Sunil Template

Contents

Chapter 1

The Convergence of Enterprise, Internet Scale, and High Performance Computing Storage Infrastructures

Jay Lofstead

Sandia National Laboratories

Eric Barton

Intel

Matthew Curry

Sandia National Laboratories

Carlos Maltzahn

University of California, Santa Cruz

Robert Ross

Argonne National Laboratory

Craig Ulmer

Sandia National Laboratories

Abstract

Large scale storage infrastructures have been significantly impacted by the growth in data analytics applications. High Performance Computing storage infrastructures, once the extreme end of the storage scale spectrum, must now adapt to technologies optimized for large scale data analytics applications. Hardware changes, such as storage class memory, are also affecting how the exascale storage stack will be constructed. We examine use cases, trends, convergent technologies, and new opportunities generated by this technology blending.

1.1 Introduction

HPC infrastructures have grown around the requirement to handle large, decomposed data structures for parallel computation. Single data objects may be hundreds of terabytes spread across an entire machine. Parallel storage systems have grown trying to address performance and storage requirements while maintaining backwards compatibility with the standard POSIX interface and semantics. Unfortunately, this is proving increasingly difficult, as the POSIX specification was not designed to efficiently support parallel storage [5].

Big data and internet-scale applications, on the other hand, focus on searching through immense volumes of small, loosely associated items looking for patterns or correlations that may lead to insights. Some science applications, such as genomics, have a workload pattern similar to these big data applications. Parallel file systems are not well suited to these workloads because the broad, relatively continuous read demand of independent items does not benefit from the coordination overheads of parallel file systems. Instead, distributing loosely coordinated storage across the compute infrastructure makes more sense. With pressures to effectively leverage a single platform for these disparate workloads and the shifting storage market, new considerations for how to design storage resources for extreme scale compute systems must be made.

Traditionally, the HPC market has focused on supporting coherent and consistent output methods from parallel sources to parallel targets. This requirement is driven from validating that the output of a single item is complete

and correct. Largely, the workload is write-intensive during the expensive, at scale computation process with a read-intensive phase lasting months on cheaper machines or at small scale with low priority. File systems like the dominant Lustre [2] and GPFS [6] systems have been carefully optimized to address these workloads.

The big data market has opposite priorities. The big computation phase requires reading large quantities of data for processing at scale. The output from this process can be handled at much smaller scale later and is orders of magnitude smaller. Given the read-dominant focus, the overhead inherent in coherent and consistent storage for write intensive workloads is both unnecessary and a heavy cost. Instead, systems like HDFS [8] dominate. These work by storing files that will be read for processing throughout the compute area, including the assumption that storage failures are common, prompting replication.

The output from initial stream or file processing for the big data workloads use distributed object-based storage technology. It offers independent, uncoordinated data access with a simplistic key search space for subsequent analysis. The profit potential for this market has caused an explosion in specialized products aimed at accelerating this processing style. For small enough data sets, in memory object stores like memcached and Radius dominate. For larger data sets, approaches like Google's Big Table are accomplishing the same function. There are also hardware products targeting this market segment, such as Kinetic [7], offering a native object interface for the devices connected directly to a network.

Adding complexity to this storage environment is the relentless performance improvements and cost reductions for solid state storage, like NAND-based flash memory. These devices have already rendered 15,000 RPM disk drives obsolete. The 10,000 RPM disk drives will not survive for more than a few more years. New disk technology like shingled drives [10] offer a path for disks to survive longer. The enormous capacities for read intensive, write infrequently workloads is very attractive for many communities. For example, storing images created sequentially for later read-intensive processing can yield a better cost/performance balance.

This chapter investigates these new technologies and how they affect extreme scale computing. We evaluate how the HPC environment can and must adapt to this new storage environment to address future computing needs and to take advantage of the different kinds of technology being developed. We will also consider the planned reintegration of large scale computing from the split of big data applications from simulation-based computing with both the necessary and forced integration of these large, expensive platforms for multi-use.

