On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS

Wittawat Tantisiriroj   Swapnil Patil   Garth Gibson
Carnegie Mellon University

Seung Woo Son   Samuel J. Lang   Robert B. Ross
Argonne National Laboratory

ABSTRACT

Data-intensive applications fall into two computing styles: Internet services (cloud computing) or high-performance computing (HPC). In both categories, the underlying file system is a key component for scalable application performance. In this paper, we explore the similarities and differences between PVFS, a parallel file system used in HPC at large scale, and HDFS, the primary storage system used in cloud computing with Hadoop. We integrate PVFS into Hadoop and compare its performance to HDFS using a set of data-intensive computing benchmarks. We study how HDFS-specific optimizations can be matched using PVFS and how consistency, durability, and persistence tradeoffs made by these file systems affect application performance. We show how to embed multiple replicas into a PVFS file, including a mapping with a complete copy local to the writing client, to emulate HDFS's file layout policies. We also highlight implementation issues with HDFS's dependence on disk bandwidth and benefits from pipelined replication.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Managment—*Distributed file systems*

Keywords

Hadoop, HDFS, PVFS, cloud computing, file systems

1. INTRODUCTION

Most large data-intensive applications fall into one of the two styles of computing – Internet services (or "cloud computing") or high-performance computing (HPC) – that both execute applications on thousands of compute nodes and handle massive amounts of input data. In both categories, the underlying cluster file system is a key component for providing scalable application performance.

High-performance computing is traditionally defined by parallel scientific applications that rely on low-latency networks for message passing and tiered cluster deployments that separate compute and storage nodes. HPC applications rely on a *parallel file system* for highly scalable and concurrent storage I/O. Examples of parallel file systems include IBM's GPFS [30], Oracle's Lustre [2], Panasas's PanFS [34], and the open-source Parallel Virtual file system (PVFS) [29]. Parallel file systems strive to provide POSIX-like semantics through the traditional UNIX file system interface. These file systems support a broad range of access patterns, particularly highly concurrent data and metadata accesses, for as many different workloads as possible.

On the other hand, Internet services seem very different from HPC: they have a much lower degree of (per-object) concurrency, they offer a more specialized and narrow storage interface to their applications, and they are explicitly aware of the anticipated workloads and access patterns. Consequently Internet services rely on custom, purpose-built storage systems and we will refer to them as *Internet services file systems*. Google's computing environment is a great example: GoogleFS is a high-performance, fault-tolerant distributed file system for applications written using unique semantics such as immutable, append-only operations [16] and it co-designed with the MapReduce job scheduling system, used by many Google services, that colocates computation on nodes that store respective input datasets [13]. Google's success at building scalable and available services has spawned both alternative storage architectures, such as lightweight key-value data stores [10, 14], and open-source reincarnations, such as Hadoop and its distributed file system (HDFS) [9, 18, 32].

Because their target use-cases and interface semantics differ, parallel file systems and Internet services file systems are often considered to be mutually inappropriate for applications of the other category. This is perhaps true for out-of-the-box use, but at a deeper level there are more similarities than differences. This observation is gaining momentum in both research and development. In fact, both the cloud computing community and HPC community are starting to support these two types of storage architectures and file systems. For example, Amazon EC2 recently launched services for running HPC applications [1] and Argonne National Laboratory's Magellan cluster is providing users with different non-POSIX storage interfaces, including Amazon S3 and HDFS [25]. Our goal in this work is to analyze the design and implementation tradeoffs made in parallel file systems and Internet services file systems, and understand how these tradeoffs affect the durability, consistency, and performance of data-intensive applications.

This paper examines PVFS, a scalable parallel file system used in production HPC at the largest scales [23], and HDFS, the primary storage system widely used in the Apache Hadoop (MapReduce) cloud computing stack. Both are open source, capable of gigabytes per second and petabytes of capacity, and both can be configured to export disks located on the same computers that are being used as compute nodes. Because the semantics of PVFS are broader than HDFS, we configure and integrate PVFS into Hadoop and evaluate HDFS versus PVFS performance on a set of data-intensive computing benchmarks and real applications. Unmodified Hadoop applications can store and access data in PVFS using our non-intrusive shim layer that implements several optimizations, including prefetching data, emulating replication and relaxed consistency, to make PVFS performance comparable to HDFS.

This shim also helps us understand the differences between HDFS and PVFS. The first difference is their fault tolerance strategy. HPC systems employ RAID erasure coded redundancy so that the cost of disk failure tolerance is less than 50% capacity overhead, while Internet services typically employ whole file replication at 200% capacity overhead in order to tolerate the loss of a complete node (or RAID controller) without data unavailability. In addition to this cost difference, HPC systems expect RAID to be implemented in hardware, while Internet service systems prefer software-managed redundancy. We evaluate two different mappings for replicating files in PVFS in terms of their performance and similarity to HDFS strategies. Using replicated files, Internet service file systems expose the location of all copies to allow the Hadoop task scheduler to place computation near its data storage. This mechanism that exposes data layout is similar to the mechanism in HPC file systems that allows users to control data layout, and we show how PVFS easily adapts to expose the layout of data replicas. The second difference between HDFS and PVFS is client cache consistency. HPC systems support concurrent write sharing, forcing limitations on aggressive client caching. When PVFS is used with inconsistent client caching and deep prefetching, as HDFS is used by Hadoop, apparent performance differences are reduced. Finally, our analysis also observes significant differences in terms of resource efficiency of these two file systems: HDFS today has a stronger dependence on disk bandwidth, a relative weakness, and makes better use of network bandwidth, a relative strength.

We present a primer on HDFS and PVFS in Section 2, the implementation of a Java shim and PVFS extensions in Section 3, the performance evaluation in Section 4, the related work in Section 5, and conclusion in Section 6.

2. HDFS AND PVFS – A DESIGN PRIMER

Despite their different origins and contrasting use-cases, both HDFS and PVFS have a similar high-level design. They are user-level cluster file systems that store file data and file metadata on different types of servers, i.e., two different user-level processes that run on separate nodes and use the lower-layer local file systems for persistent storage.[1] The file system metadata, including the namespace, block location and access permission, is stored on a separate server called the metadata server (MDS) in PVFS and namenode in HDFS. All metadata operations may be handled by a single server, but a cluster will typically configure multiple servers as primary-backup failover pairs, and in PVFS, metadata may be distributed across many active MDSs as well. All file data is stored on persistent storage on a different set of servers called I/O servers in PVFS and data servers in HDFS. Instead of using HDFS or PVFS specific terminology, the rest of paper will use the terms metadata server (MDS) and data server. In a typical cluster deployment, data is distributed over many data servers while the metadata is managed by a single MDS.

Both HDFS and PVFS divide a file into multiple pieces, called chunks in HDFS and stripe units in PVFS, that are stored on different data servers. Each HDFS chunk is stored as a file in the lower-layer local file system on the data server. Unlike HDFS's file-per-chunk design, PVFS splits a file into an object per data server, where each object includes all stripe units of that file stored on that data server as a single file in the underlying local file system. The chunk size and stripe unit size are configurable parameters; by default, HDFS uses 64 MB chunks and PVFS uses 64 KB stripe units, although modern PVFS deployments use larger stripe units (2-4 MB) to get higher data transfer bandwidth.

