DFS-Perf: A Scalable and Unified Benchmarking Framework for Distributed File Systems

Rong Gu
Qianhao Dong
Haoyuan Li
Joseph Gonzalez
Zhao Zhang
Shuai Wang
Yihua Huang
Scott Shenker
Ion Stoica
Patrick P. C. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-133
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-133.html

July 27, 2016

Acknowledgement

This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

DFS-Perf: A Scalable and Unified Benchmarking Framework for Distributed File Systems

Rong Gu[1], Qianhao Dong[1], Haoyuan Li[2], Joseph Gonzalez[2], Zhao Zhang[2],

Shuai Wang[1], Yihua Huang[1], Scott Shenker[2], Ion Stoica[2], Patrick P. C. Lee[3]

[1]*National Key Laboratory for Novel Software Technology, Nanjing University*
[2]*University of California, Berkeley* [3]*The Chinese University of Hong Kong*
Submission Type: Research

Abstract

A distributed file system (DFS) is a key component of virtually any cluster computing system. The performance of such system depends heavily on the underlying DFS design and deployment. As a result, it is critical to characterize the performance and design trade-offs of DFSes with respect to cluster configurations and real-world workloads. To this end, we present DFS-Perf, a scalable, extensible, and low-overhead benchmarking framework to evaluate the properties and the performance of various DFS implementations. DFS-Perf uses a highly parallel architecture to cover a large variety of workloads at different scales, and provides an extensible interface to incorporate user-defined workloads and integrate with various DFSes. As a proof of concept, our current DFS-Perf implementation includes several built-in benchmarks and workloads, including machine learning and SQL applications. We present performance comparisons of four state-of-the-art DFS designs, namely Alluxio, CephFS, GlusterFS, and HDFS, on a cluster with 40 nodes (960 cores). We demonstrate that DFS-Perf can provide guidance on existing DFS designs and implementations, while adding 5.7% overhead.

1 Introduction

We have witnessed the emergence of parallel programming frameworks (e.g., MapReduce [21], Dremel [30], Spark [39, 40]) and distributed data stores (e.g., BigTable [17], Dynamo [22], PNUTS [19], HBase [6]) for enabling sophisticated and large-scale data processing and analytic tasks. Such distributed computing systems often build atop a *distributed file system (DFS)* (e.g., Lustre [16], Google File System [24], GlusterFS [26], CephFS [37], Hadoop Distributed File System (HDFS) [33], Alluxio (formerly Tachyon) [29]) for scalable and reliable storage management. A typical DFS stripes data across multiple storage nodes (or servers), and also adds redundancy to the stored data (e.g., by replication or erasure coding) to provide fault tolerance against node failures.

There have been a spate of DFS proposals from both academia and industry. These proposals have inherently distinct performance characteristics, features, and design considerations. When putting a DFS in use, users need to decide the appropriate configurations of a wide range of design features (e.g., fine-grained reads, file backup, access controls, POSIX compatibility, load balance, etc.), which in turn affect the perceived performance and functionalities (e.g., throughput, fault tolerance, security, exported APIs, scalability, etc.) of the upper-layer distributed computing systems. In addition, the characteristics of processing and storage workloads are critical to the development and evaluation of file system implementations [35], yet they often vary significantly across deployment environments. All these concerns motivate the need of comprehensive DFS benchmarking methodologies to systematically characterize the performance and design trade-offs of general DFS implementations with regard to cluster configurations and workloads.

Building a comprehensive DFS benchmarking framework is non-trivial, and it should achieve two main design goals. First, it should be *scalable*, such that it can run in a distributed environment to support stress-tests for various DFS scales. It should incur low measurement overheads in benchmarking for accurate characterization. Second, it should be *extensible*, such that it can easily include general DFS implementations for benchmarking for fair comparisons. It should also support both popular and user-customized workloads to address various deployment environments.

This paper proposes *DFS-Perf*, a scalable and extensible DFS benchmarking framework designed for com-

Table 1: Comparison of distributed storage benchmark frameworks

Benchmark Frameworks	Extend to any targeted DFS	Contain workloads from real world	Support to plug in new workload	Provide app trace analysis utility	Can run without computing framework	Support various parallel model	Is aimed to benchmark DFS
TestDFSIO / NNBench	No	No	No	No	No	No	Yes
IOR	No	No	No	No	No	No	Yes
YCSB	Yes	No	Yes	No	Yes	Yes	No
AMPLab Big Data Benchmark	No	Yes	Yes	No	Yes	No	No
DFS-Perf	Yes	Yes	Yes	Yes	Yes	Yes	Yes

prehensive performance benchmarking of general DFS implementations. Table 1 summarizes the key features of DFS-Perf compared with existing distributed storage benchmarking systems (see Section 3 for details). DFS-Perf is designed as a distributed system that supports different parallel test modes at node, process, and thread levels for scalability. It also adopts a modular architecture that can benchmark large-scale deployments of general DFS implementations and workloads. As a proof-of-concept, DFS-Perf currently incorporates built-in workloads of machine learning and SQL query applications. Our DFS-Perf implementation is now open sourced at `http://pasa-bigdata.nju.edu.cn/dfs-perf/`).

We first review today's representative DFS implementations (Section 2) and existing DFS benchmarking methodologies (Section 3). We then make the following contributions.

1. We present the design of *DFS-Perf*, a highly scalable and extensible benchmarking framework (Section 4) that supports various DFS implementations and big data workloads, including machine learning and SQL workloads (Section 5).
2. We use DFS-Perf to evaluate the performance characteristics of four widely deployed DFS implementations, Alluxio, CephFS, GlusterFS, and HDFS, and show that DFS-Perf incurs minimal (i.e., 5.7%) overhead (Section 6).
3. We report our experiences of using DFS-Perf to identify and resolve performance bugs of current DFS implementations (Section 7).

2 DFS Characteristics

Existing DFS implementations are generally geared toward achieving scalable and reliable storage management, yet they also make inherently different design choices for their target applications and scenarios. In this section, we compare four representative open-source DFS implementations, namely Alluxio [1, 29], CephFS [37], GlusterFS [5, 26], and HDFS [7, 33]. We review their different design choices, which also guide our DFS-Perf design. Table 2 summarizes the key characteristics of the four DFS implementations.

