Structuring PLFS for Extensibility

Chuck Cranor
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
chuck@ece.cmu.edu

Milo Polte
WibiData, Inc.
375 Alabama Street
San Francisco CA, 94110
milo@wibidata.com

Garth Gibson
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
garth@cs.cmu.edu

ABSTRACT

The Parallel Log Structured Filesystem (PLFS) [5] was designed to transparently transform highly concurrent, massive high-performance computing (HPC) N-to-1 checkpoint workloads into N-to-N workloads to avoid single-file performance bottlenecks in typical HPC distributed filesystems. PLFS has produced speedups of 2-150X for N-1 workloads at Los Alamos National Lab. Having successfully improved N-1 performance, we have restructured PLFS for extensibility so that it can be applied to more workloads and storage systems. In this paper we describe PLFS' evolution from a single-purpose log-structured middleware filesystem into a more general platform for transparently translating application I/O patterns. As an example of this extensibility, we show how PLFS can now be used to enable HPC applications to perform N-1 checkpoints on an HDFS-based cloud storage system.

1. INTRODUCTION

Demanding High Performance Computing (HPC) applications are typically large physical simulations that compute for a long time using a large number of compute nodes and the Message Passing Interface (MPI) subsystem [20]. MPI also provides group communication synchronization features such as barriers.

Typically HPC applications protect themselves from failure by periodically pausing and concurrently writing their state to a checkpoint file in a POSIX-based distributed filesystem (DFS) such as Lustre [26, 31], PanFS [21, 29], PVFS [17, 16, 24] or GPFS [25]. Checkpoint I/O can be particularity challenging when all processes in the parallel application write to the same checkpoint file at the same time, an access pattern known as N-1 writing [5, 23, 6].

An increasing number of clusters are being configured for data analytics using the Apache Hadoop open source version of the Google Internet services tool suite (Google File System, BigTable, etc.) [10, 8]. Because HPC and Internet services analytics are both big data and big compute applications families, it is desirable to be able to mix and match applications on cluster infrastructure. Prior research has demonstrated that Hadoop applications can run efficiently on HPC cluster infrastructure [2]. However, cloud storage systems such as the Hadoop Distributed Filesystem (HDFS) [27] are not POSIX-based and do not support multiple concurrent writers to a file. In order to enable the convergence of HPC and Cloud computing on the same platform, we set out to provide a way to enable HPC applications to checkpoint their data to a Cloud filesystem, even if all processes write to the same file.

In this paper we describe how the Parallel Log Structured Filesystem (PLFS) was restructured to allow extension to different backend storage, such as HDFS, and to solve different workload problems. Our extensible PLFS, for example, provides HPC applications with the ability to concurrently write from multiple compute nodes into a single file stored in HDFS.

2. PLFS

PLFS [5, 23, 7, 6] is an interposing filesystem that sits between HPC applications and one or more backing filesystems. PLFS has no persistent storage itself. PLFS transparently translates application-level I/O access pattern so that they perform well with modern DFSs. Neither HPC applications nor the backing filesystems need to be aware of or modified for PLFS to be used.

PLFS was developed because HPC parallel filesystems suffer lock convoys and other congestion when a large number of different client machines are concurrently writing into the same region of a file. First, all writes to PLFS are separated out into per-process log files so that each process writes to its own file rather than to a single shared file. Second, each node's output stream is hashed to one of the backing filesystems, thus spreading a single file's data. Finally, individual files in a single directory are distributed among backing filesystems using hashing. The first two mechanisms improve the performance of N-1 checkpointing, and the first works even if only a single backing volume is available. When applications read data, PLFS must collect indexing information from all the write logs in order to reassemble application data.

