Chapter 14

PLFS: Software-Defined Storage for HPC

John Bent
EMC

14.1 Motivation

Just as virtualization has provided massive flexibility for computation across diverse processor hardware, software-defined storage provides that flexibility for I/O workloads across diverse storage hardware. The Parallel Log-structured File System (PLFS) is one such software-defined storage platform. Originally designed as middleware designed to solve massively concurrent checkpointing, PLFS has become a powerful reminder of how not all workloads are well suited to all storage systems. PLFS is also a compelling example of how a layer of software indirection can change the base abilities of existing storage systems. PLFS has since extended its functionality to address additional storage challenges thereby growing from single purpose middleware into a more general software-defined storage platform.

The main benefit of PLFS is that a system adminstrator can buy a single storage system, configure it just once, and then use PLFS to allow a variety of workloads to use that single storage system. The basic mechanism of PLFS is an interception and transformation of unstructured application I/O into well-structured I/O better suited to the underlying storage system. PLFS has three main modes of operation (shared, flat, and small) essentially comparable to mount time options and several interesting use cases for each mode.

14.2 Design/Architecture

PLFS is mainly designed to run as middleware on the compute nodes themselves. It can run in the user space of applications using MPI-IO and a patched MPI library, or in the user space of applications which are ported to link directly into the PLFS API. The PLFS API closely mirrors the standard POSIX API; thus porting several applications and synthetic benchmarks has been straight-forward. PLFS is also available as a FUSE file system [1] for unmodified applications which do not use MPI-IO. Since the FUSE approach can incur high overhead, there is also an `LD_PRELOAD` interface which brings PLFS into the user space of unmodified applications [8].

There are three main modes of PLFS which are set in a PLFS configuration file and defined on a per path basis. The options for each path define the path that the user will use (i.e., `/mnt/plfs/shared_file`), the mode of operation, and the underlying storage system(s) that PLFS will use for the actual storage of the user data as well as its own metadata. Typically, the underlying storage system is a globally visible storage system, and the PLFS configuration file is shared across a set of compute nodes such that each compute node can write to the same PLFS file(s) and each compute node can read PLFS files written from a different compute node.

The three main configurations of PLFS are *shared file, small file, and flat file*, each of which is intended for different application I/O workloads. Additionally, there is a *burst buffer* configuration (which currently works only in shared file mode) to transparently gain performance benefits from a smaller, faster storage tier such as flash memory. All three modes support the ability to use PLFS as an *umbrella* file system that can distribute workloads across multiple underlying storage systems to aggregate their bandwidth and utilize all available metadata servers. Finally, there is support in PLFS to run with all three modes on top of *cloud* file systems such as Hadoop.

14.2.1 PLFS Shared File Mode

Shared file mode is the original PLFS configuration [3] and is designed for highly concurrent writes to a shared file, such as a checkpoint file which is simultaneously written by all processes within a large parallel application. The architecture of PLFS shared file mode is shown in Figure 14.1. Note that the figure shows the PLFS layer as a separate layer; this is accurate from the perspective of the application but in fact the PLFS software runs on each compute node. This mode was motivated by the well-known observation that