Architecture. Alluxio, CephFS, and HDFS adopt a *centralized* master-slave architecture, in which a master node manages all metadata and coordinates file system operations, while multiple (slave) nodes store the actual file data. CephFS and HDFS also support multiple distributed master nodes to avoid a single point of failure. On the other hand, GlusterFS adopts a *decentralized* architecture, in which all metadata and file data is scattered across all storage nodes through the distributed hash table (DHT) mechanism.

Storage Style. CephFS, GlusterFS, and HDFS build on *disk-based* storage, in which persistent block devices (e.g., hard disks). On the other hand, Alluxio is *memory-centric*, and uses memory as the primary storage backend. It also supports *hierarchical* storage which aggregates the pool of different storage resources such as memory, solid-state disks, and hard disks.

Fault Tolerance. For fault tolerance, CephFS and HDFS mainly use replication to distribute exact copies across multiple nodes for fault tolerance. They now also support erasure coding for more storage-efficient fault tolerance. GlusterFS implements RAID (which can be viewed as a special type of erasure coding) at the volume level. In contrast, Alluxio can leverage its under storage systems, in the meantime, it can also adopt lineage and checkpointing mechanisms to keep track of the operational history of computations.

I/O Optimization. All four DFSes use different strategies to improve the I/O performance. CephFS adopts a *cache tiering* mechanism to temporarily cache the recent read and written data in memory, while GlusterFS follows a similar caching approach (called *I/O cache*). On the other hand, both HDFS and Alluxio enforce data locality in computing frameworks (e.g., MapReduce, Spark, etc.) to ensure that computing tasks can access data locally in the same node. In particular, Alluxio supports explicit multi-level caches due to its hierarchical storage architecture.

Exposed APIs. All four DFSes expose APIs that can work seamlessly with Linux FUSE. HDFS and Alluxio export native APIs and a command line interface (CLI) that can work independently without third-party libraries.

Table 2: Comparison of characteristics of Alluxio, CephFS, GlusterFS, and HDFS

DFS	Architecture	Storage Style	Fault Tolerance	I/O Optimization	Exposed APIs
Alluxio	centralized	memory-centric; hierarchical	lineage and checkpoint	data locality; multi-level caches	Native API; FUSE; Hadoop Compatible API; CLI
CephFS	centralized / distributed	disk-based	replication; erasure code (optional)	cache tiering	FUSE; REST-API; Hadoop Compatible API
GlusterFS	decentralized	disk-based	RAID on the network	I/O cache	FUSE; REST-API
HDFS	centralized / distributed	disk-based	replication; erasure code (WIP)	data locality	Native API; FUSE; REST-API; CLI

In particular, both CephFS and Alluxio have the Hadoop-compatible APIs and can substitute HDFS for computing frameworks including MapReduce and Spark.

Discussion. Because of the variations of design choices, the application scenarios vary across DFS implementations. GlusterFS, CephFS, and Lustre [13, 16] are commonly used in high-performance computing environments. HDFS has been used in big data analytics applications along with the wide deployment of MapReduce. Alluxio provides file access at memory speed across cluster computation frameworks. The large variations of design choices complicate the decision making of practitioners when they choose the appropriate DFS solutions for their applications and workloads. Thus, a unified and effective benchmarking methodology becomes critical for practitioners to better understand the performance characteristics and design trade-offs of a DFS implementation.

3 Related Work

Benchmarking is essential for evaluating and reasoning the performance of systems. Some benchmark suites (e.g., [11, 12]) have been designed for evaluating general file and storage systems subject to different workloads. Benchmarking for DFS implementations has also been proposed in the past decades, such as for single-server network file systems [15], network-attached storage systems [25], and parallel file systems (e.g., IOR [10]). Several benchmarking suites are designed for specific DFS implementations, such as TestDFSIO, NNBench, and HiBench [31, 8, 28] for HDFS. To elaborate, TestDFSIO specifies read and write workloads for measuring HDFS throughput; NNBench specifies metadata operations for stress-testing an HDFS namenode; HiBench supports both synthetic microbenchmarks and Hadoop application workloads that can be used for HDFS benchmarking. Instead of targeting specific DFS implementations, we focus on benchmarking general DFS implementations.

Some benchmarking suites can be used to characterize the I/O behaviors of general distributed computing systems. For example, the AMPLab Big Data Benchmark [4] issues relational queries for benchmarking and provides quantitative and qualitative comparisons of analytical framework systems, and YCSB (Yahoo Cloud Serving Benchmark) [14, 20] evaluates the performance of key-value cloud serving stores. Both benchmark suites are extensible to include user-defined operations and workloads with database-like schemas. The MTC (Many-Task Computing) envelope [41] characterizes the performance of metadata, read, and write operations of parallel scripting applications. BigDataBench [36] targets big data applications such as online services, offline analytics, and real-time analytics systems, and provides various big data workloads and real-world datasets. In contrast, DFS-Perf focuses on file system operations, including both metadata and file data operations, for general DFS implementations.

One design consideration of DFS-Perf is on workload characterization, which provides guidelines for system design optimizations. Workload characterization in distributed systems has been an active research topic. To name a few, Yadwadkar et al. [38] identified the application-level workloads from NFS traces. Chen et al. [18] studied MapReduce workloads from business-critical deployments. Harter et al. [27] studied the Facebook Messages system backed by HBase and HDFS as the storage layer. Our workload characterization focuses on DFS-based traces derived from real-world applications.

4 DFS-Perf Design

We present the design details of DFS-Perf. We first provide an architectural overview of DFS-Perf (Section 4.1). We then explain how DFS-Perf achieves scalability (Section 4.2) and extensibility (Section 4.3).