To access files, clients first ask the MDS which data servers it should contact. Clients then send all read and write requests directly to the data servers. Clients cache the data server layout information received from the MDS to interact repeatedly and directly with data servers (without going through the MDS). PVFS is able to easily cache a file's layout because a stripe unit's location in an object is algorithmically derived from its offset in the file. HDFS, on the other hand, stores lists of chunks for each file. During reads, HDFS can fetch and cache the chunk list of a file because HDFS files are immutable. But during the write phase, HDFS must contact the MDS for each new chunk allocation (as the file grows), whereas PVFS does not. Neither HDFS nor PVFS caches any file data at the client machines although HDFS does prefetch as deeply as a full chunk while PVFS does not prefetch at all.

Rest of this section contrasts the design and semantics of HDFS and PVFS (summarized in Table 1), and occasionally compares them with other scalable file systems.

2.1 Storage deployment architecture

The storage and compute capabilities of a cluster are organized in two ways: either co-locate storage and compute in the same node or separate storage nodes from compute nodes. Both HDFS and GoogleFS use the former approach with 2-12 disks attached locally in each machine with a commodity processor and few gigabytes of RAM [12]. This "disk-per-node" model is well suited for the Hadoop/MapReduce data processing abstraction seeking to co-locate a compute task on a node storing the required input data. This model is an attractive approach for cost-effective high bandwidth.

PVFS is typically deployed using the other approach: storage servers separate from the compute infrastructure.[2] In this model, pooling storage nodes together enables highly parallel I/O [17] and separating storage (from compute) helps build optimized reliability and manageability solutions [34]. This approach has marginally higher capital costs than

[1]In PVFS, a node can be responsible to store data, metadata or both. HDFS has a stricter physical separation of responsibility.

[2]PVFS can be configured so that the same nodes are used for both compute and storage, and Hadoop/HDFS can be configured to separate data and compute servers.

	Hadoop Distributed File System (HDFS)	Parallel Virtual File System (PVFS)
Deployment model	Co-locates compute and storage on the same node (beneficial to Hadoop/MapReduce model where computation is moved closer to the data)	Separate compute and storage nodes (easy manageability and incremental growth)
Concurrent writes	Not supported – allows only one writer per file	Guarantees POSIX sequential consistency for non-conflicting writes, i.e., optimized writes to different regions of a file
Small file operations	Not optimized for small files; client-side buffering aggregates many small requests to one file into one large request	Uses few optimizations for packing small files, but the lack of client-side buffering or caching may result in high I/O overhead for small write requests
Append mode	Write once semantics that allows file appends using a single writer only	Full write anywhere and rewrite support (research prototypes have extended PVFS to support concurrent appends through atomic operators)
Buffering	Client-side readahead and write-behind staging improves bandwidth, but reduces durability guarantees and limits support for consistency semantics	No client-side prefetching or caching provides improved durability and consistency for concurrent writers
Data layout	Exposes mapping of chunks to data-nodes to Hadoop applications	Maintains stripe layout information as extended attributes but not exposed to applications
Fault tolerance	Uses rack-aware replication with, typically, three copies of each file	No file system level support; relies on RAID subsystems attached to data servers
Compatibility	Custom API and semantics for specific users	UNIX FS API with most POSIX semantics

Table 1: Comparing the design and semantics of PVFS [29] and HDFS [9].

the disk-per-node model, but it comes at a lower operational complexity, which is becoming a growing bane for users of large computing infrastructure.[3]

2.2 File access semantics

Concurrency – The semantics of most Internet services file systems, including HDFS, GoogleFS and Amazon S3, are optimized for their anticipated workloads. Files in HDFS have write-once-read-many semantics and have only one writer. HDFS does not allow changes to a file once it is created, written, and closed. Opening an existing file for write truncates all old data. These semantics favor Hadoop/MapReduce applications that typically manipulate collections of files in one dataset. The lack of support for concurrent write sharing simplifies data consistency semantics in HDFS. However, unlike HDFS, GoogleFS supports restricted file mutation by appending new data to a file including "atomic" operations to support concurrent appends to a single file [16].[4]

Most parallel file systems support a wide variety of file operations, especially for highly concurrent file access. PVFS provides high throughput using "non-overlapping concurrent write" semantics. If two clients write to non-overlapping byte regions of a file, then all other clients will see the data from the writers after their respective writes have completed. However, if two clients concurrently write data to the same region of a file, the result is undefined. Thus PVFS adheres to the POSIX consistency semantics in the case of non-overlapping writes by guaranteeing sequential consistency.

Like HDFS, the current PVFS release does not support appending writes to a file. Appends can be implemented in PVFS using a "fetch-and-add" style operation using extended attributes. The End-of-File (EoF) offset of a file can be stored as an extended attribute. PVFS clients could atomically "fetch" the EoF offset and "add" a value to it, returning the initial value so that the client can then write data starting at that offset [24].

Client-side buffering and consistency – HDFS is designed to enable high write throughput (instead of low latency) batch processing [9]. It enables streaming writes through "write staging" at the clients. Clients send a write to a data server only when they have accumulated a chunk size (64 MB) of data. Initially, clients buffer all write operations by redirecting them to a temporary file in memory. Once filled, the clients flush this buffer to the data server responsible for storing that file chunk. If the file is closed when the buffer is not yet full, the buffer is flushed to the chunk's respective data server.

On the other hand, PVFS does not have any client-side buffering and sends all application level write system calls directly to an I/O server. This approach is not optimal for small incremental write workloads; however, small file I/O continues to be uncommon among scientific applications. Such applications do not benefit from caching because they operate on files that are too big to fit in memory. Moreover, by not performing any client-side buffering, PVFS does not require complex cache consistency protocols for concurrent write operations to the same file. Typically in large systems with many concurrent writers, cache consistency and synchronization mechanisms are a potential source of bottleneck. Nevertheless, other HPC parallel file systems, including GPFS, PanFS and Lustre, implement client caching. Since HDFS allows only one writer per file, its client-side write staging does not encounter any chunk inconsistency issues. PVFS does not use any explicit caching on the I/O servers either; it relies on the buffer cache of the underlying local file system on the server.

[3]In fact, this model is also adopted by some Internet services, most notably Amazon's web services platforms. In Amazon's cloud infrastructure the compute instances (called EC2) are separate from the underlying storage infrastructure (called EBS and S3) [4–6]. This allows the EBS storage system to provide features for high availability, seamless data volume migration and data backups without being dependent on EC2.

[4]Newer releases of Hadoop/HDFS support file append but only by a single writer.

2.3 Data layout and function shipping

Recall that large files in HDFS and PVFS are divided into chunks and stripe units, respectively, that are distributed across multiple data servers. However, these file systems are different in the way they divide a file and the policies used to layout the files (and in the way they make that information available to higher-level applications).

The first difference between HDFS and PVFS is the location metadata associated with a file. When HDFS creates a file, it allocates (64 MB) chunks dynamically and the number of chunks grows as the file grows in size. PVFS has a fixed number of objects for every file and the identifiers of these objects remain constant throughout the lifetime of the file. It is important to distinguish these two approaches because PVFS clients can access data from data servers directly by caching the list of (immutable) object identifiers, while HDFS clients need to request the metadata server for the identifier and location of the chunk that holds the desired data. In addition, many writers could overload HDFS's MDS with new chunk requests.