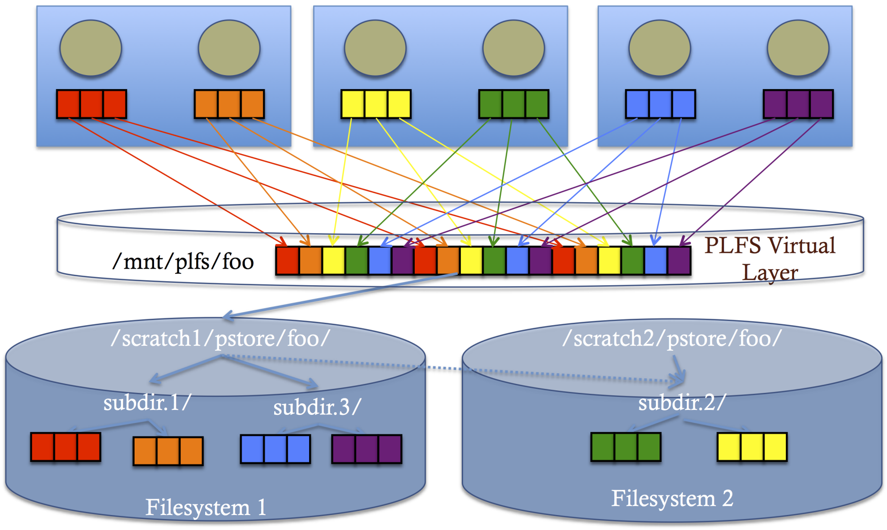

FIGURE 14.1: PLFS shared file mode transforms multiple writes to a shared file into streams of data sent to multiple subfiles on the underlying storage system(s). Not shown is the internal PLFS metadata used to reconstruct the file. [Image courtesy of John Bent (EMC).]

many applications naturally have partitions of a large distributed data structure which are poorly matched to the block alignment of many storage systems and therefore lose performance to various locks and serialization bottlenecks inherent in parallel file systems.

By decoupling the concurrent writes, PLFS sends data streams to the underlying storage systems, which avoid these locks and bottlenecks. The basic mechanism is that PLFS first creates a PLFS *container* and then stores all the individual subfiles into this container as well as the metadata necessary to recreate the logical file. In function, the container is no different than how *inodes* are used in almost as file systems since the Berkeley Fast File System [7]. When the user requests data from the file, PLFS consults the metadata within the container to resolve which subfile(s) contain the requested data and then reads from the subfile(s) to return the data to the reading application.

Note that at no point is the application aware of this transformation: all operations on the shared file work functionally, exactly the same as if PLFS was not present. One concern in PLFS however is that the amount of PLFS metadata can grow to challenging sizes; this concern is addressed by discovering hidden structure within seemingly unstructured I/O [6].

14.2.2 PLFS Flat File Mode

PLFS flat file mode was subsequently added to reduce metadata contention when many processes concurrently create files within a single directory. The basic motivation is that when multiple processes modify a shared object concurrently, they often incur locks and serialization bottlenecks in many storage systems. In the PLFS shared file mode, the shared object is a file and the concurrent modifications are writes, whereas in the PLFS flat file mode is for the case in which the shared object is a directory and the concurrent modifications are file creations. Just as PLFS shared file mode creates a virtual container to store the subfiles, the PLFS flat file mode creates a virtual directory from a set of subdirectories on the underlying storage system(s). When a user creates a file, PLFS hashes the filename to determine into which subdirectory to store that file. Reads and queries of files share the hash function so that they can quickly find requested files from the set of subdirectories. The PLFS flat file mode will achieve maximum performance improvements when it is configured to use multiple storage systems, and therefore multiple metadata servers to create its set of subdirectories.

In this way, when multiple files are concurrently created with a PLFS directory, PLFS spreads the create workload across multiple metadata servers. This is an important feature of the software-defined storage approach. It is easy for a system administrator to buy multiple storage systems but difficult to allow unmodified applications to spread their workload across them. With PLFS, it is a simple matter of modifying the PLFS configuration file to include all of the available storage systems. PLFS flat file mode is described in more detail in by Bent et al. [4].

14.2.3 PLFS Small File Mode

The third mode of PLFS is the small file mode, the architecture of which is shown in Figure 14.2. This mode is designed for the workload in which individual processes want to create a large number of small files in a short period of time. This workload is challenging for storage systems, which must allocate some amount of resource for each file. For a stream of many small files, this resource allocation quickly becomes a bottleneck. PLFS small file mode, like the PLFS shared file mode, creates a PLFS container. However, PLFS small file mode creates a PLFS small file container for each PLFS directory, whereas the PLFS shared file mode creates a PLFS shared file container for each PLFS file.

When an application creates a set of small files, PLFS small file mode will aggregate all of the data for each small file into a single large file stored on the underlying storage system. It will also create a metadata file to store the necessary information to allow finding small files and their data within the single large aggregated file.