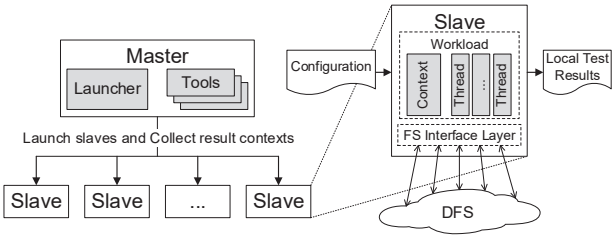


Figure 1: DFS-Perf architecture. DFS-Perf adopts a master-slave model, in which the master contains a launcher and other tools to manage all slaves. For each slave, the input is the benchmark configuration, while the output is a set of evaluation results. Each slave runs multiple threads, which interact with the DFS via the File System Interface Layer.

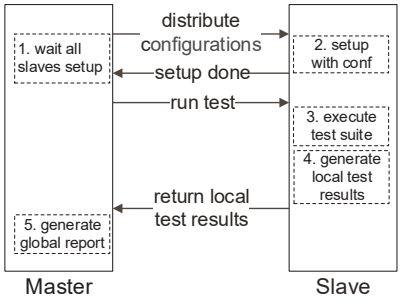


Figure 2: DFS-Perf workflow. The master launches all slaves and distributes configurations of a benchmark to each of them. Each slave sets up and executes the benchmark independently. Finally, the master collects the statistics from all slaves and produces a test report.

4.1 DFS-Perf Architecture

DFS-Perf is designed as a distributed architecture that runs on top of a DFS that is to be benchmarked. It follows a master-slave architecture, as shown in Figure 1. It has a single master process to coordinate multiple slave processes, each of which issues file system operations to the underlying DFS. The master consists of a launcher that schedules slaves to run benchmarks, as well as a set of utility tools for benchmark management. For example, one utility tool is the report generator, which collects results from the slaves and produces performance reports. Each slave is a multi-threaded process (see Section 4.2) that can be deployed on any cluster node to execute benchmark workloads on a DFS.

Figure 2 illustrates the workflow of executing a benchmark in DFS-Perf. First, the master launches all slaves and distributes test configurations to each of them. Each slave performs initialization process, such as loading the configurations and setting up the workspace, and notifies the master when it completes the initialization process. When all slaves are ready, the master notifies them to start executing the benchmark simultaneously. Each slave independently conducts the benchmark test by issuing a set of file system operations, including the metadata and file data operations that interact with the DFS's master and storage nodes, respectively. It also collects the performance results from its own running context. Finally, after all slaves issue all file system operations, the master collects the context information to produce a test report.

4.2 Scalability

To achieve scalability, DFS-Perf parallelizes benchmark executions in a multi-node, multi-process, and multi-thread manner: it distributes the benchmark execution through multiple nodes (or physical servers); each node can run multiple slave processes, and each slave process can run multiple threads that execute the benchmarks independently. The numbers of nodes, processes, and threads can be configured by the users. Such parallelization enables us to stress-test a DFS through intensive file system operations.

To reduce the benchmarking overhead, we only require each slave to issue only a total of *two* round-trip communications with the master, one at the beginning and one at the end of the benchmark execution (see Figure 2). That is, DFS-Perf itself does not introduce any communication to manage how each slave run benchmarks on the DFS. Thus, the DFS-Perf framework puts limited performance overhead on the DFS during benchmarking.

4.3 Extensibility

DFS-Perf achieves extensibility in two aspects. First, DFS-Perf provides a pluggable interface via which users can add a new DFS implementation to be benchmarked by DFS-Perf. Second, DFS-Perf provides an interface via which users can customize specific workloads for their own applications to run atop the DFS.

4.3.1 Adding a DFS

DFS-Perf provides a general *File System Interface Layer* to abstract the interfaces of various DFS implementations. Each slave process interacts through the File System Interface Layer with the underlying DFS, as shown in Figure 1.

One design challenge of DFS-Perf is to support general file system APIs. One option is to make the File System Interface Layer of DFS-Perf POSIX-compliant. But DFS implementations may not support POSIX APIs, which are commonly used in local file systems of Linux/UNIX but may not be suitable for high-performance parallel I/O [23]. Another option is to deploy Linux FUSE inside

DFS-Perf, as it is supported by a number of DFS implementations (see Section 2). However, Linux FUSE adds additional overheads to distributed file system operations. Here, we carefully examine the APIs of state-of-the-art DFS implementations and classify the general file system APIs into four categories, which cover the most common basic file operations.

- **Session Management**: managing DFS sessions, including *connect* and *close* methods;
- **Metadata Management**: managing DFS metadata, including *create*, *delete*, *exists*, *getParent*, *isDirectory*, *isFile*, *list*, *mkdir*, and *rename* methods;
- **File Attribute Management**: managing file attributes, such as the file path, length, and the access, create, and modification time;
- **File Data Management**: the I/O operations that transfer the actual file data. Currently they are via InputStream and OutputStream due to the APIs provided by supported DFSes.

To elaborate, we abstract a base class called *DFS* that realizes the interfaces of above four categories of general DFS operations. The abstract methods of *DFS* constitute the File System Interface Layer in DFS-Perf. To support a new DFS, users only need to implement those abstract methods in a new class that inherit the base class *DFS*. Users also register the new DFS class into DFS-Perf by adding an identifier (in the form of a constant number) to map the new DFS to the implemented class. DFS-Perf differentiates the operations of a DFS implementation via a URL scheme in the form of *fs://*. Note that users need not be concerned about the synchronization issues of APIs, which are handled by DFS-Perf.

Take CephFS as an example. We add a new class named *DFS_Ceph* and implement all abstract methods for interacting with CephFS. Our implementation only comprises less than 150 lines of Java codes. To specify file system operations with CephFS in a benchmark, users can specify the operations in the form of *ceph://*.

Our current DFS-Perf prototype has implemented the bindings of several DFS implementations, including Alluxio, CephFS, GlusterFS, HDFS, as well as the local file system.

4.3.2 Adding a New Workload

DFS-Perf achieves loose coupling between a workload and the DFS-Perf execution framework. Users can simply define a new workload in DFS-Perf by realizing several base classes and a configuration file, as shown in Table 3. The base classes specify the execution logic and the measurement statistics of a workload, while the configuration file consists of the settings of a workload, such as the information about testing data sizes and distributions, the file numbers and locations, and the I/O buffer sizes. All configurations are described in XML format that can be easily configured. Each slave process will take both the base classes and the configuration file of a workload.

