation has been performed or not, CoSS will make the output view available to the reader *as soon as it has the guarantee that such a view can be satisfied*, that is, when all objects required for the view are in a consistent state.

3.4 Updating a contract

Updating a contract can either *restrict* the existing views, or *widen* them.

An update that restricts the input and output views must do so in such a way that the resulting views remain matching. Restricting a view V_1 to a new view V_2 is understood in the sense that if data is provided under a view V_1, then a series of transformations can make it available under the view V_2. For example, in our previous scenario, one could want the new input view for array A to expose it as single-precision values. This new view restricts the previous one (which was exposing double-precision values) and still satisfies the requirements of the output view.

Widening a view consists of enabling more data to be visible under the new view than was under the old one. Widening updates may or may not be accepted by the storage system, since it must have kept the data required to build the new view despite it being unnecessary under the old view. For example, if the new output view requests the array B to be a $3D$ array instead of a $2D$ slice, the update to the contract will be allowed only if the storage system actually kept $3D$ arrays and did not proactively transform them.

3.5 Interfaces and convertors

We envision CoSS to expose an interface similar to that of Damaris [9] or ADIOS [19], where read and write functions are being passed a name (string) and a pointer to the data to access. The rest of the semantics (including the layout of the data in the application's memory) are accessible through the contract and metadata manager.

The API provided by CoSS can be used underneath I/O libraries that enable a high-level API and are implemented with backend modularity, such as HDF5 and its file drivers, ADIOS and its transport methods, and Damaris with its plugins. This will make any application that uses these libraries immediately compatible with the storage system.

Alternatively, generic convertors can be provided to transform objects from the storage system into files in a given format. The availability of both an interface and a convertor for a given format enables deeper automatic optimizations by CoSS. For example, if an output view declares that the data will be accessed as an HDF5 file, the system can let the reader application use the system's I/O driver in HDF5 to make sense of objects directly, or it can convert the objects into an HDF5 file to later provide a byte-stream read through HDF5's POSIX I/O driver.

4 RELATED WORK

Over the past years, researchers have proposed solutions to improve HPC I/O. Some of these solutions improve the existing parallel file system's performance by providing more efficient file-based interfaces or by providing data formats suitable to scientific datasets.

Many researchers, however, turned toward in situ analysis and visualization to bypass the storage system and connect a simulation with an analysis application. Such connections require the semantics of the data to be kept along the data path; hence they either rely on existing metadata-rich I/O interfaces such as HDF5 or ADIOS [2] or require users to modify their code. In both cases, this trend hints at the fact that an object-centric approach to data management is preferable to a file-centric one.

4.1 Data models

Damaris [9], a framework that enables using dedicated cores and nodes for I/O and data processing, uses contracts in the form of an XML file. This file describes the variables (*data objects*) expected to be output by an application. It allows dedicated processes to work with a priori knowledge of metadata. This description enables Damaris to transform the data and pass it to relevant backends such as HDF5 (for storage), VisIt [17], or ParaView [15] (for in situ visualization). In order to support in situ visualization, Damaris' XML model was extended to include the description of entities such as meshes. They effectively represent *relationships* between data objects.

FFS [13] is a data model used with EVpath [14] and meant to enable semantically rich communications across application components. Rather than relying on agreed-upon knowledge of data objects expected to be sent between components, it proposes these metadata be embedded within messages. FFS is used by the Flexpath [7] publish/subscribe mechanism, in which consumers can register to be notified of the production of data of specific types.

Bredela [11], proposed by Dreher et al. describes the data output by a simulation in a way that permits semantically aware data manipulation, in particular splitting, merging, and redistribution across components. Bredela gives addresses how our contract-base storage system can make sense of data that is written by multiple processes.

Conduit [18] uses JSON to manipulate data models. These data models include data objects as well as complex structures such as meshes. Damaris's XML, ADIOS's XML, and Conduit's JSON data models are, in our opinion, viable candidates for implementing contracts.

4.2 Data transformation

Exchanging data between two tasks often requires transforming the data. The transformation can be done in situ or as a postprocessing step, converting data from one file format to another for instance. Transformations are required for different reasons: data model mismatch between two components, data compression, selection of a subset of data for a particular analysis, and so forth.

Several in situ infrastructures offer mechanisms to transform data along the I/O path. ADIOS [2] introduced a transformation step performed synchronously when the application is writing data. PreData [28] used small *codelets* to rearrange the data structure between two tasks. Decaf [12] enhanced the communication channel between two tasks with a dedicated staging area to transform and redistribute the data between tasks.