This leads us to the second difference between HDFS and PVFS: policies used to layout the file data across servers. HDFS uses a random chunk layout policy to map chunks of a file to different data servers. When an MDS creates a chunk, it randomly selects a data server to store that chunk. This random chunk allocation may lead to a file layout that is not uniformly load balanced. PVFS, however, makes all layout decisions when the file is created and extends the end of each component object when new stripe units are added in a round robin manner.

The key facet of HDFS design is that this layout information is exposed to the Hadoop framework for scheduling a large parallel application on the cluster. Hadoop's job scheduler divides an application job into multiple tasks that operate on an input file (which is split into chunks) stored in HDFS. By default, the scheduler assigns one task per chunk – it tries to assign a "map" task to a node that also stores the respective chunk, and if such a local assignment cannot be made, the "map" task is assigned to a remote node. Once a node has a "map" task, it reads the input data chunks, performs computation on each chunk, and writes the output to intermediate files stored on local storage. A daemon process transfers these intermediate files to the local storage of nodes that run "reduce" tasks. These "reducer" nodes read these files from local storage, apply the appropriate reduce function and write the final output to files in HDFS.

Hadoop uses various optimizations proposed by Google's MapReduce. The job scheduler assigns tasks to nodes in a manner that load-balances the file I/O performed in reading the input data. Hadoop schedules tasks to be run out-of-order to overcome the potential drawbacks manifesting from the HDFS's non-uniform chunk placement resulting from the random chunk allocation policy. Hadoop also runs backup tasks that help minimize a job's response time by re-executing "straggler" tasks that have failed to complete.

Unlike HDFS, PVFS does not expose a file's object and stripe unit layout to the application by default; we implemented a mechanism that queries PVFS for this layout information and exposes it to the Hadoop/MapReduce framework (details in Section 3).

2.4 Handling failures through replication

Failures are common in large clusters and scalable file systems detect, tolerate and recover from component failures [27, 31]. Most parallel file systems, including PVFS, rely on hardware-based reliability solutions such as RAID controllers attached to data servers; one exception is PanFS which uses RAID-5 across nodes [34].

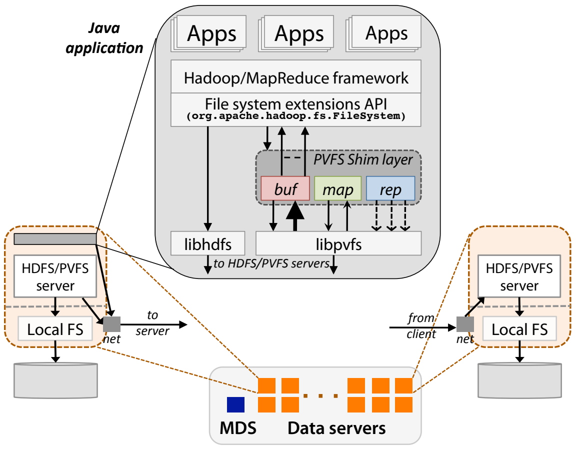


Figure 1: Hadoop-PVFS Shim Layer – ***The shim layer allows Hadoop to use PVFS in place of HDFS. This layer has three responsibilities: to perform readahead buffering ('buf' module), to expose data layout mapping to Hadoop ('map' module) and to emulate replication ('rep' module).***

On the other hand, both GoogleFS and HDFS replicate data for high availability [16, 32]. HDFS maintains at least three copies (one primary and two replicas) of every chunk. Applications that require more copies can specify a higher replication factor, typically at file create time. All copies of a chunk are stored on different data servers using a "rack-aware" replica placement policy. The first copy is always written to the local storage of a data server to avoid the network overhead of writing remotely. To handle machine failures, the second copy is distributed at random on a different data server that is in the same rack as the data server that stored the first copy. This improves network bandwidth utilization because intra-rack communication is often faster than inter-rack communication. To maximize data availability in case of a rack failures, HDFS stores a third copy distributed at random on data servers in a different rack. In HDFS, chunks are replicated by the data servers using "replication pipelining" where a data server that receives a chunk sends the chunk to the data server that stores the next copy [9]. The list of data servers that will store copies of any chunk is determined and maintained by the MDS (at file creation time).

3. HADOOP-PVFS EXTENSIONS

In this section, we describe our modifications to the Hadoop Internet services stack to plug in PVFS and the functionality extensions made to PVFS.

3.1 PVFS shim layer

Figure 1 shows the design of the PVFS shim layer that enables the Hadoop framework to use PVFS instead of HDFS. The shim uses Hadoop's extensible abstract file system API (`org.apache.hadoop.fs.FileSystem`) to use PVFS for all file I/O operations. Prior systems like the Kosmos filesystem (KFS) [11] and Amazon S3 [20] have similarly used this file

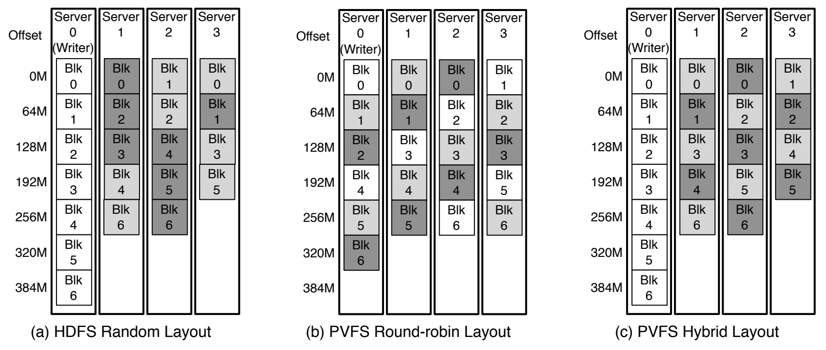


Figure 2: Different data layout schemes used by HDFS and PVFS – ***HDFS places the first copy locally on the writer's data server and the next two copies on randomly chosen data servers. PVFS's default strategy is a round robin policy that stripes the three copies on three different servers. A third policy, called the hybrid layout, was created for PVFS, which places one copy on the writer's (local) data server and stripes the other two copies in a round-robin manner.***

system API to build backend stores for Hadoop applications. The shim layer is implemented using the Java Native Interface (JNI) API to allow the Java-based Hadoop applications make calls to the C-based PVFS library. The shim layer has three key components: the *buf* module for readahead buffering, the *map* module for exposing file layout information to Hadoop, and the *rep* module for replication. Our shim layer implementation consists of 4,000 lines of code that includes support for all data layouts described in this paper.

Readahead buffering (*"buf"* module) – While applications can be programmed to request data in any size, the Hadoop framework uses 4KB as the default amount of data accessed in each file system call. Instead of performing small reads, HDFS prefetches the entire chunk (of default size 64MB) asynchronously and then synchronously reads a buffer at a time from the network socket providing the prefetch. This "readahead" mechanism is important for reducing file system overhead but it is also incompatible with a non-caching file system client like PVFS. We modified the shim to provide similar buffering for PVFS without implementing a 64 MB prefetch. For every 4KB request made by Hadoop, the shim layer synchronously reads a larger buffer (which is a configurable parameter) from PVFS; that is, without an asynchronous prefetch of the whole chunk. We found that a synchronous fetch of 4MB is almost as efficient as an asynchronous pipeline prefetching 64 MB, and much less invasive for a system like PVFS that does not implement a client cache consistency protocol.