Table 3: Components for a workload definition: base classes and a configuration file.

Sub-Component	Meaning
PerfThread	The class that contains all workload execution logic of a thread.
PerfTaskContext	The class that maintains the measurement statistics and running status.
PerfTask	The class that keeps all threads and conducts the initialization and termination work.
PerfGlobalReport	The class that generates the test report from all workload contexts.
configure.xml	The configuration file that manages the workload settings in XML format.

5 Benchmark Design

A practical DFS generally supports a variety of applications for big data processing. To demonstrate how DFS-Perf can benchmark a DFS against big data applications, we have designed and implemented built-in benchmarks for two widely used groups of big data applications, namely machine learning and SQL query applications. For machine learning, we consider two applications: KMeans and Bayes, which represent a clustering algorithm and a classification algorithm, respectively [9]. For SQL queries, we consider three typical query applications: select, aggregation, and join [32].

5.1 DFS Access Traces of Applications

We first study and design workloads of the five big data applications based on their access patterns. Tarasov et al [34] can extract workload models from large I/O traces, including I/O size, CPU utilization, memory usage, and power consumption. However, to sufficiently reflect the performance of applications, DFS-Perf focuses on the DFS-related traces, since these traces show how the applications interact with a DFS. Specifically, we run each application on Alluxio with a customized Alluxio client, which records all operations. We then analyze the DFS-related traces based on the output logs. In our study, the DFS-related access operations can be divided into four categories: sequential writes, sequential reads, random reads, and metadata operations. The first three types of operations are collectively called data operations. Here, a sequential read means reading a file sequentially from the head to the end, while a random read means reading a file randomly by skipping to different locations.

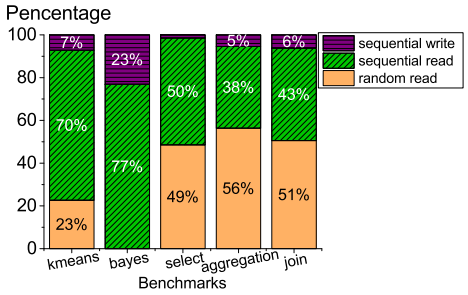


Figure 3: Summary of the five applications' DFS-related access traces. All of them issue more reads than writes, and the proportion of sequential and random read is more balanced in SQL query applications than in machine learning applications.

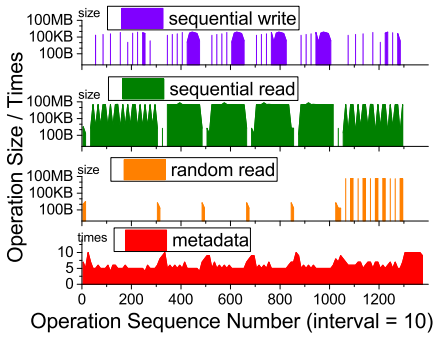
5.1.1 Overview of File Operation Traces

We first analyze the access patterns of the benchmarks. We have collected file operation-related traces, and summarizes the traces of each benchmark in terms of the percentage of read/write operation counts. Figure 3 illustrates that these workloads all issue more reads than writes, although they have different read-to-write ratios. The SQL query workloads have more balanced proportions of sequential reads and random reads than machine learning workloads. Also, the aggregation application has the highest percentage of random reads (up to 56%), while the Bayes application has no random read operation at all.

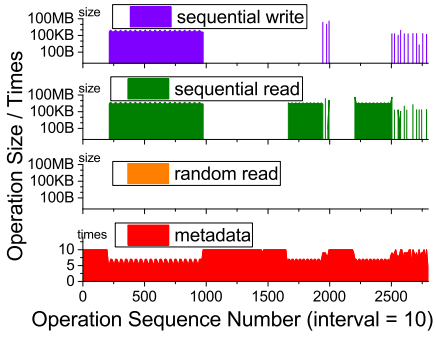
5.1.2 Machine Learning Benchmarks

We analyze the DFS access traces of the machine learning benchmarks in detail. Figure 4 and Figure 5 show the detailed traces. The X-axis is the operation sequence number, each of which corresponds to an operation; the Y-axis is the data sizes of read and write operations, or times of metadata operations. For brevity, in each benchmark, we choose an interval and measure the aggregated operation size or times in each interval. These measurements are affected by various factors in different benchmarks, but we can still summarize the access patterns from their trends.

Figure 4 demonstrates the DFS access traces of the KMeans and Bayes training processes. It is obvious that these data analytic applications access a DFS in an iterative fashion. A training process contains several iterations, each of which reads a variable size of data (ranging from several kilobytes to several gigabytes). In the last round, the result determining the size of write operations is the output. Note that sequential reads appear much more frequently than random reads.



(a) KMeans



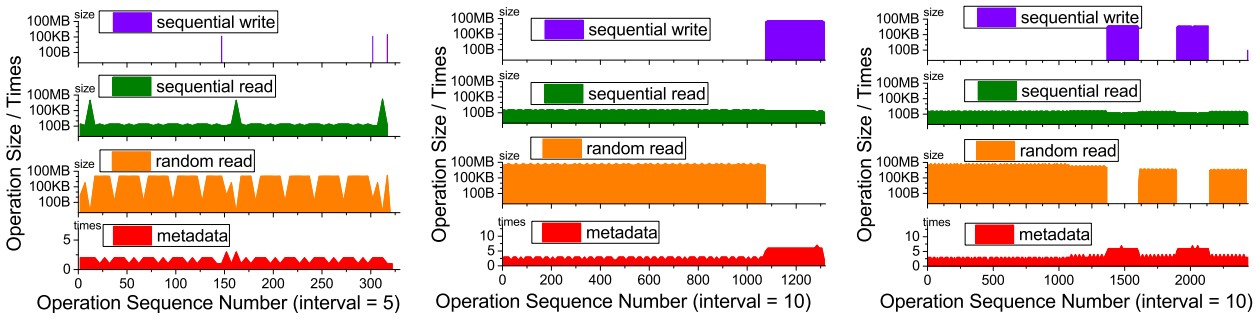
(b) Bayes

Figure 4: Detailed DFS Access Traces of Machine Learning Benchmarks. (a) The KMeans training process is iterative. It reads a lot of data in each iteration and writes more in the end. (b) The Bayes training process can be obviously divided into several sub-processes, each of which is filled with many sequential reads and few sequential writes. Note that there is no random read.