Dorier et al. [10] proposed taking into account some measures (e.g., entropy) of the relevant data to perform in situ reduction of part of the data and accommodate the need in subsequent visualization. Such techniques could be integrated in a contract-based storage system since views can describe the required level of details.

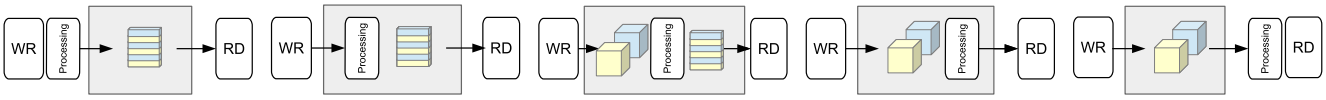

Figure 3: Possible places for a transformation filter to be applied. The grey box represents the storage system. A producer application is on the left, and a consumer on the right.

In all these systems, the user is providing the functions necessary to transform data between tasks. However, these transformations could be automated with more information from the user or from the tasks themselves.

The notion of a *contract* was introduced by Childs et al. [4] to describe the inputs and outputs of components. This information enables new methods to automatically transform data between the tasks. It is what enables VisIt to optimize its data flows within a visualization pipeline. In VisIt, each component of a pipeline describes its requirements and its impact on the data. For example, given a component A reading a $3D$ dataset and a component B extracting a slice from it, the requirements of component B ("I need only the parts of the data that intersect with the slice") are propagated upstream in the pipeline so that component A reads only the appropriate parts of the input file. This is similar to our contracts, in which a priori knowledge of the output views can be propagated back to the writing application and used by the storage system so that only the relevant data is stored and potentially transformed along the data path.

Mommessin et al. [20] adopted a similar approach for in situ workflows. Their contracts describe the data that a component will output and what it expects as input. This approach enables components to be connected by matching their contracts, and it effectively filters the content of a message output by a producer to contain only the data required by the consumer. It also enables an in situ runtime to potentially transform data automatically between two tasks, for instance transforming an array of double-precision floating point values into an array of single-precision floating point values.

4.3 Data formats

The HDF5 format enables data objects to be stored in a hierarchical manner. It allows the user to work with data objects rather than with the file itself, by abstracting how the data is laid out in the underlying file. While HDF5 is unable to express how data objects can be composed to make up a complex structure like a mesh, additional information can be provided by using XDMF.[3] HDF5 files coupled with an XDMF description can easily be read by visualization tools such as VisIt. VizSchema[4] targets a goal similar to that of XDMF.

The popularity of HDF5, NetCDF, and other similar data formats exposing a rich data model supports the fact that users prefer to work with data objects rather than with files. This motivates the design of a storage system that conserves the same level of semantics as such formats.

4.4 Storage

Many of today's parallel file systems such as Lustre [8], Ceph [24], PVFS [3], and PanFS [26] are characterized as "object-based" in the sense that they decouple metadata management from data storage and store data as objects in a flat namespace. These objects are usually hidden from the users, who see a typical file system interface with its hierarchy of folders and files. This shows that the right trend to managing objects has existed in the community for years, but the convenience of a file system interface prevented it from being used to its full potential. Ceph does expose a direct access to objects from its underlying RADOS object store.

The concept of *view* at storage level can be found in the Vesta parallel file system [5], where different views of the same file can be exposed to different processes in a way similar to what MPI file views do at application level. However such views do not rely on high-level data semantics. They describe regions of the file's data, for example to help partitioning a file's data across processes. The views we propose describe series a transformations that can be applied to the data in order to go from its stored form to a requested form.

Closer to our proposed storage system is SciDB [6]. SciDB is a storage system for scientific applications. It is array-oriented rather than file-oriented. It keeps a high-level data model and enables running data manipulation queries (e.g., array slicing, chunking), within the database and in a parallel manner. Contrary to our contract-based storage model, however, SciDB does not know anything about the intended use of the data and cannot optimize the data layout or transform the data ahead to respond faster to readers later.

The notion of programmable storage systems also has started to appear [22]. Such programmability would be a way to let users implement object filters to enable views from nonconventional data structures.

5 CONCLUSION

We have proposed CoSS, and the concept of a *contract-based storage system*. Such a storage system is an object store augmented with the notion of a contract between producers, storage, and consumers. This object-centric system is more natural to scientific data storage than the traditional file system approach; and the knowledge gained by the storage system about the intended use of the data enables automatic optimizations by allowing the system to choose *what* to store and *how*, *when*, and *where* to transform the data from what a producing application generates to what a consuming application expects.