Data mapping and layout (*"map"* module) – The Hadoop/Mapreduce job scheduler distributes computation tasks across many nodes in the cluster. Although not mandatory, it prefers to assign tasks to nodes that store input data required for that task. This requires the Hadoop job scheduler to be aware of the file's layout information. Fortunately, as a parallel file system, PVFS has this information at the client, and exposes the file striping layout as an extended attribute of each file. PVFS uses two extended attributes to expose the file striping layout. The first attribute, `system.pvfs2.md`, describes how blocks are striped across servers; we are using "`simple_stripe`" which stripes a file across all servers in a round-robin pattern using a 64MB stripe unit. The second attribute, `system.pvfs2.dh` stores the list of all servers involved in this striping pattern.[5]

[5]Actually this is a list of the PVFS object handles, but the server address list can be generated from this list.

Replication emulator (*"rep"* module) – Because a fair comparison of replication and RAID is likely to be controversial and because Internet services file systems rely on software mechanisms for reliability, we have modified PVFS to emulate HDFS replication. PVFS uses its distributed layout policies to control replica placement across the cluster. Although PVFS does not embed rack-level topology in its configurations, our techniques can still place most of the chunks on servers in different racks.[6]

The shim emulates HDFS-style replication by writing on behalf of the client to three different data servers with every write; that is, it sends three write requests to different servers, unlike HDFS's replication pipeline. The destination of each write is determined using the layout scheme described in Figure 2. HDFS's random chunk layout policy is shown in Figure 2(a) and PVFS's default round robin striping policy is shown in Figure 2(b). However PVFS's default policy is not comparable to HDFS's layout, which writes one replica locally and two replicas to remote nodes selected in a rack-aware manner. In order to provide such a local write within PVFS, we added a new layout policy, "PVFS hybrid" in Figure 2(c), where one copy of each block is written to the writer's data server (like in HDFS) and the other two copies are written to other servers in a round-robin manner (like the default scheme in PVFS). Although this scheme reduces the network traffic by using a local copy, it may cause an imbalance unless all clients are concurrently writing to different files. Our approach was motivated by the simplicity of emulating replication at the client instead of making non-trivial changes to the PVFS server implementation.

3.2 PVFS extensions

An early version of our work shows that PVFS can be plugged in to the Hadoop stack without any modifications [33]. However, this does not allow the client-side shim layer to control replica placement and consistency semantics – two features that required about 400 lines of new code in PVFS.

Replica placement – As mentioned in the previous section, the "rep" module in the shim layer issues three requests to write three copies of a file and sends each of these requests to the appropriate server computed using layouts shown in

[6]PVFS can be modified to use rack-awareness like HDFS, but our goal was to minimize the changes to PVFS and still achieve properties similar to HDFS.

Figure 2. Because the shim is doing all the replication work, the role of our PVFS extensions is only to place the first object in the local server, i.e., we modified PVFS to enable the HDFS policy of writing the first copy of the object on the data server residing on the same machine as the writer.

`flush()` issues – In PVFS, a `flush()` causes the data server to call `fsync()` to write the buffer cache to disk. But `flush()` call in HDFS does not behave in this manner. For fair comparison with HDFS, we disabled all `fsync()` calls made by PVFS during `flush()`. However, PVFS will still synchronously write to disks when starting or stopping the server. Note that disabling all `fsync()` operations in PVFS does not change the semantics from the application perspective. Both the BDB cache and the OS buffer cache are still consistent with respect to operations from different clients.

4. EXPERIMENTAL EVALUATION

This section explores the performance of HDFS and PVFS, with our new layouts, using microbenchmarks, Hadoop benchmarks and real scientific applications.

4.1 Methodology

Our experimental cluster, called *OpenCloud*, has 51 nodes with Hadoop-0.20.1 (including HDFS), PVFS-2.8.2 with appropriate patches for different layout schemes and Java SE Runtime 1.6.0. Both HDFS and PVFS use a single dedicated metadata server and 50 data servers. Each cluster node has a 2.8 GHz dual quad core Xeon 5300 CPU, 16 GB RAM, 10 Gbps Ethernet NIC, and four Seagate 7200 RPM SATA disk drives. Because PVFS is typically not used with multiple independent file systems in the same data server, we use only one disk drive. These machines are drawn from two racks of 32 nodes each with an Arista 7148S top-of-rack switch and these rack switches connect to an Arista 7124 head-end switch using six 10 Gbps uplinks each. Each node runs a Debian Lenny 2.6.32-5 Linux distribution with the XFS file system managing the disk under test.

While one disk per 10 Gbps Ethernet network link is a much more disk-bottlenecked balance than in many cloud configurations, it serves to emphasize a file system's dependence on disk performance. To explore the opposite extreme, where there is much more storage bandwidth than network bandwidth, we run some tests with a RAMDISK per node as a model of the highest bandwidth solid-state disks.

For all experiments, we measure the *aggregate user throughput* as the number of bytes read or written by the user application during the time it took to complete the benchmark on all nodes that participate. This "user" throughput is what is observed by the application; in other words, if an application writes a 1 GB file in 10 seconds, the user throughput is 0.1 GB/s, even if the cluster file system writes three copies of this file (i.e., 3 GB of raw data). For deployments with parallel clients, we measure throughput as the total user data moved between the time when the test starts to when the last node completes. This throughput measurement technique, instead of summing up the average throughput of each client, accounts for the "slowest component" in a large parallel application. One such example is slow tasks, called stragglers, in MapReduce jobs [13, 35].

4.2 Baseline performance

To understand the baseline performance of PVFS relative to HDFS, we run a large Hadoop `grep` application that

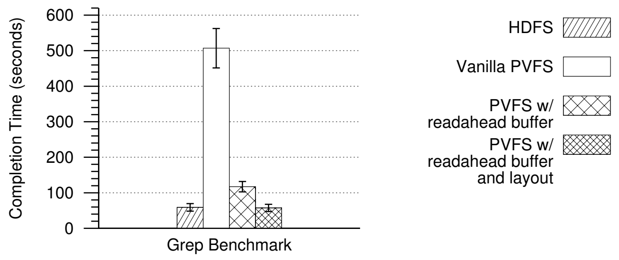


Figure 3: Benefits of using readahead buffering and layout information – ***By using both these techniques in its shim layer, PVFS matches the performance of HDFS for the read-intensive Hadoop* `grep`.**

searches for a pattern in a 50 GB input dataset stored on the 50 node setup; this dataset has 50 data files, each 1 GB in size (i.e., ten million 100-byte records), distributed over 50 data servers. Note that `grep` is a highly read-intensive benchmark that scans the entire dataset once to search for the desired pattern. We configure Hadoop to use vanilla PVFS through the Hadoop-PVFS shim with both readahead and layout awareness disabled. Figure 3 shows that vanilla PVFS is more than an order of magnitude slower than HDFS because the latter benefits significantly from its readahead buffering and Hadoop's use of file layout information (while the former has neither of these features).