This mode transforms the performance extracted from the underlying file

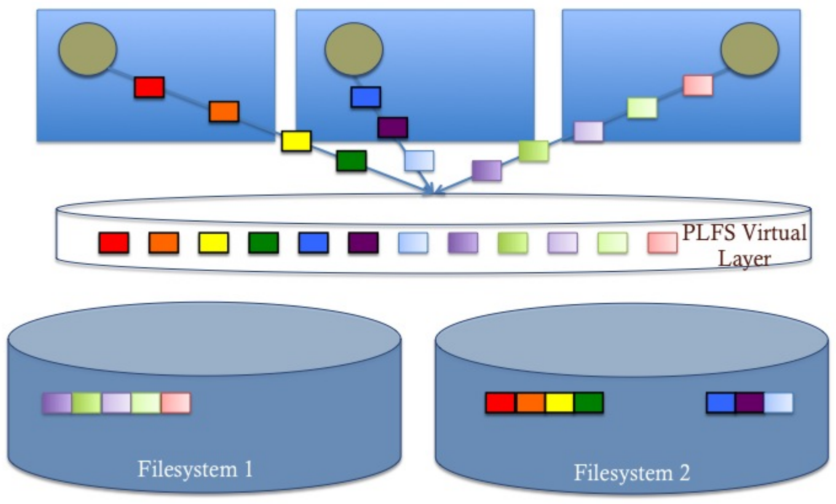

FIGURE 14.2: PLFS small file mode transforms a stream of small file creates from an individual process into two large streaming files sent to the underlying file system (one for user data and one for PLFS metadata). Note how PLFS can also aggregate the performance of multiple file systems; flat-file mode is the same except that PLFS stores each logical file as its own physical file instead of as a chunk within a single physical file as shown here. Flat-file mode therefore also does not need another file for PLFS metadata since the mapping between logical and physical is purely algorithmic. [Image courtesy of John Bent (EMC).]

system to match its maximum bandwidth for streaming data instead of matching its maximum file creation rate. The amount of performance gained is dependent on the size of the individual files.

14.3 Deployment, Usage, and Applications

PLFS is currently installed on most of the classified and unclassified supercomputers at LANL. It has also been directly ported into several LANL applications. Additionally, the PLFS shared file mode has been used in two use-cases not originally envisioned when PLFS was first developed: for burst buffers and for enabling parallel I/O to file systems which do not natively support parallel I/O.

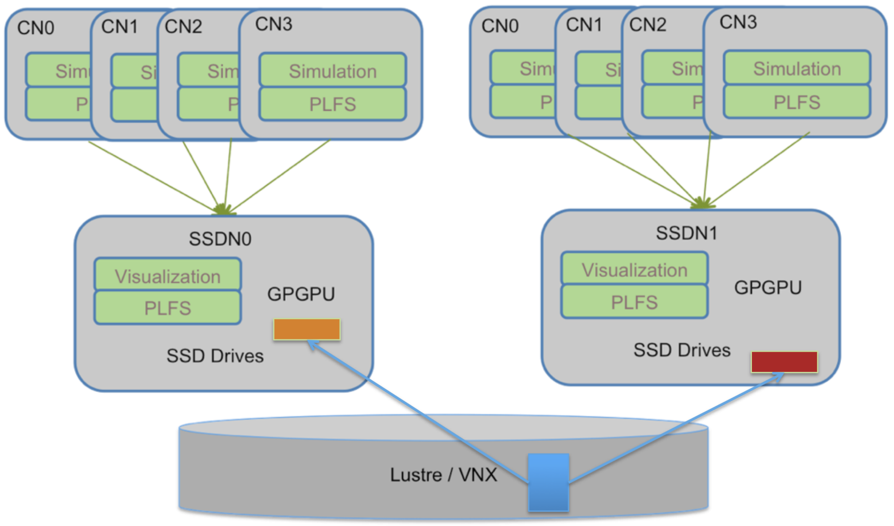

FIGURE 14.3: By using the PLFS middleware layer, the illusion of a single file is preserved in a manner completely transparent to the application and the user. Physically however, PLFS transforms the I/O to leverage both the global visibility of the parallel file system to store its metadata as well as the faster performance of the storage in the burst buffers to store user data. Later, the user data is migrated to the parallel file system. This figure shows an example burst buffer architecture where a simulation application sends simulation data from compute nodes to burst buffers nodes which have been augmented with GPU's to allow in-transit analysis by a visualization program (see Chapter 23). The data will be later migrated to a Lustre file system running on VNX storage. [Image courtesy of John Bent (EMC).]