1.2 Existing File Systems

HDFS developed to support the Hadoop implementation of the MapReduce system. It offers a distributed, replicated file store optimized to support the MapReduce processing configuration. Ceph also addresses this distributed computing infrastructure, but with a different emphasis. It seeks to offer scale out performance for objects across a storage infrastructure. However, Ceph does not offer the ability to scale up: Each object must fit within a single storage device. With the rise of cloud systems using object-based storage, such as Amazon's S3, the interaction style offered by Ceph has been adopted for similar workloads. Ceph has features to handle storage devices failing and new ones joining a running system without interruption. Ceph offers a complete file system including metadata management support as well as an object system for users. GlusterFS focuses on providing a network attached storage interface to storage distributed throughout a cluster. Rather than providing metadata services, GlusterFS relies on the underlying storage file system for most basic capabilities, such as security.

Parallel file systems are optimized to support large files that must be spread across multiple devices. For example, a 100 TB file cannot fit on any current storage device and cannot be stored with any performance. Parallel file systems solve this problem by using multiple devices spread across multiple servers together as if they are a single device. Data is striped across devices, all of which can be written to or read from simultaneously. This parallel access offers aggregate performance enabling manipulating very large files with reasonable performance. Because of these characteristics, parallel file systems are deployed on most simulation intensive large scale compute systems to handle the large single object output characteristic of these applications. Lustre is arguably the most popular parallel file system appearing on a majority of the Top500 machines. IBM's GPFS is increasing in popularity as optimizations focusing on addressing big data workloads are incorporated. PVFS offers a rethinking of some of Lustre's earlier limitations to give better scalability characteristics.

Discussion
The different optimizations distributed vs. parallel file systems offer are at the cost of supporting the other kind of workload. As was mentioned above, parallel file systems aim to support very large objects and aggregate simultaneous writing and reading for a single object across the array. For workloads consisting of entirely small objects, this functionality and overhead is a cost. Similarly, the inability of the distributed file systems to handle arbitrarily large objects and massive parallel simultaneous access to a single object make them unsuitable for simulation workloads.

For both workloads, a more flexible object-based interface are being considered. This is discussed more below.

1.3 Object-Based Stores

The earliest object store is probably the Wisconsin Storage System [3] published in 1985. It offered a general storage infrastructure for both databases and file systems. Many current systems were built using a similar infrastructure, such as Lustre [2], GPFS [6], and Panasas [9]. The general idea is to offer s a standardized way to address an arbitrary storage space with a key-based access. These object storage systems assume that some sort of structure is imposed on top to track what objects correspond to which user items.

While object storage may have been used behind the scenes for years, raw object storage exposed to the end-user programmer did not into vogue until the big data era arrived. System developers for big data processing realized that the overhead imposed by the object management forced serialized or at least coordinated access. By shifting the mapping load to the end-user programmer level and using the object storage layer directly, greater perceived performance can be achieved. In many cases, by stripping down the requirements to the absolute minimum required semantics for a particular application, actual performance gains are achieved. The explosion of specialized storage systems like HDFS and GFS represent this model. Key for this model achieving performance is the ability of each process to work independently without any required consistency or coherence with neighbor processes potentially working on part of the same data set. The dominant object stores are systems like Memcached [4]. This stands in stark contrast to how supercomputing applications generally operate.

The supercomputing domain maintained the consistency requirements due to the bulk synchronous parallel processing. Instead, parallel, that is coordinated, file systems were embraced. The challenge today is that parallel file systems are having difficulty effectively scaling to handle the IO demands.

Here we want to talk about how object-based key-value stores are used for big data applications summarizing the specific features that identify this market segment.

1.3.1 HPC Oriented Object Stores

Parallel file systems inherently have an object-like layer beneath the surface. The requirement to spread a single file across multiple devices for capacity reasons alone prompts this approach. The actual implementation may vary, such as using individual files within a local file system, each representing part of parallel file. Popular examples include Lustre [2] and GPFS [6].

In some cases, directly using a key-value store for HPC applications is being considered [11]. The next generation Lustre project is also considering a key-value infrastructure [1] to address performance challenges.

The real challenge with key-value stores for HPC applications is the metadata management. All of these projects have taken a similar step to the big data application is that the applications are required to manage the object list to determine what data is stored in which object.

1.4 Next Generation HPC Storage Systems

Here we want to talk about, at a proposal sort of level, what we think needs to be done.