We repeat the above experiment with both readahead and file layout functionality enabled in the Hadoop-PVFS shim. Figure 3 shows that using a 4 MB readahead buffer alone (without file layout optimization) enables PVFS to reduce the application completion time by 75%. By doing both readahead and allocation of tasks nearer to the input data based on PVFS's file layout information, Hadoop-on-PVFS matches the performance of Hadoop-on-HDFS. The rest of the evaluation uses the shim, with readahead enabled and file layout exposed, for all PVFS I/O.

As described in Section 3.1, the shim performs readahead in a synchronous manner and this makes the readahead buffer size an important design parameter. To choose the appropriate buffer size, we perform an experiment where a single client sequentially reads a 1 GB file that is striped over 50 data servers. Starting from the beginning of the file, a test application reads a parameterized amount, starting with 4 KB, in each request until it reaches the end of file. In addition to the OpenCloud cluster, referred to as "oc" in Figure 4, we repeat this experiment on another 16-node cluster, called *"ss"*, with slower five-year old disks and 1 GigE NICs. Both Hadoop and PVFS use 64 MB stripe units (chunks) and the PVFS shim did not use any readahead buffering.

If a client's Hadoop application requests 4 KB of data, PVFS without readahead buffering fetches only 4 KB from a server. By performing such small reads, PVFS's read throughput is dominated by high message processing overhead and network latency. As shown in Figure 4, PVFS reads data at less than 10 MB/s (for the 4 KB buffer size) for both clusters. Unlike PVFS, HDFS tries to read the entire 64 MB chunk asynchronously and then sends it to the application as soon as the first 4 KB is received. But the disk may not fetch all 64 MB immediately because the network stack buffering may fill and temporarily block the prefetching thread in the data server. Figure 4 shows how

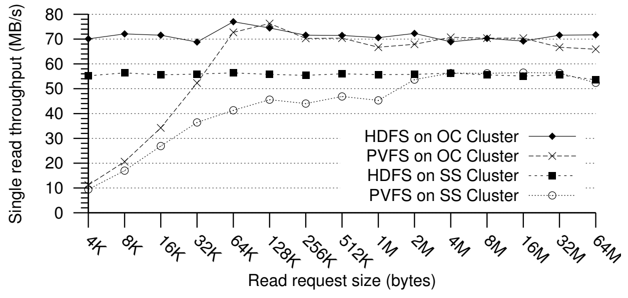


Figure 4: Improved single client read throughput using readahead – ***Using a large enough readahead buffer, the PVFS shim delivers a read throughput comparable to HDFS's in-built prefetching mechanism that reads the entire 64 MB chunk for all requests.***

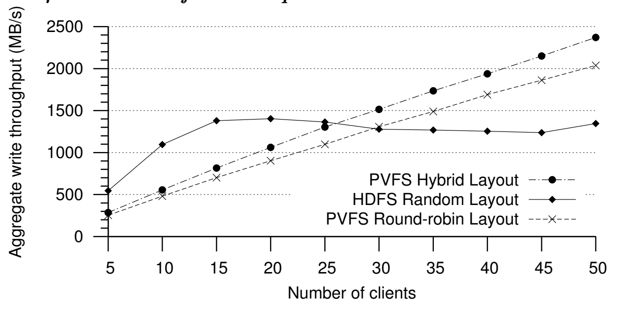


Figure 5: ***N*****-clients concurrently writing to** ***N*** **separate 1 GB files –** ***By writing one copy locally, PVFS with hybrid layout out-scales PVFS with round-robin layout, and both scale much better than HDFS's random layout policy for more clients.***

PVFS's read performance improves with larger readahead buffers implemented in the shim. In the slower *"ss"* cluster, the readahead performance peaks when the shim layer uses a 4 MB readahead buffer; using bigger buffers does not yield any higher read throughput because the server's local file system is saturated and is delivering peak read throughput. However, in the faster *"oc"* cluster, the readahead performance peaks with only a 64 KB buffer. Rest of the paper uses a 4 MB buffer for the Hadoop-PVFS shim (optimizing for the clusters with slower components) and a 128 KB buffer for HDFS (the recommended size for Hadoop [19]).

4.3 Microbenchmarks using file system API

We use file system microbenchmarks to help us understand the behavior of HDFS and PVFS with different data layout schemes for various read and write access patters. In these microbenchmarks, each client either writes or reads a 1 GB file using the file system API (and not as a Hadoop job). We use the file system API directly because it gives us better control for understanding the different policies of the file system, the effects of which may be masked by Hadoop's task assignment policies. Each file makes three copies that are stored using the layouts described in Section 3.2. All reported numbers were an average of three runs (with a negligible standard deviation across runs).

Concurrent writes to different files – This microbenchmark measures the write performance when multiple clients are concurrently writing to separate files that are each spread over the same 50 data servers – an access pattern that emulates the "reduce" phase of a MapReduce job, when the "reducer nodes" all generate and write different output files.

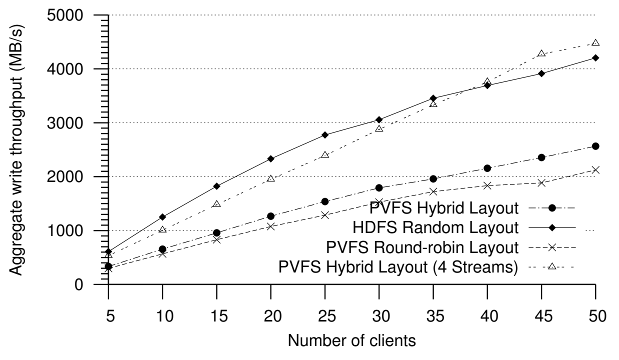


Figure 6: Impact of disk-network speed balance on write performance – ***We use a RAMDISK to show that the HDFS scalability problem in Figure 5 was caused by slow chunk creation by a busy disk.***

Figure 5 shows the performance of writing fifty 1 GB files using the three different layout schemes. The interesting observation in this figure is that PVFS does not saturate while HDFS saturates earlier and at a bandwidth much lower than expected. We discovered that the result in Figure 5 is strongly dependent on the file size (1 GB) and on the memory resources of our cluster (16 GB memory per node) – in this case, all data being written (three copies of each file) can fit in the write-back cache of the lower-layer local file system on each node. Although all data may fit in the write-back cache, the dirty data is written to the disk in the background by a Linux process, called `pdflush`, as the amount of dirty data in the write-back cache reaches a threshold [21]. Once this benchmark uses 15 clients, the amount of data in each dirty write-back cache exceeds the threshold. During this period, we observed that the time taken to create one file can be up to a few hundred milliseconds; this slow create during writeback is not a problem for PVFS because all creation is done at file creation, so it continues to fill the writeback buffer while data is being flushed to disk. But HDFS creates a file for each chunk dynamically and no new data is written into the writeback buffer while the file for a chunk is being created. In this way, HDFS performance is often limited and saturated by the time to create files while the disk is busy flushing dirty data.