14.3.1 Burst Buffers

As HPC transistions from petascale into exascale, the economics of storage media dictate a new storage architecture. HPC users have two basic checkpointing requirements from their storage systems: they require a minimum bandwidth for checkpointing to enable a sufficient utilization of the supercomputer and they require a minimum capacity to store a sufficient amount of checkpoint data. In the petascale, disk-based storage systems provisioned for the capacity requirement met the bandwidth requirement as well. However, as the improvements in disk capacity have outpaced the improvements in disk bandwidth, disk-based storage systems for exascale HPC will need to be provisioned for bandwidth. Unfortunately, purchasing disks for bandwidth is extremely economically inefficient. Luckily, emerging media such as NVRAM and flash storage allow an economically efficient way to purchase bandwidth, but they are not a solution for capacity. Therefore, exascale will require a new storage architecture which has become known as a burst buffer storage architecture. This name reflects its intended usage for checkpointing in which checkpoint data can be quickly saved to a small, fast flash-based tier (i.e., the burst buffer); while the computation resumes, the checkpoint data can be more slowly transferred to the larger, slower disk-based storage system.

The PLFS containers used within the PLFS shared file mode have proven to be very effective in enabling burst buffer storage systems [2]. By storing the metadata in the container, PLFS can store (and retrieve) the data itself very quickly into subfiles in a burst buffer tier. Later, the data is transferred to the disk tier and the PLFS metadata is correspondingly modified. The advantage of this approach is that the user only sees a single namespace. PLFS transparently knows whether the data is available in the flash and/or disk tier and will retrieve it appropriately. This allows unmodified applications to benefit from the performance enhancements of the burst buffer architecture. The architecture of PLFS running in burst buffer mode is shown in Figure 14.3.

14.3.2 Cloud File Systems for HPC

PLFS can also be used to enable HPC parallel workloads using storage that wasn't designed for parallel I/O. For example, early versions of the Hadoop file system, HDFS, do not allow multiple writers to a single file nor do they allow appending data to a previously created file. By layering PLFS above these file systems, the user can use them for parallel I/O since PLFS transparently transforms the shared file writing into individual processes writing to individual subfiles within the PLFS container [5].

14.4 Conclusion

Through a variety of configurations and use cases, PLFS is a dramatic example of the power of software-defined storage. An underlying storage system can be configured to work well with well-arranged streams of data. The PLFS software is then layered about that storage system and can be configured to export it for a variety of different workloads, such as shared file, small file, and flat file for better metadata load balancing. Additionally, PLFS can enable parallel I/O using storage systems not defined for parallel I/O. Finally, PLFS can create a single virtual file system from a collection of multiple storage systems both for bandwidth aggregation as well as metadata distribution. PLFS development continues today to expand PLFS functionality for an exascale future in which POSIX is finally replaced with a more parallel amenable interface using well-defined concurrency abstractions, and transactional consistency across sets of related objects.

Bibliography

[1] FUSE: File System in Userspace. `http://fuse.sourceforge.net/`.

[2] John Bent, Sorin Faibish, James Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon Woodring. Jitter-Free Co-Processing on a Prototype Exascale Storage Stack. In *28th IEEE Symposium on Massive Storage Systems and Technologies*, MSST, 2012.

[3] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint File System for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[4] John Bent, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. Storage Challenges at Los Alamos National Lab. In *28th IEEE Symposium on Massive Storage Systems and Technologies*, MSST, 2012.

[5] Chuck Cranor, Milo Polte, and Garth Gibson. HPC Computation on Hadoop Storage with PLFS. Technical Report CMU-PDL-12-115, Parallel Data Lab, Carnegie Mellon University, November 2012.

[6] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/O Acceleration with Pattern Detection. In *ACM Symposium on High-Performance Parallel and Distributed Computing*, HPDC 13, New York, NY, June 2013.

[7] Marshall Kirk Mckusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems*, 2:181–197, 1984.

[8] S. A. Wright, S.D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S.A. Jarvis. LDPLFS: Improving I/O Performance without Application Modification. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1352–1359, 2012.