Each file stored in a PLFS filesystem is mapped to one or more "container directories." An example of this is shown in Figure 1. The figure shows a N-1 checkpoint file called `ckpt` being written into a PLFS file by six processes distributed across three compute nodes. The PLFS virtual layer shown is backed by three volumes of a DFS. Each process' blocks

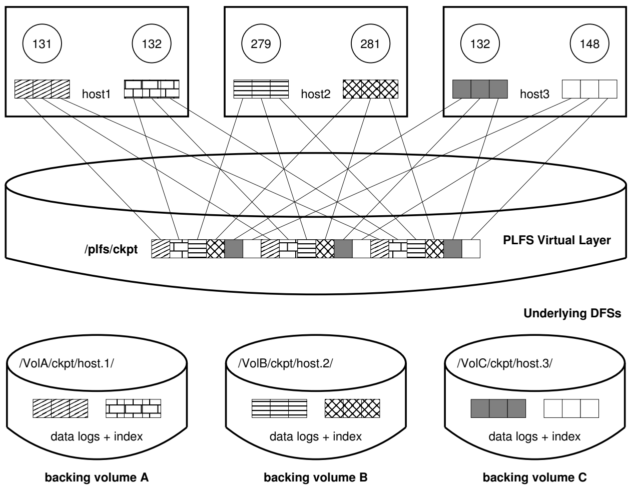

Figure 1: PLFS container structure

of data written to the checkpoint file are strided, resulting in a process' blocks being logically distributed throughout the checkpoint file.

PLFS assigns each host writing to the checkpoint file `ckpt` a backing volume to store data in. For example, writes from `host1` are mapped to backing volume `A`. Then within that volume, each process writing to the checkpoint file is allocated its own private data log file to write to. PLFS keeps indexing information for all writes so that it can reassemble the data if it is read. Without the PLFS layer, the application would be storing all its data in a single checkpoint file on a single backing DFS volume. This would result in poor checkpoint I/O performance due to bottlenecks in the DFS. However, with PLFS, the checkpoint data is separated into per-process log files that are distributed across multiple backing volumes allowing the application to better take advantage of parallelism in the DFS.

3. PLFS ARCHITECTURE EXTENSIBILITY

PLFS, as described above, was originally targeted at improving the performance of N-1 checkpointing for HPC applications. To use PLFS, applications only need to be configured to store their checkpoint files in PLFS — no application source code modifications are required. To achieve this level of transparency, PLFS uses mechanisms such as FUSE [9] userspace filesystems or a PLFS instance of MPI's ADIO layer [28]. In the LDPLFS [30] project, PLFS has been further extended to work with the dynamic linker to replace POSIX C library calls with their PLFS equivalents. These mechanisms allow us to dynamically change HPC application I/O patterns without having to recompile or relink the applications themselves.

After having successfully shown that PLFS can transparently improve an HPC application's I/O pattern for the N-1 case we generalized the PLFS architecture beyond N-1 checkpoints. PLFS is now a general purpose middleware filesystem that can transform an application's I/O access patterns in a number of new ways. Figure 2 shows how the PLFS architecture has been extended in three ways to give it new flexibility in transforming I/O patterns.

First, the PLFS Logical Filesystem interface was inserted beneath the top-level API used by MPI and FUSE. This

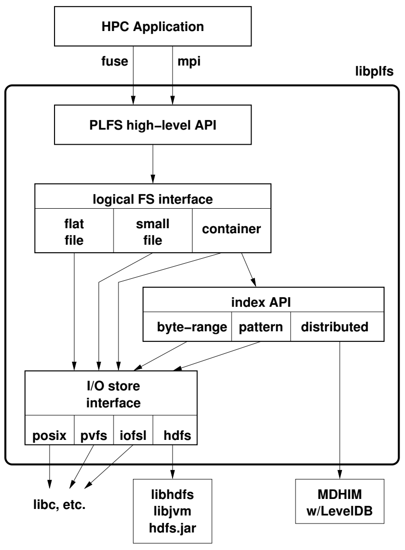

Figure 2: PLFS extensibility

allows other types of filesystems besides the original log-based container filesystems to be inserted into PLFS in a modular way. Two new filesystem types have been added to PLFS: the Flat Filesystem and the Small File Filesystem. Both these new filesystems target an N-N workload. Much like the PLFS Container Filesystem, the Flat Filesystem spreads files in a PLFS directory out across multiple backing filesystems to distribute the metadata load, but the Flat Filesystem does not create container directories or log files, so it avoids the additional overhead associated with those files. The Small File Filesystem is a hybrid between the Flat Filesystem and the Container Filesystem. It combines all the files written by a single process in a directory into a log file, thus making inserting new files in a directory fast at the cost of making reading the directory and file data more complex.

Second, the I/O Store interface was added to the PLFS backend interface to allow PLFS Logical Filesystems to use non-mounted non-POSIX filesystems to store application data. There are currently four PLFS I/O Store modules in PLFS: POSIX, PVFS [17], IOFSL [22, 15], and HDFS [27, 3]. The I/O Store interface enables us to transparently divert application data between radically different filesystem types.