5.1.3 SQL Query Benchmarks

We now analyze the DFS access traces of the SQL Query Benchmarks. Many big data query systems, such as Hive, Pig, and SparkSQL, convert SQL queries into a series of MapReduce or Spark jobs. These applications mainly scan or filter data by sequentially reading data from a DFS in parallel. Also, they use indexing techniques to locate values, and hence trigger many random reads to a DFS.

In Figure 5(a), we find that the select benchmark has at least thousands of times more random reads than sequential reads. However, the result size is small and thus there are only few sequential writes. The aggregation benchmark is more complex than the select benchmark. As shown in Figure 5(b), the aggregation benchmark keeps reading data and executing the computation logic, and finally it writes the result to a DFS. The number of the random reads is still more than that of sequential reads. Also, there is an obvious output process due to the large size of the result. In Figure 5(c), the whole process of the join benchmark can be split into several read and write sub-



(a) Select (b) Aggregation (c) Join

Figure 5: DFS Access Traces of SQL Query Benchmarks. The random read size is always more than the sequential read size. (a) Select: At about the halfway mark and the end of the whole process, there are sequential writes with a larger data size. (b) Aggregation: The sequential writes are concentrated at the end of the whole process. (c) Join: This is the longest process and we can obviously see two sequential write sub-processes.

processes. The read sub-processes are filled with random reads, while the write sub-processes are mixed with sequential reads and sequential writes.

In summary, the SQL query benchmarks generate more random reads than sequential reads. In addition, the reads and writes are mixed with a certain ratio which is determined by the data size.

5.2 Workloads

Based on the collected access traces in Section 5.1, we accordingly design five built-in workloads in DFS-Perf to simulate the distributed I/O characteristics of typical big data applications. Each workload in DFS-Perf has its configurations and execution logic. The configurations make the workload flexible to evaluate testing cases with various scales and loads, while the execution logic represents a set of applications or typical cases. The DFS operators of the workloads include sequential reads, random reads, sequential writes, and metadata operations. Further, we support more sophisticated workloads, which have configurable DFS access patterns of real-world typical applications, such as mixed read/write workloads and iterative read/write workloads.

Metadata Operations Workload: This workload focuses on performing metadata operations such as *create*, *exist*, *rename*, and *delete*. In metadata-centralized DFS, e.g., Alluxio and HDFS, this workload can test the performance of the saturated metadata node. While in metadata-decentralized DFS, e.g., GlusterFS, it can perform a stress test for the correctness and synchronization performance of the whole cluster. Meanwhile, the number of connections is configurable, so that we can use this workload to evaluate the upper limit of concurrency.

Sequential/Random Read/Write Workload: The simple read and write workloads are to sequentially read and write a list of files, respectively. As the basic operations of a file system, they are used to test the throughput of a DFS. In addition, we provide a random read workload to represent the indexing access in databases or the searching access in algorithms. This workload randomly skips a certain length of data and then reads another certain length of data instead of sequentially reading the whole file. Note that a DFS usually provides data streams for reading across the network, and thus the only way for current supported DFSes to perform random skips is to create a new input stream when skipping backward. However, DFS-Perf reserves the randomly reading interface that new DFS can have its own implementation. The write workload is to write content into new files, so it is also the data generator of DFS-Perf with the configurable data sizes and distributions.

Mixed Read/Write Workload: In general, the read-to-write ratio varies across applications. This workload is composed of a mixture of read and write workloads with a configurable ratio. It resembles the real-world applications with heavy reads (e.g., hot data storage like online videos) or heavy writes (e.g., historical data storage like trading information). In addition, mixed read and write workloads are often used to evaluate the cache and eviction performance of a hierarchical DFS.

Iterative Read/Write Workload: The iterative computing pattern is often found in large-scale graph computing and machine learning problems [39]. This workload represents the applications in which the output of the former iteration is the input to the next one. Specifically, we provide two modes called *Shuffle* and *Non-Shuffle* for data accesses. In *Shuffle* mode, each slave process reads data from other nodes; in the *Non-Shuffle* mode, each slave process only reads the files written by itself, which keeps data locality.

Irregular Massive Access Workload: Many applications

have complex read or write patterns, like web servers. The features that we simulate with these applications are randomization and concurrency. In this irregular massive access workload, files are read or written randomly and concurrently. It can reflect the throughput performance of a DFS cluster close to a real-world situation. Moreover, similar to the iterative workload, this workload also has both *Shuffle* and *Non-Shuffle* modes.

5.3 Automatic Workload Generator

In addition to the built-in workloads, DFS-Perf provides users a tool called the *Workload Generator* that can generate specific workloads automatically. Similar to Section 5.1, DFS-Perf provides each supported DFS with a wrapped client implementation that can record all the DFS-related operations. And with a few extra configurations, applications are able to access DFSes with these customized clients. In this way, the workload generator can collect and analyze the DFS-related access traces and statistics for an application that runs on a DFS.

Figure 6 shows how the workload generator works. First, it traces the DFS-related behaviors of the application and logs the intermediate information. It then analyzes the information to produce both the DFS-related statistics and traces. The statistics contain the statistical numbers such as the operation times, total data size, etc. With these statistics, the user can further configure the built-in workloads to match their behaviors to the application. On the other hand, the traces consist of all exact behavior records that represent how the application interacts with the underlying DFS. With these traces, DFS-Perf can generate a new workload which completely replays the DFS-related behaviors of the application, even on different parallel modes. In summary, the workload generator provides customized workloads for DFS-Perf to match the characteristics of real-world applications, so as to measure the pure DFS performance or to compare the performance of different DFS implementations for the same application.

