An Innovative Storage Stack Addressing the Requirements of Extreme Scale Platforms and Big Data Applications

Jay Lofstead*, Ivo Jimenez†, Carlos Maltzahn†, Quincey Koziol‡, John Bent§, Eric Barton¶
*Sandia National Laboratories gflofst@sandia.gov
†University of California, Santa Cruz ivo@cs.ucsc.edu,carlosm@soe.ucsc.edu
‡HDF Group koziol@hdfgroup.org
§EMC john.bent@emc.com
¶Intel eric.barton@intel.com

***Abstract*—The DOE Extreme-Scale Technology Acceleration Fast Forward Storage and IO Stack project is going to have significant impact on storage systems design within and beyond the HPC community. With phase 1 of the project complete, it is an excellent opportunity to explore the complete design and how it will address the needs of extreme scale platforms. This paper not only provides a timely summary of important aspects of the design specifications but also capture the underlying reasoning that is not available elsewhere.**

While many of the design elements are best practices from current research, there are many new contributions as well as different integration approaches offering different performance and quality of service guarantees. The specifics of how the different layers interact offer a new approach for managing scientific data for extreme scale platforms.

The integration of the HDF5 interface as the user interface gives a familiar way to imteract with all of the new functionality while largely insulating end-users from the intracisies of dealing with the advanced functionality offered by the stack.

The optional components aid in the scalability and simplifying implementation across a wide range of platforms.

The data storage layer offers support both for local machine and data center wide storage arrays as part of the design.

I. Introduction

Current production HPC IO stack design is unlikely to offer sufficient features and performance to adequately serve the needs of an extreme scale platform. To address these limitations, a joint effort between the US Department of Energy's Office of Advanced Simulation and Computing and Advanced Scientific Computing Research commissioned an effort to develop a design and prototype for an IO stack suitable for the extreme scale environment. This is a joint effort led by Lawrence Livermore National Laboratory, with the DOE Data Management Nexus leads Rob Ross and Gary Grider as coordinators and contract lead Mark Gary. The participating labs are LLNL, SNL, LANL, ORNL, PNL, LBNL, and ANL. Additional industrial partners contracted include the Intel Lustre team, EMC, DDN, and the HDF Group. This team has developed a specification set [8] for a future IO stack to address the identified challenges. The first phase recently completed with a second phase currently getting underway as of early 2014. The core focus of the first phase was basic functionality and design. Overriding many of the decisions during this and any subsequent phases is the reality of budgets. Placing GBs of NVRAM on every node, while a potentially advantageous approach, is not financially feasible. With this in mind, the second phase will refine this design incorporating fault recovery and other features missing from the first phase.

The overall design seeks to offer high availability, byte-granular, multi-version concurrency control. Through the use of a copy-on-write style mechanism, multiple versions of an object can be stored in potentially greatly reduced space. It assumes the client interface will be through an IO library affording a more complicated interface that offers richer functionality requiring only minimal end-user code changes. Managing most data access in a platform-local layer rather than requiring writing to centralized storage will better support the performance and energy requirements of extreme scale integrated application workflows. At a more detailed view, the various layers of the IO stack each contribute different functionality and performance implications.

The basic architecture incorporates five layers. The top layer is a high level IO library, such as the demonstration HDF-5 library [18]. The intent is to only have access to the storage stack through such an API to manage the complexity of working with the lower layers. Below the user API is an IO forwarding layer that redirects IO calls from the compute nodes to the IO dispatching layer. This IO forwarding layer is analogous to the function of the IO nodes in a BlueGene machine. The next two layers have considerable functionality. The IO dispatcher (IOD) serves as the primary storage interface for the IO stack and is the only way to reach the persistent storage array in lower layers. The Distributed Application Object Storage (DAOS) layer serves as the persistent storage interface and is intended to be the foundation on which everything else is built with no dependence on any technologies specified above it. For example, the IOD layer and burst buffers are not required for DAOS to operate properly. At the bottom is the Versioning Object Storage Device (VOSD). It serves as the interface for storing objects of all types efficiently.