Finally, for the PLFS Container Filesystem we are implementing a modular indexing API. The initial PLFS Container Filesystem uses simple index log records to store data location. This means that each process wanting to read data from a PLFS Container file must read all the index records and merge them together first (when the file is opened for reading). This causes file open operations to be slow and can consume excessive memory since each node in the cluster has a complete copy of the index in RAM. The slow open times can be improved by reading the index in parallel and exchanging the index records between nodes over a high speed interconnect [19], but this does not help the memory usage problem. The indexing API allows alternate indexing schemes to be tried. For example, the indexing module can look for I/O patterns in the index records and compress them into a compact formular [13, 14]. We are also exploring the use of a distributed index (using MDHIM range servers [1] built on top of LevelDB [18]).

In addition to these three extensions, the flexible PLFS framework has also been used to prototype exascale-level I/O subsystems such as burst buffers [4, 7]. A burst buffer is a local solid state storage device (SSD) that quickly absorbs a burst of I/O (e.g. a checkpoint) from an HPC application, allowing the application to continue processing. After the burst, the I/O system drains all burst buffers to permanent backing storage so that they are ready for the next burst. The PLFS Container Filesystem has been used to prototype burst buffers by having each node in a cluster have two types of backends: a local "shadow" backend for the SSD and a "canonical" backend for the permanent storage. When a checkpoint to the SSD is complete, PLFS forks off an asynchronous daemon to drain the burst buffer to the canonical container without having to involve the application in the data transfer.

As an example of PLFS' extensibility, in the rest of this paper we will examine how the PLFS I/O Store layer we built can be used to allow HPC applications to perform N-1 checkpoints to a new filesystem type (HDFS) that they normally would not be able to use.

4. HDFS I/O STORE LAYER FOR PLFS

Cloud storage systems such as HDFS do not support concurrent writing into one file. Fortuitously, data written to PLFS files is broken up and separated into log files in a PLFS container directory. However, the original PLFS assumes that log data files, log index files, and directories can be accessed through a mounted filesystem using the standard POSIX I/O system calls. HDFS, designed only to be accessible through its Java-based API, was not supported.

As shown in Figure 2, our HDFS I/O Store converts the PLFS I/O Store calls into HDFS API calls and passes them to libhdfs. The libhdfs library is a thin C JNI-based wrapper over the Java HDFS API. To use libhdfs, PLFS must be linked with a Java virtual machine (libjvm) so that it can make calls to HDFS Java routines stored in the the Hadoop hdfs.jar file.

We encountered two types of implementation issues when designing and testing PLFS HDFS. First, we had to adapt the HDFS API to match the I/O Store interface, and there were cases where the semantics of the HDFS API did not line up well with the I/O store interface. Second, using the HDFS API requires us to pull in the entire Java virtual execution environment into our application and there were platform issues associated with that.

There were a number of areas where the HDFS API required help to match the I/O Store interface. These areas include:

File group/ownership: HDFS uses text-based user and group names, while PLFS uses integer-based UIDs and GIDs, so we had to use the local password and group file access functions to establish a mapping between these identities for API functions such as `chown`.

Object creation mode: The PLFS I/O Store interface follows the POSIX semantics of allowing an object's permission to be established when a file or directory is first created. HDFS does not provide this semantic, so creating an HDFS object with a given permission requires two HDFS operations: a create followed by a chmod operation.

Reading memory mapped files: HDFS does not support reading files using the `mmap` memory mapped file interface. Since PLFS only reads and never writes files with `mmap`, this interface can be emulated by doing a normal HDFS read into a a memory buffer allocated with `malloc`.

Directory I/O: HDFS does not have a POSIX-like `opendir`, `readddir`, `closedir` interface. We emulate this by caching the entire listing of a directory in memory when it is opened and performing `readdir` operations from this cache.

File truncation: HDFS cannot truncate files to smaller non-zero lengths. This is ok because PLFS only truncates backend files to size zero. HDFS cannot truncate files to size zero either, but this operation can be emulated by opening an file for writing. In this case HDFS discards the old file and creates a new zero length file.