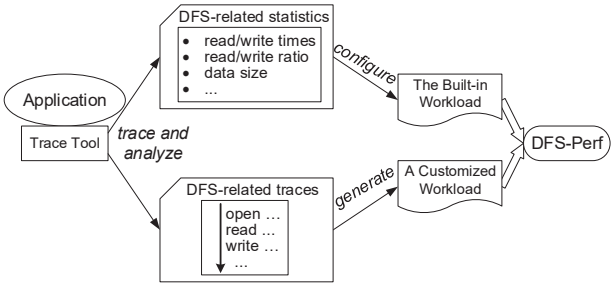


Figure 6: The *Workload Generator* collects DFS-related statistics and traces for an application that runs on a DFS. It outputs built-in workloads that are configurable, or customized workloads that match the application.

6 Evaluation

In this section, we present evaluation results for using DFS-Perf to benchmark several representative implementations, including Alluxio, CephFS, GlusterFS, and HDFS. The highlights are:

- The characteristics of different DFS implementations have considerable impact on the performance. For example, the centralized and decentralized architectures significantly influence the metadata performance. Fault tolerance and I/O optimization characteristics can lead to various degrees of performance gains by orders of magnitude (Section 6.2).
- DFS-Perf is scalable to evaluate the performance upper-bounds of a DFS using different parallel modes. For the evaluated DFS, the throughput difference between the multi-process and multi-thread modes can reach $1.7\times$ (Section 6.3).
- DFS-Perf has an overhead (about 5.7%), which ensures reliable benchmarking performance results. On the other hand, TestDFSIO (another DFS benchmarking tool) shows an increasing overhead (nearly 20%) as the concurrency degree increases (Section 6.4).

6.1 Experimental Setup

We conduct our experiments on a cluster with one master node and 40 slave nodes. The master has two Intel Xeon E5-2660v2 CPUs with total 20 cores (40 hyper-thread cores), while each slave node has two Intel Xeon E5-2620v2 CPUs with total 12 cores (24 hyper-thread cores). In total, the cluster contains 40 nodes with 960 hyper-thread cores for processing file data operations. Each node has 64 GB DDR3 memory and 6 TB SAS RAID0 hard disk. All these nodes are connected with 1 Gb/s Ethernet. Each node runs RHEL 7.0 with Linux 3.10.0, Ext4 file system, and Java 1.7.0.

We deploy Alluxio 1.1.1, CephFS 0.94.6, GlusterFS 3.5.6, Hadoop HDFS 2.7.0, and our DFS-Perf on all nodes. The Ceph MDS (Metadata Server) runs on the master node and each slave node has a Ceph OSD (Object Storage Device) Daemon. For GlusterFS, each slave node runs both the client and server processes, and we configure it with FUSE. HDFS runs the NameNode daemon on the master node and the DataNode daemon on each slave node. Alluxio runs the Master daemon on the master node and the Worker daemon on each of slave nodes. Each Alluxio Worker is configured to have 32 GB RAMFS, accounting for a total of 1280 GB memory across all slave nodes.

Alluxio is a memory speed virtual distributed file sys-

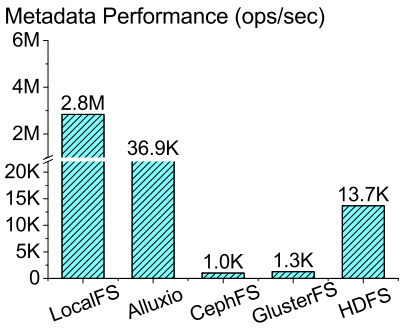


Figure 7: Metadata operation performance, in terms of number of operations per second. For LocalFS, the performance is measured on a single node, while for each DFS, the performance is measured on the whole cluster.

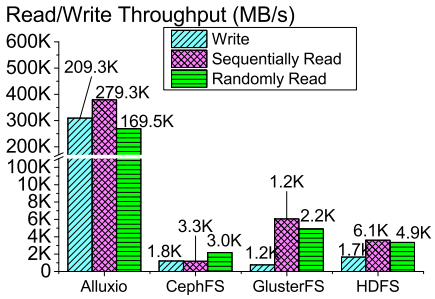


Figure 8: Sequential/random read/write throughput. This is the total throughput of the entire cluster.

tem that allows users to explicitly access data from/to distributed memory in Alluxio's space or the underlying file system which is HDFS in our deployment. The performance of accessing underlying file system by Alluxio is similar to the performance of accessing underlying file system itself. In practice, users mainly reside data in Alluxio's space when using Alluxio. Therefore, we evaluate the performance of Alluxio memory operation in the following experiments. We also take the local file system, denoted as *LocalFS*, for comparison.

6.2 DFS Benchmarking

We apply the built-in workloads in DFS-Perf (Section 5.2) to benchmark different DFS implementations, so as to examine how different DFS characteristics affect the performance. For the multi-thread mode, as each slave has 24 hyper-thread cores, each node runs one process with 24 threads. The performance results are illustrated in Figures 7 to 11.

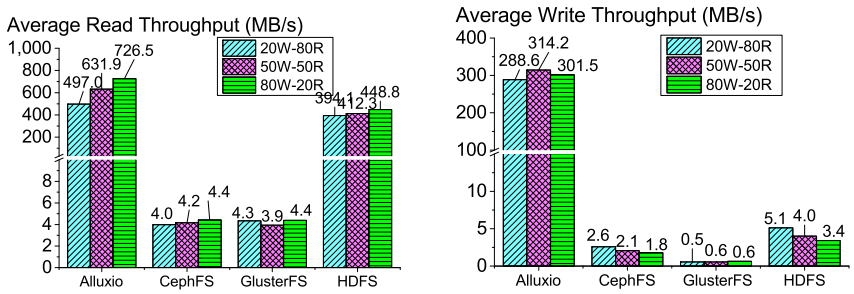
Metadata Operations. Figure 7 illustrates the metadata performance, in terms of the number of operations per second. As expected, LocalFS outperforms others in the metadata performance, since its metadata operations run in single-node memory without any network or disk interaction. In particular, for DFS metadata operations, we observe that the centralized metadata management can achieve an up to around $30\times$ speedup than the decentralized one. GlusterFS has low metadata performance since its decentralized metadata management needs to synchronize metadata information over the cluster. CephFS has the worst performance although it is metadata-centralized, because the Ceph MDS only handles metadata requests and all metadata is stored in several Ceph OSDs on different nodes. In contrast, Alluxio and HDFS achieve higher metadata performance than GlusterFS since they both store and manage metadata in a centralized approach.