The focus of this paper is primarily the IOD layer given the critical role it has in the performance and functionality of the entire stack. Most of the key features explored in this paper all have a strong presence in the IOD layer motivating the focus of this examination.

The core idea for IOD is to provide a way to manage the IO load that is separate from the compute nodes and the storage array. Communication intensive activities, such as data rearrangement, can be moved to the IOD layer offloading the communication load from the compute nodes. IOD has three main purposes. First, if the optional burst buffer is available, it works as a fast cache absorbing write operations for the slower trickle out to the central storage array. It can also be used to retrieve objects from the central storage array for more efficient read operations and offers data filtering to make client reads more efficient. Second, it offers the transaction mechanism for controlling data set visibility and to manage faults that would prevent a data set from being used. Third, data processing operations can be placed in the IOD. These operations are intended to offer data rearrangement, filtering, and similar operations prior to data reaching the central storage array.

While these ideas are not necessarily new, they are new twists on best of class efforts for these technologies. For example, offloading the collective two-phase data sieving from the compute nodes to reorganize data has proven effective at reducing the total time for writing data due to fewer participants involved in the communication patterns [13]. Beyond these broad items, there are many important details some of which are examined in more detail below.

Along with the analysis of the published design documents, a discussion of the design philosophy representing the overall intent is presented. This information represents information that may or may not have been written down, but is the intent of ultimate product. These insights were gained based on personal conversations with the team members [4], [9], [3] discussing some of the potential challenges with the design as written. These ideas are presented to give a fuller picture of where the project is going rather than dwelling on the limitations of the published documents.

The rest of the paper is organized as follows. Section II discusses some of the features of incorporating burst buffers as designed and suggests some considerations and alternatives for the next generation of this project. Section III discusses the transactions approach offered in the IOD layer and the corresponding epochs in the DAOS layer. It also offers a comparison to the D^2T system given the very similar high-level design and motivating use case. Section IV discusses the system overall with recommendations on what design elements should be considered based on broader issues with current HPC data centers. The paper is concluded in Section V with a summary of the broad issues covered in the paper.

II. Burst Buffers

The idea of burst buffers were initially explored in the context of data staging [2], [1], [14], [20]. These initial designs all use extra compute nodes to represent the data storage buffer given the lack of any dedicated hardware support for this functionality. The desired outcome of these initial studies is to motivate how such functionality might be incorporated and the potential benefits. Later, these concepts were proposed to be incorporated as part of the IO stack [15], [6], [5]. The current Fast Forward IOD design recommends incorporating SSDs, but specifically lists these devices as optional. Unfortunately, not incorporating burst buffers and the use of SSDs in the IOD layer may be problematic. First, the IOD design currently is written assuming burst buffers. This means that the bulk of IO operations will only hit the IOD layer and proposed functionality, such as the function shipping, do not discuss the impact on the DAOS layer or the operators themselves should a burst buffer not be available. Consider the important functionality of data rearrangement and doing things like changing the fast array dimension on shared, spinning media. The design of DAOS assumes that it will only be involved when persisting a completed transaction and only for a fraction of the total transactions created. Transaction frequency at the IOD layer can be higher since IOD does not "flatten" transactions. Flattening is the process of transferring the set of copy-on-write changes from transaction/epoch n to transaction $n + m$ into the DAOS layer. The DAOS layer mirrors the copy-on-write functionality, but stores the differences between epochs rather than individual transactions. The primary difference is that at the DAOS layer, data is stored in large, logically contiguous chunks to manage the metadata load and hopefully improve read performance.

One of the bigger concerns is the observation that the original data staging proposals all used compute nodes while the newer proposals seek not only to make them a fixed portion of the IO stack, but also shared across all machine users. The PreDatA [20] paper in particular examines the potential costs and advantages of where to place operators similar to the IOD proposed function shipping. There are two key takeaways from PreDatA. First, placement matters. Depending on the communication intensity vs. computation intensity, where along the IO path to place the operation can matter significantly. Second, and more importantly, the amount of time spent processing for the operators was stretched to the point where it consumed nearly all of the time between IO operations. The given ratios of compute processes to staging process examined is representative for future extreme scale platforms. If anything, the ratios offer more staging processes than IOD processes would be available.