There are several filesystem semantics that HDFS does not provide, but they do not prevent PLFS from operating. These semantics include opening a file in read/write mode, positional write (`pwrite`) operations, and symbolic link related operations. Since PLFS is a log structured filesystem, it does not need or use read/write mode or writing to any location of a file other than appending to the end of it. PLFS also does not require symbolic links in the backend when used with one backing filesystem (but symbolic links have been added to HDFS in version 0.23).

In addition to these issues there are two cases where the semantics provided by the HDFS I/O API are unusual. First, the HDFS API used to create directories (`hdfsCreateDirectory`) functions like the Unix "`mkdir -p`" command — it will create multiple levels of directories at the same time and will not fail if the directories path give already exists. Second, the HDFS API used to rename files and directories ("`hdfsRename`") operates more like the Unix "`mv`" command than the POSIX `rename` system call. Attempts to rename a file or directory to a name of a directory that already exists causes the object being renamed to be moved into the existing target directory rather than having the rename operation fail with a file exists error. This partially breaks PLFS code that handles concurrent file creation races: PLFS still works properly but it is not as efficient as it would be in the POSIX case.

Finally, the HDFS API's handling of errors is unusual because part of it is based around Java exceptions. When HDFS encounters a condition that generates a Java exception, the C-level libhdfs API returns -1 and does not set a meaningful error number. These kinds of exception error occur when trying to perform I/O on an open file that has been unlinked, a situation allowed in POSIX but not HDFS (PLFS can ignore the errors in this case).

5. EVALUATION

To evaluate how the HDFS I/O Store module handles HPC workloads using a Cloud-based storage system, we ran a series of HPC benchmark runs using the open source File System Test Suite checkpoint benchmark from LANL [12]. Our hardware testing platform is the PRObE Marmot cluster [11]. Each node in the cluster has dual 1.6GHz AMD

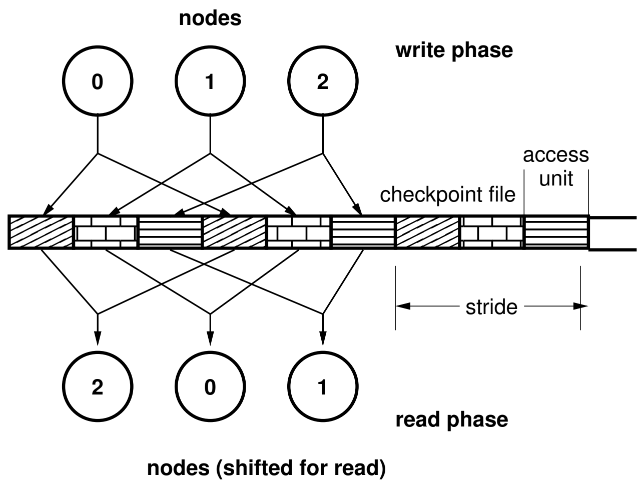

Figure 3: Checkpoint benchmark operation

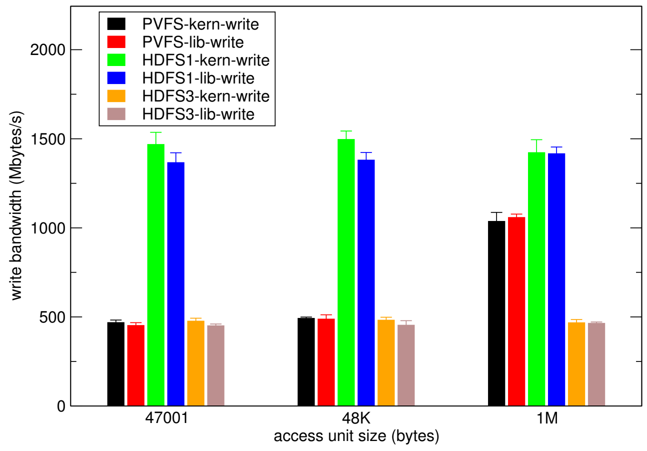

Figure 4: Write bandwidths

Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our tests we run the 64 bit version of the Ubuntu 10 Linux distribution.