Talk about the disconnect between metadata and storage and the complications it introduces and some ideas on what we plan to do about it.

Talk about the major efforts

1.4.1 Lustre/DAOS

The US Department of Energy is using a "Fast Forward" series of projects to kickstart needed developments in the vendor space. One such project is the Storage and I/O (FFSIO). The basis for this is extending the existing Lustre infrastructure to offer an object interface, incorporate support for burst buffer-like technoligies, and demonstrate use through a client I/O library. The first phase completed in 2014 and the second phase started in late 2015.

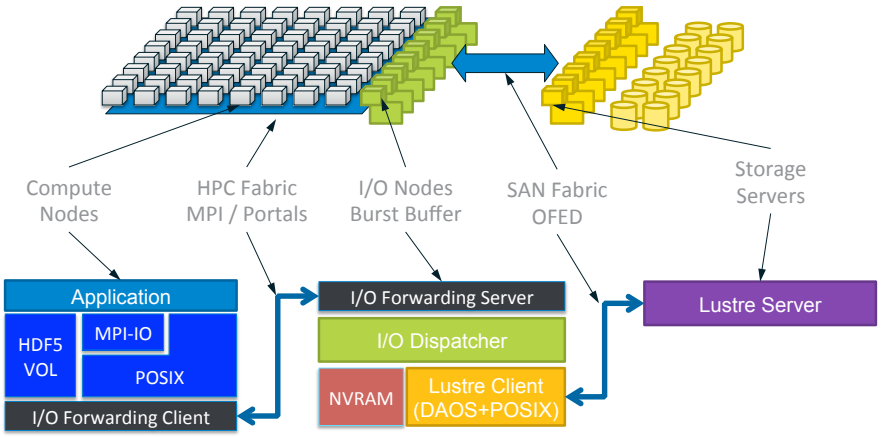

FIGURE 1.1: FFSIO Architecture and Component Mapping

Overall, the system design (see Figure 1.1) consists of a friendly end-user

API that manipulates either the I/O Dispatcher layer or the DAOS (Distributed Asynchronous Object Storage) layer.

The first phase design focuses on a few themes. First, transactions are integral to offering decoupled performance. Second, storing many individual objects rather than logically organized fewer objects is prefered. And third, hide additional API complexity behind a friendly end-user API.

1.4.1.1 Transactions

The system jitter and other causes for processes to fade out of sync affect overall I/O performance as well. While transactions are typically seen as a synchronous operation that blocks while all participants manage state, FFSIO takes a passive approach and manages transactions asynchronously. The key idea is to keep writes in a log-style copy-on-write structure and annotate each write with which transaction it is part of. An overall transaction mapping knows how many writes are expected and can decide when a transaction is complete or not. By monitoring the system status, it can passively detect many failures abort the affected transactions. The potential overheads to this approach are refining the idea, but the core concept shall remain the same.

Transactions are manifest in two different ways. First, the end-user API layer can opt to have the FFSIO system manage the transactions by that layer tracking the status of all clients directly. Second, transactions can be managed by the end-user layer themselves with a single process indicating the transaction state to the FFSIO layer. The idea for the former approach is that it offers transactional functionality without undue end user code changes. At the IOD and DAOS layer, transactions are handled slightly differently.

The IOD layer is intended as a staging area for data that may or may not need to be written to persistent storage. This is one implementation of the burst buffer concept. The key idea is that the transactions in this layer may or may not ever be pushed down to the DAOS layer. Instead, once the data set is processed, only the analyzed results will be saved eliminating the need to write large, temporary data sets to the slower, but much larger capacity storage array. However, these transactions are connected to those at the DAOS layer.

The DAOS layer is intended to function as a general object store that would replace an existing data center wide shared parallel file system. Anything written to this layer is expected to have longer lifetimes and potentially be accessible from other platforms. Since the transactions work a bit differently, in particular some sequential transactions may be missing, the name is changed to epochs instead. The slightly different idea for an epoch is to represent some important saved state. Since it is likely a revision of an existing version, differencing mechanisms are tested to determine how to efficiently store both versions with less space and minimal reconstruction overhead. When small changes are persisted, the space savings can be enormous.

Other, more general approaches to transactions have been developed [**?**, **?**],

but they have higher end-user API complexity to support the more general use. Other than the synchronous requirement for failure detection, the performance is excellent.