In the case of the written IOD design, it describes a fixed-sized staging area that is partitioned on a per-application basis. This is unlikely to be useful because of the limited compute and communication capacity to spare to perform these operations at a bottleneck in the IO path. The use of a separate data staging area intentionally separate from the IO path allows using operators on limited resources leaving the IO path clear for strictly data movement. A nuance of this design is discussed in the Design Philosophy below.

By concentrating the Burst Buffers and function shipping into the storage stack, three problems arise. First, the amount of network bandwidth, IO bandwidth, and compute power consumed for example operations from a single application is likely to completely monopolize the IOD processes. Second, if space and time partitioning is used instead, the functionality risks being too small to be useful. Third, the long-term hardware performance advantage for SSDs is questionable. Recent studies have shown that the erase-before-write and interference between reading and writing with flash-based SSDs can cause severe performance problems [17]. The inclusion of an optional use SSD layer in the new Trinity machine at Los Alamos will offer a test bed to determine how likely these observed problems would affect a production extreme scale platform.

Given these features, the optionality and even incorporation of burst buffers in the current design should be carefully considered. Much of the advanced, key functionality proposed as they are currently designed ultimately relies on the existence of burst buffers to work. Further thought about how to have an IOD layer both with and without a burst buffer is required before they can be considered optional. As the design stands today, they are a required part of the IOD layer for proper functioning. Unfortunately, it is not clear that they can address the performance concerns they are intended to cover.

A. *Design Philosophy*

The burst buffers design, as presented in the IOD documents, limits the placement of the function operators and SSD buffers to the IO nodes. The team does acknowledge the limitations of this design and intend to ultimately focus on spreading the IOD layer from the IO nodes into the compute area as well. This is intended both to help address the limitations of the IO bandwidth and compute capability of these few nodes for data processing, but also to take advantage of new layers in the storage hierarchy. By incorporating NVRAM into compute nodes, new options for buffering data prior to being moved to centralized storage become available and addresses some of the concerns about SSD performance. For example, including a small amount of Phase Change memory into many or most compute nodes offers a way to move data outside of both the compute and IO path for data and communication intensive operations. Other projects [20] have suggested this will have value, but the cost will have to be considered as part of the overall platform budget. This lessens the impact of some operators while offering additional options for places to store data.

Burst buffers being optional is a high level goal, but not one considered at a detailed level within the design. For example, if there is no burst buffer, all of the advanced functionality proposed for the IOD layer would have to work against the DAOS layer instead. For example, function shipping assumes it will operate on fast, local data within the IOD layer rather than against the globally shared DAOS layer. With the additional desire to support using compute node resources for these operations, serious work will be required to make a fully functional end-to-end IOD layer implementation for a production system.

Another concern that is acknowledged, but no thought has been applied to, is the requirement that a single IOD process of the set assigned to an application is the master for any operation. Should the number of concurrent applications exceed the available nodes, sharing an IOD process will be required. The requirements both in terms of scheduling and resource management were cut from the project due to funding limitations. Since partitioning of the IOD processes for exclusive use by particular applications is the assumed operating mode, should insufficient IOD resources be available, either a job could be delayed or IOD resources could be reallocated from a different process could be redeployed for use by the new job. Handling resilience concerns for the IOD processes must also be address. These sorts of considerations still need to be made for a full production system.

III. Transactions and Epochs

The transaction mechanism manifests in two forms. At the IOD layer, they are called transactions and are used to judge whether or not a set of distributed, asynchronous modifications across a set of related objects is complete or not. It is also used to control access by treating the transaction ID of committed transaction as a version identifier. At the DAOS layer, they are called epochs and represent persisted (durable) transactions from the IOD layer. Each of these offers different functionality, but are connected as is explained below. How these differ from the D^2T approach is also explored. While IOD's and D^2T's transactions are seemingly very different, they use a similar high-level design, but very different implementation, to solve the same problem.