The building blocks of such a storage system already exist. Object stores make up the storage layer of many current parallel file systems. File formats and in situ analysis libraries provide the high-level of semantics and the filters required to implement the contracts, respectively.

We plan to develop the concept behind CoSS further and to design, implement, and evaluate a prototype.

[3] http://www.xdmf.org/index.php/XDMF_Model_and_Format
[4] https://ice.txcorp.com/trac/vizschema

ACKNOWLEDGEMENTS

This material was based upon work supported by the U.S. Department of Energy, the Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02- 06CH11357, program manager Lucy Nowell. This work was done in the context of the DOE SSIO project "Mochi" (http://press3.mcs.anl.gov/mochi/), a Software Defined Storage Approach to Exascale Storage Services.

REFERENCES

[1] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. IEEE, 1–12.

[2] D.A. Boyuka, S. Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, J. Jenkins, E.R. Schendel, N. Podhorszki, Qing Liu, S. Klasky, and N.F. Samatova. 2014. Transparent I Situ Data Transformations in ADIOS. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 256–266.

[3] Phil H Carns, Walter B Ligon, Robert B Ross, and Rajeev Thakur. 2000. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux Showcase and Conference*.

[4] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. 2005. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*. IEEE, 191–198.

[5] Peter F Corbett and Dror G Feitelson. 1996. The Vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)* 14, 3 (1996), 225–264.

[6] Philippe Cudré-Mauroux, Hideaki Kimura, K-T Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L Wang, Magdalena Balazinska, Jacek Becla, and others. 2009. A demonstration of SciDB: a science-oriented DBMS. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1534–1537.

[7] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert Podhorszki. 2014. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 246–255.

[8] Stephanie Donovan, Gerrit Huizenga, Andrew J Hutton, C Craig Ross, Martin K Petersen, and Philip Schwan. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*.

[9] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. 2016. Damaris: Addressing performance variability in data management for post-petascale simulations. *ACM Transactions on Parallel Computing (TOPC)* 3, 3 (2016), 15.

[10] Matthieu Dorier, Robert Sisneros, Leonardo Bautista Gomez, Tom Peterka, Leigh Orf, Lokman Rahmani, Gabriel Antoniu, and Luc Bougé. 2016. Adaptive Performance-Constrained In Situ Visualization of Atmospheric Simulations. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 269–278.

[11] Matthieu Dreher and Tom Peterka. 2016. Bredala: Semantic data redistribution for in situ applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 279–288.

[12] M. Dreher and T. Peterka. 2017. *Decaf: Decoupled Dataflows for In Situ High-Performance Workflows*. Technical Report.

[13] Greg Eisenhauer, Matthew Wolf, Hasan Abbasi, Scott Klasky, and Karsten Schwan. 2011. A type system for high performance communication and computation. In *IEEE Seventh International Conference on e-Science Workshops (eScienceW)*. IEEE, 183–190.

[14] Greg Eisenhauer, Matthew Wolf, Hasan Abbasi, and Karsten Schwan. 2009. Event-based systems: opportunities and challenges at exascale. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 2.

[15] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Gevecik, Michel Rasquin, and Kenneth E Jansen. 2011. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 89–96.

[16] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 36–47.

[17] T Kuhlen, R Pajarola, and K Zhou. 2011. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*.

[18] Lawrence Livermore National Laboratory. Conduit: A scientific data exchange library for HPC simulations. http://software.llnl.gov/conduit/index.html. (????).

[19] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, and others. 2014. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[20] Clement Mommessin, Matthieu Dreher, and Tom Peterka. 2017. Automatic Data Filtering for In Situ Workflows. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE.

[21] Russ Rew and Glenn Davis. 1990. NetCDF: An interface for scientific data access. *IEEE Computer Graphics and Applications* 10, 4 (1990), 76–82.

[22] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A Programmable Storage System. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 175–190.

[23] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*. IEEE, 182–189.

[24] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*. USENIX Association, 307–320.

[25] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. 2007. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale Data Storage: held in conjunction with Supercomputing'07*. ACM, 35–44.

[26] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. 1–17.

[27] Erez Zadok, Dean Hildebrand, Geoff Kuenning, and Keith A Smith. 2017. POSIX is Dead! Long Live... errr... What Exactly?. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association.

[28] Fang Zheng, H. Abbasi, C. Docan, J. Lofstead, Qing Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. 2010. PreDatA - preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS'10)*. 1–12.