Figure 6 shows the same experiment except that rather than using the local disk on each node we create a 4 GB in-memory file system (Linux `tmpfs`) and use it as the storage under test. This results in much higher throughput: HDFS throughput rises by a factor of three while PVFS throughput changes very little. HDFS performance, when all the data being written fits in the write-back cache, is strongly dependent on a time taken to create a file while PVFS performance is not.

Figure 6 also shows that using four concurrent instances (streams) of the benchmark on each node to write to PVFS with hybrid layout matches HDFS's higher throughput. PVFS network transfer protocols are not achieving as much concurrency as HDFS. This happens because HDFS uses its data servers to implement file replication; that is, when a client application writes a chunk of a file to a server, this server writes the second copy to another server, and this second server writes the third copy of the chunk to yet an-

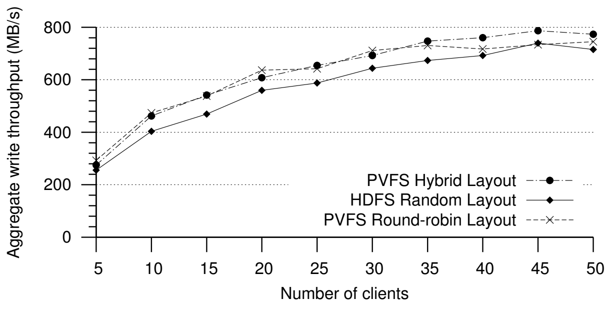


Figure 7: Impact of smaller write-back cache on write performance – ***HDFS's random layout policy loses the benefit from its replication pipeline and the amount of time taken to create files has less effect on HDFS's random layout policy performance.***

other server. As a result, when a single chunk is written by a client, this HDFS replication pipeline invokes two more servers to act as "proxy clients" to write the replicas on other servers [9]. HDFS involves more nodes in the cluster to write a file than PVFS because the PVFS shim layer at the client is performing all the replication work serially: it writes the first copy, then the second copy and then the third copy. If the application has sufficient concurrency, this serial versus pipeline replication does not matter but at lower concurrency it does matter. In both Figure 5 and Figure 6, PVFS with hybrid layout continuously outperforms PVFS with round-robin layout because the former avoids some network traffic by writing the first copy locally.

Figure 7 shows the disk version of the same experiment where the size of Linux's write-back cache is reduced to one-tenth of the original size; in other words, we are emulating a case where disks become the bottleneck much sooner. For all layout policies in both file systems, the write throughput saturates with an increasing number of clients because the amount of write-back cache available to each client decreases. HDFS's random layout policy loses the benefit from its replication pipeline because its better use of cycles in all nodes does not matter when writing is blocked on a full writeback buffer and the disks are saturated. HDFS throughput is only slightly lower than PVFS with hybrid layout throughput because the time to create a file for an HDFS chunk is significantly lower than the time to write 64 MB of chunk data to disk.

Finally, to demonstrate the scalability of HDFS and PVFS with our modification, we perform an experiment that writes 1 GB per client to a setup with variable number of server resources All the preceding evaluation was done on a cluster with 50 data servers; Figure 8 shows the performance of HDFS and PVFS with a varying number of data servers. In this experiment, we vary the number of servers from 5 to 50 and we configure each data server to also act as a client; for example, the 10 server case shows a configuration with 10 nodes, each acting as a data server and as a client. Since the number of data servers is equal to the number of clients, the amount of write-back cache available to each client is fixed. Figure 8 shows that the performance of all layouts starts lower but increases almost linearly without saturating.

Concurrent reads from different files – In this microbenchmark, each client reads the file it just wrote; this emulates the common read pattern in the MapReduce framework. We use 5-50 clients, each sequentially reading a dif-

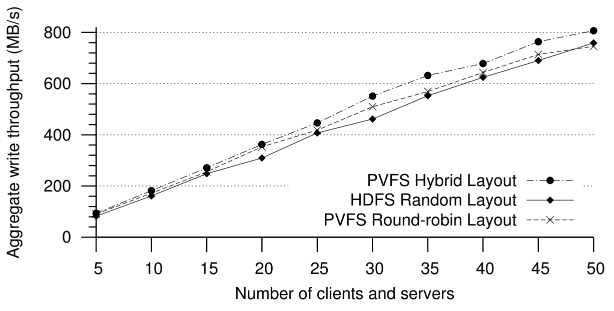


Figure 8: Performance of HDFS and PVFS as a system scales out servers as well as clients for concurrent writing.

ferent 1 GB file, stored on the same node, and all caches are flushed to ensure that data is read from the disk.

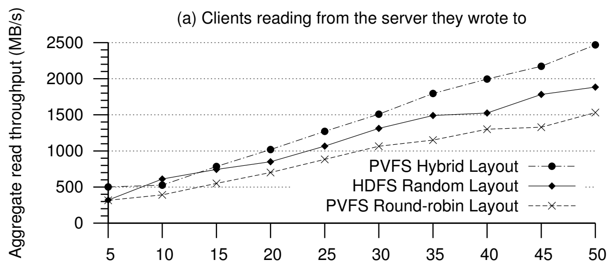
Figure 9(a) shows that both PVFS with the hybrid layout and HDFS with its random layout outperform PVFS's default round-robin layout. This happens because the latter scheme always writes all copies on remote servers and does not see the benefit that the other two layouts receive from writing the first copy locally.

To show the performance impact of not using the local copy, we use a microbenchmark similar to the previous experiment (in Figure 9(a)) where 50 clients are concurrently reading different 1 GB files than the ones they wrote, guaranteeing that there is rarely a "local" copy. The main difference in this microbenchmark is that all client reads are from a *remote* server. Figure 9(b) shows that all layout configurations deliver the same aggregate read throughput, but the performance is significantly lower than when the clients read a local copy and is about the same as PVFS with round-robin layout, as shown in Figure 9(a). Note that this difference in performance with and without a local copy may be much larger on a cluster with 1 GigE links (Figure 9 reports results from a cluster with 10 GigE links and only one disk under test).

Finally, we perform an experiment to evaluate the scalability of read performance of HDFS and PVFS by varying the number of data servers. Unlike Figure 9 that used a cluster with 50 data servers, we vary the number of data servers from 5 to 50 and we configure each data server to also act as a client; for example, the 10 server case shows the configuration with 10 nodes, each acting as a data server and as a client. Figure 10 shows that the read throughput of all layouts scales linearly as the number of data servers increases. Both PVFS with the hybrid layout and HDFS with its random layout outperform PVFS's default round-robin layout because of the benefit of a complete local copy.

Lessons:

- With the vast resources in data-intensive clusters, significant performance benefits are possible if a coupling to disk performance is avoided (when possible).
- Delegating replication to servers (as "proxy clients") can significantly improve parallelism and resource utilization over driving replication from clients when disks are not the bottleneck.
- Changing the cluster file system's layout policy to write one copy "locally" on the same node as the writer has a significant benefit for both read and write throughput.