A. *IOD Transactions*

To understand how transactions are used in the IOD layer, some terminology and concepts must be explained first. At the coarsest grain level is a container. Each container provides the single access context through which to access a collection of objects. Transactions are the way that a series of modifications to the objects within a container are treated atomically. Conceptually, containers corresponds to a something akin to an HDF5 file in a traditional file system. The objects in each container represent different data within a file. The three initially defined object types are key-value stores, multi-dimensional arrays, and blobs. The easiest way to understand these types is to evaluate these from the perspective of an HDF-5 file, the initial user interface layer. The key-value store represents a collection of attributes or groups. The array represents a potentially multi-dimensional array. The blob represents a byte stream of arbitrary contents. The fundamental difference between an array and a blob is that the array has metadata specifying the dimension(s). At the physical layer within the IO nodes, all of these objects may be striped across multiple IO nodes. Given this context, the transactions come in two forms.

First is a single leader transaction where the IOD manages based on calls from a single client. The underlying assumption is that the client side will manage the transactional operations itself and the single client is capable of reporting to the IOD how to evolve the transaction state.

The second form is called multi-leader and has the IOD layer manage the transactions. In this case, when the transaction is created, a count of clients is provided to the IOD layer. As clients commit their changes to the container, the reference count is reduced. Once the count reaches 0, the transaction is automatically committed.

1) Design Philosophy: Undocumented, but inherent in the design of these transactions is how faults are detected. The initial design assumes the current Lustre fault detection mechanism that can determine if a process or node is no longer reachable. This detection happens at the DAOS layer and when a fault is detected, the rollback process is pushed up to the IOD layer for all non-persisted or non-committed transactions. This defines how a fault will be detected and what will trigger a passive fault recovery (i.e., transaction abort).

There are two steps for beginning a transaction on a container. The first step is for one or more process to open the container. This handle can be shared eliminating the need for every participating process to hit the IOD layer to open the file. The second step is a call to determine how many leaders will participate in the transaction. In the single leader case, there is no aggregation of success/fail statuses to determine the final transaction state. Instead, it is assumed that the client will fully manage the transaction. In the multi-leader model, some subset from 2 to n where n is the count of all processes, declare themselves a leader for this container operation to the IOD layer. Any number of processes can participate in modifying container without regard to whether or not they are a leader. Once each leader has finished, with the assumption that any clients they may be responsible for are finished as well, the IOD layer aggregates those responses to either commit or abort the transaction.

Ultimately, with the passive detection of faults for transaction leaders, the transaction mechanism can work very well. A mostly unstated restriction that is being relaxed is that every sequential transaction on a container is considered dependent on the earlier transaction. Should one output be delayed and the subsequent five succeed, when the delayed process finally fails, all six transactions are rolled back. The thought of using this mechanism to store subsequent checkpoint outputs in the same container to both save space, but not care if one fails, cannot work in the current form. This has been acknowledged and is being relaxed requiring a new parameter to the creation of a transaction determining if it will be dependent or not.

B. DAOS Epochs

The Epoch mechanism differs from transactions. Instead of focusing on when a particular output is complete, an epoch represents incremental persisted copies of a container. To simplify the mapping between an IOD transaction and the DAOS epochs, when an IOD transaction is persisted to DAOS, the IOD transaction ID is the used as the epoch ID. The key difference is that at the DAOS layer, some transaction (epoch) IDs will not be represented since not all IOD transactions are necessarily persisted.

C. Metadata Management

Metadata management has been a perennial challenge for parallel storage systems. Eliminating metadata management as a special case and instead treating it just as data is a central design goal of the Fast Forward project. This is a hybrid approach to metadata management that is half-way between providing no inherent metadata support and having a fully integrated, but separate metadata management system.

Eliminating metadata as a core component of a file system is not new. It has been explored as part of the Light Weight File Systems project [16]. In LWFS, the metadata service is explicitly limited to a user task with the storage layer limited to data storage/retrieval, authorization, and authentication. This approach proved workable. Using this hybrid approach is less common [19] and introduces other issues.

IOD and DAOS both share a philosophy that they will have to maintain the metadata about how the physical pieces of the logical objects are striped and where they are placed. The primary metadata management is done at the DAOS layer with the IOD layer relying on the DAOS layer for all authoritative information about containers and objects. The only place where the IOD layer manages metadata for itself is to manage how the different objects are striped across the IO nodes.