5.1 File System Test Benchmark

For all of our tests, we configured the benchmark to generate a concurrently written N-1 checkpoint (Figure 3). In the first phase of the benchmark each node opens the freshly created checkpoint file for writing and then waits at a MPI barrier. Once all nodes are ready, the benchmark starts concurrently writing checkpoint data. Each node writes a fixed number of chunks of data and then waits at a MPI barrier for all writing to complete. Access units are written to the checkpoint file in strides. Each stride contains one access unit from each node. Once writing is complete, the nodes sync data to disk, close the file, and wait at a final barrier before exiting.

Before starting the read phase we terminate all processes and unmount the filesystem in order to flush all caches. After the filesystem has been remounted and restarted, the benchmark reads the checkpoint in the same way it was written, however we shift the nodes around so that each node reads data that some other node wrote (rather than data that it just wrote). This ensures that the benchmark has to use the network to obtain the data it wants to read.

Each node (of 64) generates 512MB of checkpoint data for all our tests for a total of 32GB. We used 3 access unit sizes for our tests: 47001 bytes, 48KB, and 1MB. The 47001 size is a small, unaligned number symptomatic of actual applications causing the worst problems for file systems. The 48K access unit size is close to the 47001 size, but aligned to the system page size. The 1MB size is a more disk friendly access unit size.

5.2 Filesystem Organization

We used two backing filesystems for our tests: PVFS and HDFS (through PLFS). PVFS is our baseline traditional HPC distributed filesystem. For PVFS we used the OrangeFS 2.4.8 distribution, and for HDFS we used Hadoop version 0.21.

Each file in PVFS and HDFS is broken down into fixed-sized chunks. We configured PVFS to use a chunk size of 64MB to match HDFS. The first chunk of a PVFS file gets assigned to an arbitrary node, and subsequent chunks are assigned using round-robin across storage nodes. Unlike HDFS, PVFS does not replicate data as it is assumed to be running on top of a RAID-based underlying filesystem.

Under Hadoop, HDFS is normally not used with hardware RAID. Instead HDFS is configured to write each block of data to three different servers for fault tolerance. For our benchmarking we used HDFS in two modes: HDFS3 and HDFS1. HDFS3 is normal HDFS with 3-way replication, while HDFS1 is HDFS with the replication factor set to 1 (no replicas). We included HDFS1 results because that mode of operation is similar to what PVFS provides (no replicas). HDFS always writes the first copy of its data to local disk. HDFS3 sends a second copy to a node randomly chosen from the same rack and the third copy to a node in a different rack (sent by the second node). For our experiments, all nodes were connected to the same switch, so the second and third nodes are chosen at random.

Both PLFS and PVFS can be used in one of two ways. First, both filesystem can be accessed through a kernel mount-point: PVFS provides its own Linux kernel module for this, while PLFS uses Linux FUSE [9]. Second, PLFS and PVFS can be used as client libraries to avoid the overhead of routing data through the kernel. For our benchmark runs, we have results from PVFS and the HDFS I/O Store under PLFS using both kernel mount points and library interfaces. HPC MPI developers typically use its ADIO [28] abstract-device interface to portably access filesystem client libraries and avoid kernel module overheads.

5.3 Results

Figure 4 shows the write bandwidths for PVFS and PLFS with the HDFS I/O Store module under both kernel and library configurations using access unit sizes of 47001, 48K, and 1M. The results for each test have been averaged over 5 runs made on our 64 node cluster. The error bars indicate the standard deviation across the 5 runs.

The plot shows that the HDFS I/O Store can support concurrent write workloads well with the worst case access unit size of 47001 bytes. To interpret the numbers, note that PVFS is one remote copy, HDFS1 is one local (no net-