For future transactional use in the storage hierarchy, some mixture of the three approaches is likely. The passive transactions will be used for backwards compatibility. The end-user, single process managed transactions will work well for storage system interactions. The D2T approach could be used in isolation or to support the end-user, single process managed transactions.

1.4.1.2 Objects

Potentially extreme overheads for writing logically contiguously formatted data is well documented. For example, ParColl [**?**] showed as much as 90% overhead on as few as 512 processes. For the Six Degrees of Scientific Data [**?**] work, the authors were unable to perform some performance comparisons because a single data output could not be written in the maximum normal job time. These sorts of impacts are prompting rethinking the logically contiguous format favored by popular APIs like HDF5 and PnetCDF. The perceived advantages when these formats were selected have not proven to scale to very large, 3-D domain decompositions in particular. DAOS, offers the object storage infrastructure while the HDF5 API at the end-user level is maintained almost exactly. The only end-user API changes are related to incorporating transactions. This shift almost exactly maintains backwards compatibility while taking advantage of more scalable infrastructures.

1.4.1.3 API Complexity

One of the complexities of the FFSIO stack is addressing various scale platforms. For the largest platforms, including an IOD layer makes sense. The particular architecture may vary, but that kind of technology will be important to reduce compute throughput due to waiting for I/O to complete. For smaller platforms, either the IOD layer is deemed unnecessary or it may be an extra expense that offers little advantage. In this case, omitting this component both simplifies the system and offers more direct performance.

The challenge with this approach is that the end-user level must address both the IOD and DAOS APIs directly and be able to choose for fully portable code. The advantages of being able to effectively scale down were maintained by using a higher-level end-user API, like HDF5. For such a professionally produced library, offering a second version is not a major undertaking. Further, it enhances the HDF5 value by offering greater platform scalability.

1.4.1.4 Discussion

Overall, the shift from a traditional parallel file system to something like the FFSIO system will be a big step forward towards supporting more big

data style hardware directly. While the design is not perfect, the second phase is addressing most, if not all, of the detected shortcomings.

1.4.2 Kelpie/Data Warehousing

Kelpie is a distributed, in-memory object store from Sandia National Laboratories that serves as a building block in high-performance computing for implementing custom, data-management services. The fundamental goal of Kelpie is to provide a way for users to decompose their complex datasets into data objects that the library can move between nodes in a safe but efficient manner. Kelpie provides simple abstractions for dealing with distributed data, and utilizes the Nessie communication library to orchestrate how data migrates between nodes. Nessie provides (1) a low-level RDMA layer that has been ported to different HPC fabrics and (2) an RPC layer that enables users to invoke function calls and initiate RDMA transfers on remote nodes. Kelpie uses the former to make an application's data objects accessible by the network interface, and the latter to cooridinate data handoffs between nodes.

Kelpie manages data objects for applications, where a data object is a simply a contiguous block of application data that is labeled with a globally-unique key. The contiguous constraint is necessary because Kelpie registers the memory with the network interface, which in turn allows the hardware to RDMA the data without involving the kernel in virtual to physical address translation. The key used to label an object has three components: an application-specific identifier and a two-dimensional user label. The application-specific identifier enables higher-level services to house different datasets in Kelpie with isolation. Users are not required to use the second dimension of the user label portion of the key. However, the second dimension is often useful for grouping related items together in the store. For example when storing complex mesh datasets in Kelpie, the first dimension of the key (or row) may be used to describe a particular region of a mesh. The second dimension of the key can therefore be used to organize different variables associated with the region (e.g., node locations, pressure, or temperature). This approach allows each variable's data to be stored in its own, independent block of contiguous memory, and provides an opportunity for a user to easily downselect the items they need when retrieving data from Kelpie.

Kelpie nodes are equipped with a Local Key/Value (LKV) structure for managing data objects that are available in the local node. The LKV performs three important tasks. First the LKV provides a means of tracking data objects that are currently in transit and protect the system from deallocating an object before remote nodes have finished transferring it. Second, the LKV structure provides a means for data to be staged at a remote node with only trivial involvement from the destination. For example, an application may push multiple data objects to a node in anticipation of work that will be scheduled on the remote node at a future time. Finally, the LKV structure provides a location for applications to store callbacks to execute when data

does arrive at a node. These callbacks make the data store more active and are fundamental to applications that are highly asynchronous or event driven.