1) Design Philosophy: While the metadata design is not fully defined, there are a few things that are intended. For example, there is a standard, well-known container that is the system metadata. This includes the list of all other containers. This container is treated like any other data in the system and striped as appropriate. Unfortunately, this still couples the metadata to a single object that must serialize access. If the metadata, including information about striping and other data layout operations were separated completely from the data path, more scalable throughput could be achieved. The real challenge of this is introduced by the IOD, DAOS, and VOSD layers collectively. Each of these requires some different metadata storage and the migration is transparent to the user. Supporting fully independent metadata with this model is difficult. Serious thought on how to do this effectively outside the data path should be considered for phase two.

Based on the lessons from the D^{2}T metadata service [10] construction and the prior experiments with LWFS, having a completely separate metadata service is workable. Rather than making it a bottleneck in the IO path, it is another service that users must interact with if they need those services. Users can manage everything by maintaining the metadata including the list of objects themselves. However, there are drawbacks to this 100% client-side approach.

With a client-side only approach, there is a serious risk of the metadata service and the object store getting out of sync. While having a metadata-less object storage service is desirable, the different semantics from traditional file systems requires some considerations. In this case, should these services get out of sync, three particular risks are introduced. First, a client could create a dangling entry in the metadata service that does not correspond to any objects in the object store. Second, a client could create orphaned objects that have no associated metadata entries. Third, updates to the metadata or object store service should be an atomic operation, but due to a lack of coordination, a window where the system is inconsistent appears.

Ultimately, the consistency semantics required must be determined. If a metadata service is required and it must be in sync with the object storage service, then additional work must be performed. In traditional file systems, the metadata and object storage updates are atomic. With decoupling metadata from object storage, should this atomicity still be desired, it requires both the ability for the services to participate in a task that is part of a larger atomic operation and a higher-level mechanism to manage the atomic operation.

Overall, while additional work is required to maintain a client-side only metadata service, it eliminates any potential bottlenecks related to updating metadata related to the object storage. The burden of tracking striping and other metadata that has traditionally been part of the metadata associated with the file system will have to be maintained by the object storage service. The lack of a centralized, serialized bottleneck to store that information improves concurrency.

D. Comparison to D^{2}T

The D^{2}T project [12] sought to develop an efficient approach for handling ACID-style transactions in an environment with parallel clients and multiple servers (doubly distributed). Rather than being aimed solely at data movement operations, D^{2}T seeks to address the general problem of managing any operation with multiple clients and servers. Consider the management of the analysis/visualization area, potentially similar to the IOD concept. The transaction protocol is used to help manage resizing of the resource allocation to the various analysis and visualization components. For the purposes of this discussion, D^{2}T could also be used to manage changing how IOD processes and/or nodes are used without exposing these changes to the client processes prematurely. This has been described and analyzed previously [7].

The example metadata and data storage services created as part of the D^{2}T project have no dependencies between transactions that prevent visibility should an older version be incomplete. This additional, intentional requirement by IOD offers different functionality than D^{2}T's example services. In the case of D^{2}T, the functionality is more minimal, but also avoids some of the concerns outlined below.

The second iteration of the protocol [11] fixed scalability issues and demonstrated a scalable client-side coordination model with excellent performance. The performance measured for a complex transaction with D^{2}T is illustrated in Figure 1. This performance is explored in detail in a previous paper [11].