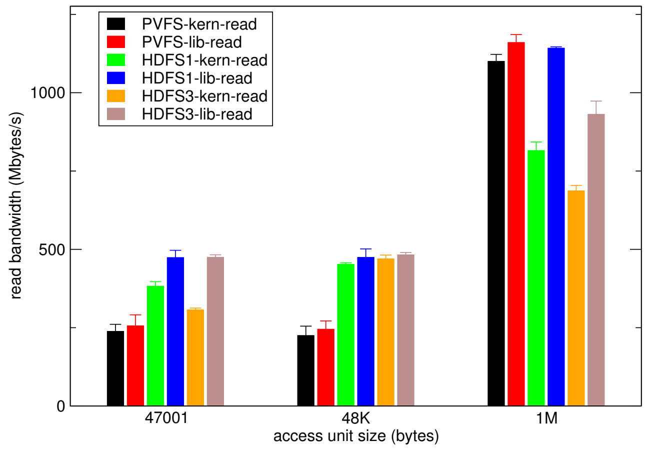

Figure 5: Read bandwidths

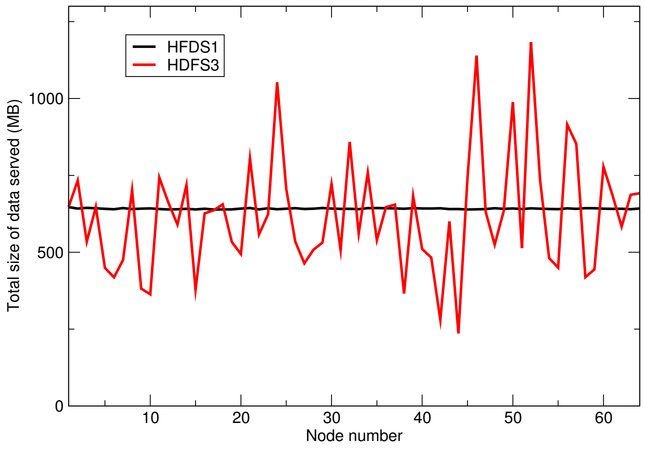

Figure 6: HDFS1 and HDFS3 I/O access patterns

work) copy, and HDFS3 is one local and two remote copies. The HDFS1 bandwidth is limited to around 1450 Mbytes/s by the speed of the local disk. HDFS3 achieves around 1/3 of the HDFS1 bandwidth due to the extra copying, network transits and disk writing. HDFS write bandwidths are unaffected by access unit size because PLFS uses buffered log structured writes so it is disk bottlenecked. The PVFS bandwidth is limited by the synchronous network at 47001 bytes and increases as more efficient access unit sizes are used (1M). PVFS write gets much faster with larger access unit sizes because it does access sized transfers, so a larger access unit size results in fewer different transfers.

Figure 5 shows the read bandwidths achieved by the benchmark after writing the checkpoint file. These results are also averaged over 5 runs with standard deviation error bars shown. Note that nodes are shifted for readback so that no client reads the data it wrote, as shown in Figure 3. This means that for HDFS1 the data being read will always be on a remote node.

In the readback, both HDFS1 and HDFS3 do well. For small access unit sizes HDFS outperforms PVFS. This is because of the log structured writes that PLFS performs with HDFS. PVFS does not have the advantage of log grouping of striped data on readback. For the unaligned 47001 access unit size versus the 48K access unit size, the main change is an improvement in the HDFS readback bandwidth under the kernel mount case. This is because unaligned readback through the kernel mount must go through the Linux buffer cache which stores file data in units of memory pages. In order to fill out the data outside of the 47001 access unit size to page boundaries, the kernel must fetch checkpoint data from neighboring blocks in the checkpoint file. This extra work to align the data to page sized boundaries for the buffer cache results in a 20% performance loss. The library case is not affected by this loss because it does not use the Linux kernel buffer cache.

For HDFS with the large 1M access unit, reading from one of three copies with HDFS3 is around 20% slower than reading from one copy with HDFS1. This is because with three copies HDFS has a scheduling choice as to which of the three copies it reads, where as with HDFS1 it has no choice. The HDFS1 copy of the checkpoint file is perfectly balanced and HDFS scheduling cannot make that worse. With HDFS3 two of the three copies are randomly placed, so the HDFS scheduler can force itself into unbalanced I/O patters (shown by amount of data served per node in Figure 6).

For readback with a 1MB access unit size, the PVFS bandwidth is about the same as the write bandwidth. For HDFS, reading 1MB does not go as fast as PVFS, though the library version of HDFS comes close. HDFS has the added overhead of an extra data copy between the the HDFS Java virtual machine and the PLFS code, and in the case of HDFS1-kernel the FUSE implementation may not be as efficient as the PVFS VFS kernel module. In addition to this, the HDFS3 bandwidth also suffers from the excess scheduler freedom relative to HDFS1.

The performance of the benchmark under HDFS when it is directly linked to the PLFS library (described above) should be close to the kind of performance HPC applications would see when using HDFS with the PLFS ADIO interface under MPI. This is because the PLFS ADIO module is built on top of the PLFS library API, thus avoiding the overheads of going through FUSE and potential problems with the kernel buffer cache when the access unit size is not a multiple of the virtual memory page size.