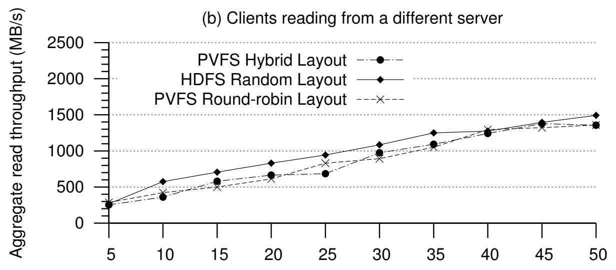


Figure 9: Benefits of keeping a local copy for read performance – ***N-clients concurrently reading from N files from (a) the same server they wrote to, and (b) a server different from the one they wrote to (i.e., non-local copy).***

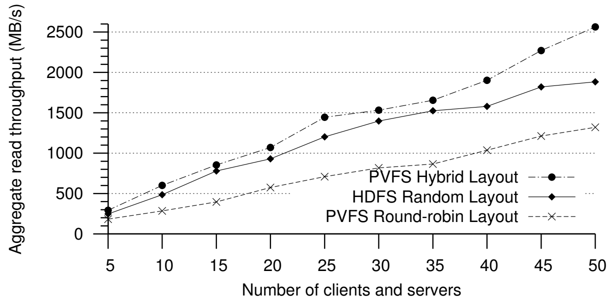


Figure 10: Performance of HDFS and PVFS as a system scales out servers as well as clients for concurrent reading.

4.4 Benchmarks using Hadoop applications

The previous section showed the performance of different layout schemes for applications that use the file system directly; in this section we will evaluate file system performance for Hadoop jobs. That is, each job, written as "map" and "reduce" tasks, is inserted into Hadoop's queues and Hadoop schedules a task per 64 MB of data according to its load balancing policies (quite possibly out of order within the file). We run these jobs on the cluster configurations described earlier in Section 4.3 and use a dataset of fifty 1 GB files (with three copies of each file). Each experiment is run 20 times and we report the average completion time (and its standard deviation) of the Hadoop job.

Figure 11 shows the performance of four Hadoop benchmarks: one write-intensive, two read-intensive and one read-write intensive. The write-intensive benchmark, called *write*, uses Hadoop to write a large dataset using 50 "map" tasks that each write a single 1 GB file to the underlying file system. This data is then processed by two read-intensive and one read-write intensive Hadoop applications. The former consists of a *dummy read* application that simply reads the whole dataset without processing (or generating any output) and a `grep` application that searches for a rare pattern (with a hit rate smaller than 0.0006%) in the dataset, and the latter is a `sort` application that reads the whole dataset, sorts it and writes back the same amount. While the latter three applications use 750 "map" tasks that each processes a single 64 MB chunk, they process the input data in a different manner: the *dummy* read and *grep* applications have zero "reduce" tasks, and *sort* runs with 90 "reduce" tasks to write out the sorted output. We run only one of these three applications at any given time.

Figure 11 shows a large standard deviation in run time of each benchmark, even after 20 iterations. Large runtime variance is common in MapReduce usage [7]. Given that the runtime of most experiments is within one standard deviation of the mean of comparable experiments with other layout schemes, it is not clear that there is much difference in performance when using Hadoop (unlike the significant differences shown in prior sections). There is some indication in Figure 11, however, that the *write* benchmark performs as expected from results in the previous section; by writing the first copy locally and the remaining two writes in a well-balanced round-robin manner, PVFS's hybrid layout allows applications to complete a little faster than if they were using the other two layout schemes. However, the read-intensive benchmarks, both *dummy* read and *grep*, do not exhibit the benefits of local copy and round-robin distribution shown in the previous section. Figure 11 shows that both of these applications run to completion faster when they use HDFS's random layout because Hadoop's job scheduler is able to mask the sub-optimal performance of HDFS's non-uniform file layout by scheduling tasks in a manner that achieves load balancing across all nodes. Note that `grep` runs a bit longer than the *dummy* read application because of the additional processing time required to search the data for the matching pattern. Because the `sort` application writes the same amount of data as it reads, it takes twice as long to complete and PVFS round-robin layout writes more slowly, so it is slower in this test.

Lesson:

- For read-intensive jobs, Hadoop's job scheduler masks the inefficiencies of a layout policy. Coupling job scheduling policies with file system layout strategies can help to balance I/O workloads across the system and significantly improve system utilization.

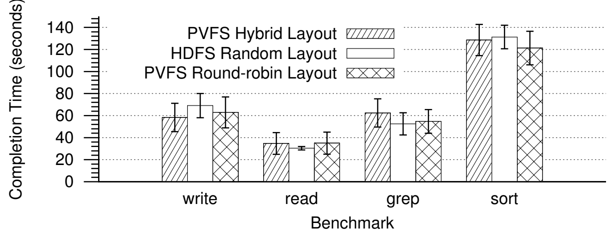


Figure 11: Performance of Hadoop benchmarks – ***On 50 nodes processing 50 one GB files, the standard deviation of run times is large (for 20 runs of each test).***

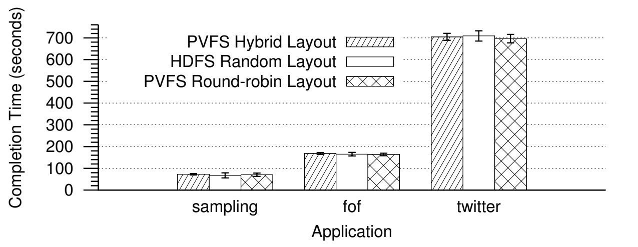


Figure 12: Comparing HDFS and PVFS performance for real Hadoop-based applications (completion time and standard deviation).

	Sampling	FoF	Twitter
Number of map tasks	1,164	1,164	1,504
Number of reduce tasks	124	124	100
Data read from cluster FS (GB)	71.4	71.4	23.9
Data written to cluster FS (GB)	0.002	0.0	56.1
Data read from local temporary files (GB)	0.004	155.7	106.5
Data written to local temporary files (GB)	0.012	140.7	155.6

Table 2: Information about Hadoop applications.

4.5 Performance of Scientific Applications

We compare the performance of PVFS with HDFS using three data-intensive scientific applications programmed to use the Hadoop/MapReduce framework with replication enabled (three copies). These applications were obtained from users of a production Hadoop cluster at Carnegie Mellon University. First two applications, *sampling* and *FoF clustering*, are based on a distributed astrophysics algorithm, called DiscFinder, that identifies large-scale astronomical structures from massive observation and simulation datasets [15]. The main idea is to partition the input dataset into regions, then execute a sequential clustering procedure for each partition, and finally join the clusters across partitions. In the Hadoop implementation of DiscFinder, the *sampling* job determines the appropriate regions and the *FoF clustering* job executes the clustering and joining phases. DiscFinder is designed to operate on datasets with tens of billions of astronomical objects, even when the dataset is much larger than the aggregate memory of compute cluster used for the processing [15]. The third application is a *Twitter analyzer* that processes raw Twitter data into a different format for other tools to operate on [22]. This is primarily an I/O intensive application that uses CPU only to reformat the input data using JSON serialization.

For each application, Table 2 reports the size of the Hadoop job (number of map and reduce tasks) and the amount of I/O performed by these applications using the cluster file system (HDFS and PVFS) and temporary files in the local file system on the node. Figure 12 shows that the performance of PVFS is essentially the same as with HDFS for all three data-intensive applications. *Sampling* is a read-only workload that only reads and processes the data and writes a neglible amount to the file system, while both *FoF clustering* and *Twitter analyzer* read and write twice as much from the local file system than the cluster file system. These results show the HDFS disk bottlenecks that limited scalability in our small (1 GB per node) synthetic experiments are not often exposed to data-intensive applications managed by the Hadoop task scheduler.