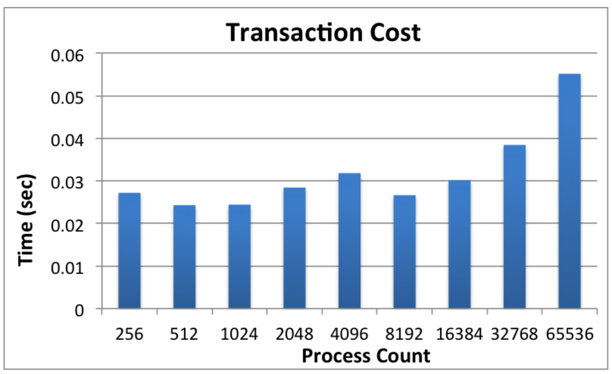

Fig. 1. Total Transaction Overhead

At a high level, both D^{2}T and the IOD transactions have the same design. In both cases, a hierarchical model is employed. In the case of D^{2}T, it is a purely client-side tree using semi-synchronous messaging. The messaging itself, in the current implementation, uses asynchronous MPI messages. The synchronous component comes from the timeout mechanism used to detect faults. It forces a level of coordination and synchronization for the protocol. For IOD, it is a server-side tree and fully asynchronous relying on the existing Lustre fault detection mechanism for failure detection. In both cases, there is a master in charge of managing the transaction and a collection of workers that aggregate into the master through second-level leaders. Beyond that, there are some significant differences. Some of the different choices made by IOD raise some possible concern.

The multi-leader model introduces the possibility of forcing a rollback of an entire transaction when a partial retry might be sufficient for success. Since the transactions are managed at a high level rather than the individual tasks, a failure in a limited distributed task can cause the entire transaction to fail. For example, consider 10 processes each have 5 tasks, but 3 of those 10 have an additional shared task to complete. If the task shared by the 3 processes fails on any of the three, the entire transaction would roll back because it is a coarse-grained success/failure. If a concept like sub-transactions at a task granularity were used, then it would be possible for the one process that failed to report just that failure. Then the transaction manager could reassign the resources for these 3 processes and try just that operation again. If it now succeeds, then the overall transaction can be marked successful only redoing the minimum amount of work required.

D^{2}T has addressed theses issues in a couple of ways. First,

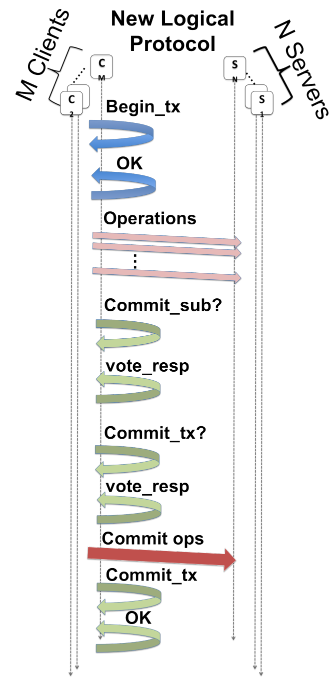

Fig. 2. Optimized Protocol

the sub-coordinators each have a list of processes from which they expect messages. Should a message be missed, it is noticed and corrective action can be taken. Second, D^{2}T has the concept of sub-transactions. The messaging requirements are illustrated in Figure 2. Sub-transactions represent finer grained operations than the entire output, D^{2}T can manage multiple writes per client by using a sub-transaction to represent the output for any item to the file (container). Because of how the sub-transactions are managed, the singleton sub-transactions, ones in which only a single process participates, must be declared before the transaction begins so that its existence can be broadcast as part of the begin transaction message. This ensures there is global knowledge that the sub-transaction is expected. That way if the coordinator (transaction leader) fails, whichever process takes over that role knows to expect a completion message for that sub-transaction or the overall transaction cannot complete. Global sub-transactions can be defined at any time since they are a global, synchronized operation broadcasting their existence. While this additional layer does introduce messaging, the overhead is quite small.

The advantages of eliminating these messages is not performance as demonstrated by the performance of D^{2}T. Instead, it offers a much less synchronous model that matches with different programming models, such as Charm++ or other task-based approaches. Since it can work for the bulk-synchronous model also, it is a more broadly applicable approach. This assumes that the observed potential issues can be addressed successfully.

IV. Broader Design

At a broader level, there are some concerns that were partially clarified through conversations with the team. Consider a shared file system across an HPC data center. The current design maintains the metadata in its own container. Since copying data from the IOD layer to the DAOS layer requires an explicit persist call, how and when synchronizing the metadata across the layers and potentially across machines occurs is undefined. Delaying synchronization until an explicit persist is called will reduce the update frequency, but delays the data visibility on other platforms. Ideally, the metadata object would need to be automatically persisted every time a container transaction is persisted to the DAOS layer.