6. CONCLUSIONS

It might seem that HPC's use of concurrently written checkpoint files would be incompatible with the single-writer, immutable file semantics offered by Hadoop's HDFS Cloud filesystem. In fact, the mechanisms needed to support concurrently written files, and most of the rest of the POSIX API suite commonly used by HPC, can be provided by using PLFS' Container filesystem. In this paper we present an adaptation of PLFS's storage layer to the particulars of the HDFS client API as an example of the extensions possible with our restructured PLFS code base. Using an abstraction architecture, PLFS is both fast and extensible.

7. ACKNOWLEDGMENTS

The work in this paper is based on research supported in part by the Los Alamos National Laboratory under subcontract number 54515 and 153593 (IRHPIT), by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE), and by the Qatar National Research Fund under award number NPRP 09-1116-1-172 (Qloud).

8. REFERENCES

[1] MDHIM: Multi-Dimensional Hashed Index Metadata/Middleware. `https://github.com/mdhim/mdhim-tng/`.

[2] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari. Cloud analytics: Do we really need to reinvent the storage stack? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HOTCLOUD '2009)*, San Diego, CA, USA, June 2009.

[3] Apache. Welcome to hadoop distributed filesystem! http://hadoop.apache.org/hdfs/.

[4] 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 *IEEE Conference on Massive Data Storage*, April 2012.

[5] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: a checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[6] J. Bent and G. Grider. Usability at Los Alamos Nation Lab. In *U.S. Department of Energy Best Practices Workshop on File Systems and Archives*, September 2011.

[7] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. Storage Challenges at Los Alamos National Lab. In *IEEE Conference on Massive Data Storage*, April 2012.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX OSDI 2006*, Seattle WA, Nov. 2006.

[9] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[11] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research. In *USENIX ;login:*, volume 38, June 2013.

[12] G. Grider, J. Nunez, and J. Bent. LANL MPI-IO Test. `http://institutes.lanl.gov/data/software/`, July 2008.

[13] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. Discovering Structure in Unstructured I/O. In *Parallel Data Storage Workshop*, November 2012.

[14] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. I/O Acceleration with Pattern Detection. In *ACM Symposium on High-Performance Parallel and Distributed Computing*, June 2013.

[15] D. Kimpe, K. Mohror, A. Moody, B. V. Essen, M. Gokhale, R. Ross, and B. de Supinski. Integrated In-System Storage Architecture for High Performance Computing. In *International Workshop on Runtime and Operating Systems for Supercomputers*, Venice, Italy, June 2012.

[16] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2009.

[17] R. Latham, N. Miller, R. Ross, and P. Carns. A Next-generation Parallel File System for Linux Clusters. In *Linux World*, January 2004.

[18] LevelDB: A Fast and Lightweight Key/Value Database Library. `http://code.google.com/p/leveldb/`.

[19] A. Manzanares, J. Bent, M. Wingate, and G. Gibson. The Power and Challenges of Transformative I/O. In *IEEE Cluster*, September 2012.

[20] MPI Forum. Message Passing Interface. `http://www.mpi-forum.org/`.

[21] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[22] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa. Optimization Techniques at the I/O Forwarding Layer. In *IEEE International Conference on Cluster Computing*, pages 312–321, Heraklion, Crete, September 2010.

[23] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf. ...And Eat It Too: High Read Performance in Write-Optimized HPC I/O Middleware File Formats. In *Parallel Data Storage Workshop*, Portland, OR, November 2009.

[24] PVFS2. Parallel Virtual File System, Version 2. `http://www.pvfs.org`.

[25] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02 Conference on File and Storage Technologies*, Monterey CA, Jan. 2002.

[26] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[27] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *26th IEEE MSST*, Lake Tahoe NV, May 2010.

[28] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE, 1996.

[29] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.

[30] S. Wright, S. Hammond, S. Pennycook, I. Miller, J. Herdman, and S. Jarvis. LDPLFS: Improving I/O Performance Without Application Modification. In *IEEE International Parallel and Distributed Processing Symposium*, 2012.

[31] W. Yu, J. Vetter, R. S. Canon, and S. Jiang. Exploiting lustre file joining for effective collective io.

In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.