5. RELATED WORK

Understanding the similarities and differences between Internet services and HPC is starting to generate significant interest from both researchers and practitioners. Developers of most major parallel file systems have demonstrated the feasibility of using their respective file systems with the Hadoop stack; this includes GPFS [8], Ceph [26], and Lustre [3]. An early version of our work was the first to demonstrate how PVFS can replace HDFS in the Hadoop framework [33]; and this effort was simultaneously reproduced by GPFS [8] and Lustre [3] users. All these efforts focused on showing the benefits of efficient file striping (such as large stripe units and exposing layout distribution) and optimal data prefetching to the Hadoop layer. In comparison, we make several new contributions including PVFS extensions to emulate HDFS-style data layout, replication and consistency semantics, and studying how tradeoffs in HDFS and PVFS affect application performance.

Another project, similar in spirit but from a different direction, proposed to use HPC-style separated storage and compute nodes in the Hadoop framework [28]. This work configured a few HDFS datanodes, designated as Super-DataNodes, to have an order of magnitude more disks than the rest of the datanodes which allowed Hadoop clusters to be configured for workloads that need a much higher ratio of storage to compute nodes.

6. SUMMARY

The last few years have seen an emerging convergence of two data-intensive scalable computing paradigms: high performance computing and Internet services. HPC clusters are beginning to deploy Internet services stacks, such as the Hadoop framework, and their storage systems, such as HDFS and Amazon S3.

This paper explores the relationship between modern parallel file systems, represented by PVFS, and purpose-built Internet services file systems, represented by HDFS, in the context of their design and performance. While both file systems have contrasting deployment models, file access semantics and concurrency mechanisms, we observe that the biggest difference stems from their fault tolerance models and consistency semantics. We show that lightweight middleware and simple file system extensions can be employed to make HPC file systems functionally equivalent to Internet services file systems and operate efficiently in the Hadoop Internet services stack. Our middleware shim layer performs readahead buffering and file layout caching to enable unmodified Hadoop applications to store and access data in PVFS. We also proposed file layout extensions to PVFS that allow our shim to provide HDFS's rack-aware replication functionality in PVFS. Our evaluation, using microbenchmarks and real data-intensive applications, demonstrates that PVFS can perform as well as HDFS in the Hadoop framework. We observed, particularly when the dataset size is comparable to memory size, that writing one copy of data locally when replicating data has a significant performance benefit for both writes and reads.

Acknowledgments

We would like to thank several people who made significant contributions: Robert Chansler, Tsz Wo Sze, and Nathan Roberts from Yahoo! helped diagnose HDFS behavior, Philip Carns from ANL helped debug the `flush()` issues in PVFS, and Bin Fu and Brendan Meeder helped us with their scientific applications and data-sets for evaluation. This work is based on research supported in part by the Department of Energy under award number DE-FC02-06ER25767, by the Los Alamos National Laboratory under contract number 54515-001-07, by the Betty and Gordon Moore Foundation, by the Qatar National Research Fund under award number NPRP 09-1116-1-172, by the National Science Foundation under awards CCF-1019104 and SCI-0430781, and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract DE-AC02-06CH11357. We also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

References

[1] Amazon Web Services - High Performance Computing. `http://aws.amazon.com/hpc-applications/`.

[2] Lustre File System. `http://www.lustre.org`.

[3] Using Lustre with Apache Hadoop. `http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf`, Jan. 2010.

[4] Amazon-EBS. Amazon Elastic Block Storage (Amazon EBS). `http://www.amazon.com/s3`.

[5] Amazon-EC2. Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/ec2`.

[6] Amazon-S3. Amazon Simple Storage Service (Amazon S3). `http://www.amazon.com/s3`.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2010)*, Vancouver, Canada, October 2010.

[8] 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.

[9] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. `http://hadoop.apache.org/core/docs/r0.16.4/hdfsdesign.html`.

[10] 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 *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2006)*, Seattle, WA, USA, November 2006.

[11] Cloudstore. Cloudstore distributed file system (formerly, Kosmos file system). `http://kosmosfs.sourceforge.net/`.

[12] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. Slides from Keynote talk at the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT '2006) on September 18, 2006 in Seattle WA.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2004)*, San Francisco, CA, USA, December 2004.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '2007)*, Stevenson, WA, USA, October 2007.

[15] B. Fu, K. Ren, J. Lopez, E. Fink, and G. Gibson. DiscFinder: A data-intensive scalable cluster finder for astrophysics. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '2010)*, Chicago, IL, USA, June 2010.

[16] S. Ghemawat, H. Gobioff, and S.-T. Lueng. Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '2003)*, Bolton Landing, NY, USA, October 2003.

[17] G. A. Gibson, B. B. Welch, D. B. Nagle, and B. C. Moxon. ObjectStorage: Scalable Bandwidth for HPC Clusters. In *Proceedings of 1st Cluster World Conference and Expo*, San Jose, CA, USA, June 2003.

[18] Hadoop. Apache Hadoop Project. `http://hadoop.apache.org/`.

[19] Hadoop-Docs. The Hadoop Cluster Setup. `http://hadoop.apache.org/core/docs/current/cluster_setup.html`.

[20] Hadoop-S3. Using the Amazon S3 backend on Hadoop. `http://wiki.apache.org/hadoop/AmazonS3`.

[21] N. Horman. Understanding Virtual Memory In Red HatEnterprise Linux 4. Red Hat white paper, Raleigh, NC, 2005, `http://people.redhat.com/nhorman/papers/rhel4_vm.pdf`.

[22] U. Kang, B. Meeder, and C. Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In *Proceedings of the 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD '2010)*, Hyderabad, India, June 2010.

[23] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the 22nd ACM/IEEE Conference on High Performance Computing (SC '2009)*, Portland, OR, USA, November 2009.

[24] S. Lang, R. Latham, R. B. Ross, and D. Kimpe. Interfaces for coordinated access in the file system. In *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER '2009)*, New Orleans, LA, USA, August 2009.

[25] Magellan. Argonne's DOE Cloud Computing: Magellan, A Cloud for Science. `http://magellan.alcf.anl.gov/`.

[26] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. Weil. Ceph as a scalable alternative to the Hadoop Distributed File System. *;login: The USENIX MAGAZINE*, 35(4), August 2010.

[27] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '2007)*, San Jose, CA, USA, February 2007.

[28] G. Porter. Towards Decoupling Storage and Computation in Hadoop with SuperDataNodes. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS '2009)*, Big Sky, MT, USA, October 2009.

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

[30] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '2002)*, Monterey, CA, USA, January 2002.

[31] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '2007)*, San Jose, CA, USA, February 2007.

[32] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '2010)*, Incline Village, NV, USA, May 2010.

[33] W. Tantisiroj, S. V. Patil, and G. Gibson. Data-intensive file systems for Internet services: A rose by any other name ... . Technical Report CMU-PDL-08-114, Carnegie Mellon University, Oct. 2008.

[34] 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 *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '2008)*, San Jose, CA, USA, February 2008.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '2008)*, San Diego, CA, USA, December 2008.