Sequential/Random Read/Write. Figure 8 shows the total throughput results of sequential/random read/write operations. We have cleared the OS buffer cache before each test to precisely evaluate the performance of each operation. We observe that the memory-centric DFS implementations can achieve higher throughput than the disk-based ones by more than $100\times$. Specifically, Alluxio reads and writes faster than others since all its operations are in local memory. For the read/write throughput, Alluxio is about two orders of magnitude faster than other storage systems. In addition, the fault tolerance mechanism has significant impact on the performance. To maintain fault tolerance, CephFS, GlusterFS, and HDFS need to replicate data across the cluster, while Alluxio has the option to leverage lineage and checkpoint.

Mixed Read/Write. Figure 9 shows the respective read and write throughput subject to different read-to-write ratios. We did not explicitly clear the page cache before each operation to better simulate the real scenarios for the following workloads. CephFS and HDFS show that the read throughput increases and the write throughput decreases when the write ratio rises. In GlusterFS, both the write and read operations affect the I/O cache regardless of the read-to-write ratio, so the read-to-write ratio is not the major factor to affect the throughput. Alluxio shows an increasing read throughput as the write ratio rises but its write throughput degrades.

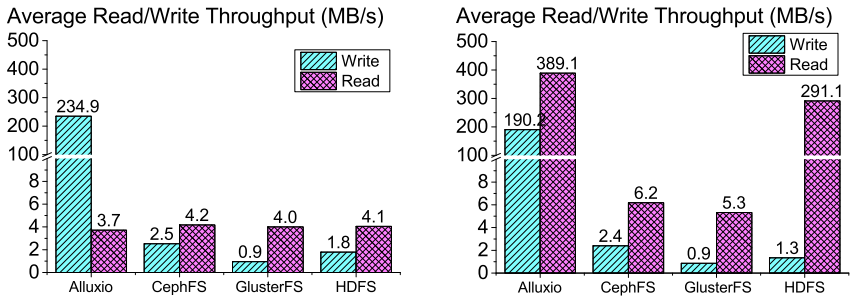
Iterative Read/Write. We now consider the iterative workloads, and the performance results are shown in Figure 10. Compared the *Shuffle* and *Non-Shuffle* modes, data locality can lead to tens to even hundreds times of performance improvement. The read throughput of HDFS in *Non-Shuffle* mode increases by $70\times$ of the *Shuffle* mode, while this value of Alluxio is more than $100\times$. However, for CephFS and GlusterFS, they distribute files instead of storing files locally, so their throughput is limited by the network.

In addition, Alluxio in the *Non-Shuffle* mode reads and writes data through local memory at the same time, which means the read and write may influence each other. This leads to a lower write throughput than in the *Shuffle* mode. However, Alluxio still outperforms others in writes since



(a) Read (b) Write

Figure 9: Mixed read/write throughput. This is the average throughput of each thread.



(a) *Shuffle* (b) *Non-Shuffle*

Figure 10: Iterative read/write throughput in (a) *Shuffle* Mode and (b) *Non-Shuffle* Mode. This is the average throughput of each thread.

it does not transfer data over network.

Irregular Massive Access.

Finally, Figure 11 shows the throughput of irregular massive access. Compared with the iterative workload, this irregular massive workload has similar results. In *Shuffle* mode, all the operations are limited by the network, except the write operation of Alluxio. And in *Non-Shuffle* mode, the read throughput of Alluxio and HDFS increases by about $100\times$ due to their data locality. In addition, the read throughput of Alluxio in Figure 11(b) is about 50% higher than that in Figure 10(b), because the multi-level caches in Alluxio work better for irregular access than the sequentially iterative access.

6.3 Scalability

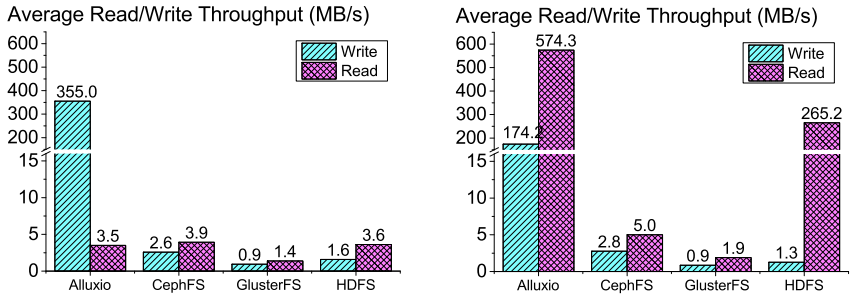
In this subsection, we evaluate the scalability of DFS-Perf by comparing its performance in multi-thread and multi-process parallel modes. We choose the throughput results of Alluxio and HDFS as the representative examples. First, we run the workloads from 1 to 48 threads or processes on a single node. Then the same workloads are run from 1 to 40 nodes, either in 24 threads or 24 processes (24 is the hyper-threading upper limit). The speedup performance is shown in Figure 12.

In multi-node mode, the throughput of HDFS is limited by the cluster network bandwidth, while Alluxio achieves near-linear scalability because it accesses data from local memory. The experimental results in Figure 12 also indicate that DFS-Perf has good scalability performance. Furthermore, this experiment can also be used for performing stress tests in a scalable way to detect the bottleneck of a DFS.

Meanwhile, for the multi-process and multi-thread modes, we find that their behaviors are similar. The throughput grows with the increase of the concurrency, until the concurrency degree reaches 24, i.e., the hyper-threading upper limit in each of our machines. When the concurrency degree goes beyond the hyper-threading upper limit, the throughput performance stays the same or even decreases a bit. In addition, HDFS performs better in the multi-thread mode (about $1.5\times$) while Alluxio performs better in the multi-process mode (about $1.7\times$) on a single node.