In order to address scalability concerns, data management services for HPC typically decompose their work in a hierarchical manner that maps owenership of different portions of the dataset to different groups of nodes that are close in proximity. Kelpie provides the ability to assemble multiple teams of nodes together through a resource management interface. This interface allows users to create and reference a team of nodes together to function as a single data resource that employs a standard data API. A resource has three components: a path-like name that allows a resource to be globally referenced and located by the runtime, a list of physical nodes that implement the resource, and client-side software that defines that policy for how data is managed in the resource. Kelpie provides common implementations of resources, but is easily extended with user-defined modules. A distributed hash table (DHT) is an example of a commonly-used resource in Kelpie. A DHT is composed of N Kelpie servers where data is distributed by using a hash of the first dimension of the key to specify which node should store the object. The DHT resource client software simply maintains the list of servers and then references the proper destination when a user performs put or get operations. Users with reliability concerns have made similar resources just by creating new client software that stores a data object to the hashed server, and a replicated copy to its neighbor. While resources may contain API extensions to support higher levels of functionality, they all support basic communication primitives that enable users to swap one type of resource in for another in most cases.

An example of how Kelpie is being utilized to support data management for higher-level applications can be found in Sandia's DARMA project. DARMA is developing an asynchronous, many-task (AMT) approach to computing that will help codes scale to next-generation platforms. AMT codes specify their execution in the form of a large, directed acyclic graph of tasks (or task DAG) that can be scheduled by a runtime on distributed resources. The data objects consumed and produced by each task form a dependency graph that dictate when and where work can be scheduled. By utilizing Kelpie as the mechanism by which data objects are migrated between different nodes and resources in the system, DARMA can focus on the complex job of making policy decisions about how the work should be orchestrated in the platform.

1.4.2.1 Discussion

Kelpie is clearly taking a different approach from the FFSIO project. By pushing the key-value structure all the way up to the application, it requires a whole new interaction between the applications and the storage infrastructure, but at a potential advantage of much richer, higher performance. For example, by integrating with the AMT management layer, Kelpie can predictively move data as necessary or even guide the AMT manaagement layer to place computation in a different location to avoid the data movement. Also, by

offering object level access beyond what an application would normally write to storage, potentially new data access routines for richer analysis could be integrated without requiring any application changes. Simply by registering all data through Kelpie for use in the AMT management layer, other users could take advantage of the metadata and select data for additional analysis without worrying about explicit staging or writing to some persistent store.

1.4.3 Hybrid Models

SSIO project from ORNL/Sandia

1.5 Conclusions

This is our overall view on things

Acknowledgements

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.

Bibliography

[1] Eric Barton. Lustre*-fast forward to exascale. *Lustre User Group Summit*, 2013.

[2] Peter J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, November 2002. http://www.lustre.org/docs/lustre.pdf.

[3] H-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the wisconsin storage system. *Software: Practice and Experience*, 15(10):943–962, 1985.

[4] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[5] Dries Kimpe and Robert Ross. Storage models: Past, present, and future. In Quincey Koziol and Prabhat, editors, *High Performance Parallel I/O*, chapter 30, pages 335–345. Chapman & Hall/CRC, 2014.

[6] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX FAST '02Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[7] Segate. The segate kinetic open storage vision. http://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/, 2014.

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[9] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In Mary Baker and Erik Riedel, editors, *Proceedings of the USENIX FAST'08Conference on File and Storage Technologies*, pages 17–33. USENIX, February 2008.

[10] Roger Wood, Mason Williams, Aleksandar Kavcic, and Jim Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *Magnetics, IEEE Transactions on*, 45(2):917–923, 2009.

[11] Yanlong Yin, Antonios Kougkas, Kun Feng, Hassan Eslami, Yin Lu, Xian-He Sun, Rajeev Thakur, and William Gropp. Rethinking key-value store for parallel i/o optimization. In *Data Intensive Scalable Computing Systems (DISCS), 2014 International Workshop on*, pages 33–40. IEEE, 2014.