The implication of this is that every transaction persist is double operation to account for the metadata persist. More importantly, the IOD-layer version of the metadata container may contain readable transactions that have not been persisted to the DAOS layer. How to handle this inconsistency between the two layers still needs to be explored.

A point of confusion rather than a potential design challenge is the change in definitions between the IOD layer and the DAOS layer. For the IOD layer, a container is a collection of objects. For the DAOS layer, a container is a collection of objects across a set of shards. For the IOD an object may be a shard of a global array. For DAOS, a shard can host a set of DAOS objects. Having the same names with locally correct, globally conflicting definitions serves to confuse how the system should work.

V. Conclusions

The Fast Forward Storage and IO Stack project has designed a good first pass at addressing the requirements for an extreme scale data storage mechanism. The split between the IOD layer and the DAOS layer offers a fast place for intermediate data without requiring the overhead of writing to persistent storage. The envisioned transaction mechanism, while not perfect in the current form, is another good attempt to address both failures and prevent access to incomplete or incorrect data by downstream data consumers. Integrated with the IOD functionality, this concept represents the consensus approach for what will be required.

The partial metadata management incorporated into the IOD layer and the lack of consideration for how to handle and recover from failures are oversights in the current documents. It is our understanding that these will be addressed in the next phase and we hope to help inform that effort with our experiences.

We hope that the efforts made in the D^{2}T, Lightweight File Systems, and other efforts to explore the requirements for this space, along with the analysis presented in this paper will prove useful for the next phase of the Fast Forward project.

VI. Acknowledgements

The authors would like to thank Eric Barton, John Bent, Gary Grider, and Quincey Koziol for their early review comments and discussions that clarified the details of the design, the intent of the design, and the future plans.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf. Extending i/o through high performance data services. In *Cluster Computing*, Luoisiana, LA, September 2009. IEEE International.

[2] H. Abbasi, M. Wolf, and K. Schwan. LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 341–348, Washington, DC, USA, 2007. IEEE Computer Society.

[3] E. Barton. 2014-03-21. personal communication.

[4] J. Bent. 2014-03-21. personal communication.

[5] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[6] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage challenges at los alamos national lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5, April 2012.

[7] J. Dayal, J. Cao, G. Eisenhauer, K. Schwan, M. Wolf, F. Zheng, H. Abbasi, S. Klasky, N. Podhorszki, and J. Lofstead. I/o containers: Managing the data analytics and visualization pipelines of high end codes. In *In Proceedings of International Workshop on High Performance Data Intensive Computing (HPDIC 2013) held in conjunction with IPDPS 2013*, Boston, MA, 2013. Best Paper Award.

[8] Fastforward storage and i/o stack design documents. Intel FastForward Wiki, February 2014. https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and +IO+Program+Documents.

[9] Q. Koziol. 2014-03-21. personal communication.

[10] J. Lofstead and J. Dayal. Transactional parallel metadata services for application workdflows. In *In Proceedings of High Performance Computing Meets Databases at Supercomputing*, 2012.

[11] J. Lofstead, J. Dayal, I. Jimenez, and C. Maltzahn. Efficient transactions for parallel data movement. In *The Petascale Data Storage Workshop at Supercomputing*, Denver, CO, November 2013.

[12] J. Lofstead, J. Dayal, K. Schwan, and R. Oldfield. D2t: Doubly distributed transactions for high performance and distributed computing. In *IEEE Cluster Conference*, Beijing, China, September 2012.

[13] J. Lofstead, R. Oldfiend, T. Kordenbrock, and C. Reiss. Extending scalability of collective io through nessie and staging. In *The Petascale Data Storage Workshop at Supercomputing*, Seattle, WA, November 2011.

[14] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[15] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest checkpoint storage system for large supercomputers. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pages 1 –5, nov. 2008.

[16] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[17] D. Skourtis, D. Achlioptas, C. Maltzahn, and S. Brandt. High performance & low latency in solid-state drives through redundancy. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, page 6:16:9, New York, NY, USA, 2013. ACM.

[18] The HDF Group. Hierarchical data format version 5, 2000-2014. http://www.hdfgroup.org/HDF5.

[19] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, Nov. 2006.

[20] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - preparatory data analytics on Peta-Scale machines. In *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.