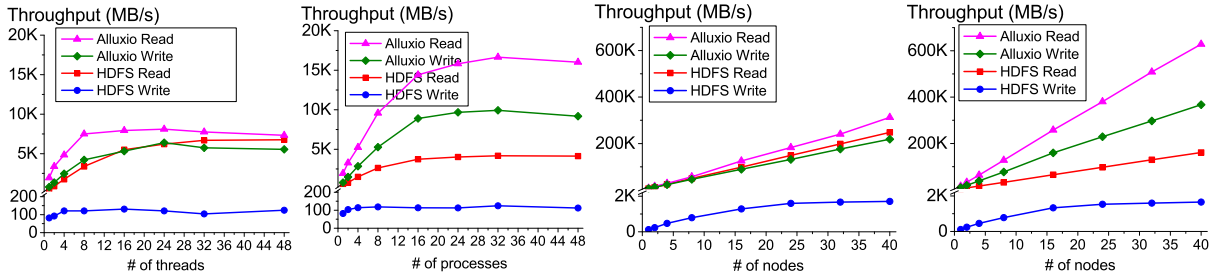
6.4 Framework Overhead

In this subsection, we evaluate the overhead of the DFS-Perf framework for benchmarking. The overhead may come from the few communications between DFS-Perf Master and Slaves, and the extra statistics collecting step. And we need to know how much the overhead impacts the



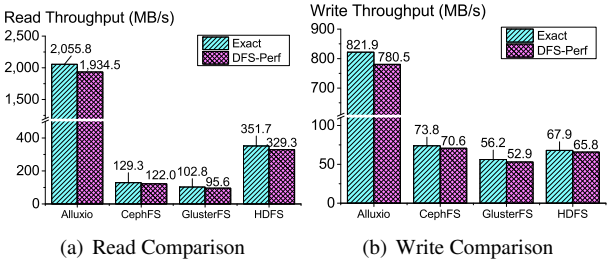
(a) *Shuffle* (b) *Non-Shuffle*

Figure 11: Irregular massive access throughput in (a) *Shuffle* Mode and (b) *Non-Shuffle* Mode. This is the average throughput of each thread.



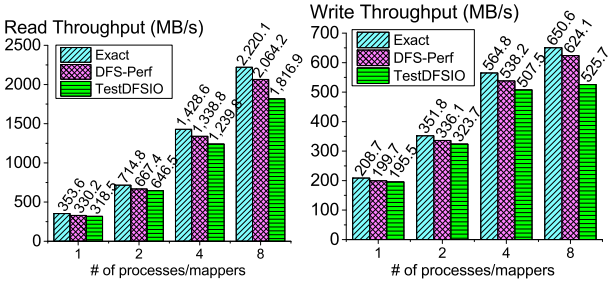
(a) Single-Node Multi-Thread (b) Single-Node Multi-Process (c) Multi-Node Multi-Thread (d) Multi-Node Multi-Process

Figure 12: Scalability of DFS-Perf. In single-node we take the value of 1 process x 1 thread as the baseline and in multi-node we take the value of one node as the baseline. (a) single-node multi-thread. (b) single-node multi-process. (c) multi-node, each node is in 16 threads. (d) multi-node, each node is in 16 processes.



(a) Read Comparison (b) Write Comparison

Figure 13: Comparisons between the Exact and DFS-Perf. (a) Read Throughput. (b) Write Throughput.



(a) Read Comparison (b) Write Comparison

Figure 14: Comparisons of the Exact, DFS-Perf and TestDFSIO. (a) Read Throughput. (b) Write Throughput.

performance and whether it will get higher when scaling out.

For comparison, we hard-coded a lightweight tool that directly reads and writes DFS to gather the exact throughput performance metric without any impact on the benchmarking framework itself, namely *Exact* in Figure 13 and Figure 14. First, we measure the overhead of the DFS-Perf framework by comparing its performance with the exact performance. As shown in Figure 13, **DFS-Perf only has 5.7% (3% - 7%) difference on average**.

Then, we compare the overhead of DFS-Perf with TestDFSIO, a built-in test tool of HDFS. For fair comparisons, we conduct these experiments on the same DFS, a single node HDFS. Both DFS-Perf and TestDFSIO are configured with the same number of processes or mappers. Figure 14 reveals that DFS-Perf has less overhead and achieves more accurate results than TestDFSIO. Moreover, DFS-Perf has a stable difference (about 6.6% on read and 4.4% on write), but TestDFSIO has an increasing difference as the concurrency gets higher (from 9.9% to 12.7% on read and from 6.3% to 19.2% on write). The reason is that TestDFSIO relies on the MapReduce frame-

work, a general parallel processing platform, in which more mappers consume more system and network resources than the DFS-Perf implementation.

7 Experience

We discovered several performance issues in DFS while using DFS-Perf to evalute them. We have contributed some of our patches to open source communities. For example, the DFS-Perf sequential read benchmark has detected that, in Alluxio, there exists critical overhead in the open and close steps when many clients access the file metadata concurrently [3]. Another example is that DFS-Perf helped us detect a memory statistics bug in Alluxio [2]. The statistics are important for monitoring and allocating the storage space in Alluxio. This bug only takes effect when multiple users read the same in-memory file concurrently. Thus, the sequential utility tools and unit tests in Alluxio can hardly capture it. Because of DFS-Perf's multi-thread and multi-process testing mechanism, we detected the bug easily. In addition, the *alluxio-perf* module in Alluxio is derived from our DFS-Perf work and now has become an important performance benchmarking tool of the Alluxio project.

8 Conclusion

A distributed file system is a key component of virtually any cluster computing system. We believe that comprehensive performance evaluation of various DFS implementations is important. In this paper, we present DFS-Perf, a benchmarking framework that can be designed for evaluating the performance of various DFS implementations. DFS-Perf is scalable and general enough to adapt to various parallel and distributed environments. Also, it is extensible that users can add new workloads and plug in new DFS backends. We have also designed and implemented a group of representative workloads with the file access patterns summarized from the real-world applications. Moreover, DFS-Perf includes an extensible workload generator that enables users to customize specific workloads automatically. We have conducted extensive experiments to evaluate state-of-the-art DFS implementations using DFS-Perf. Based on the evaluations, we discussed the critical factors that impact the performance of DFS. The experimental results also demonstrate that DFS-Perf has good scalability performance and negligible overhead (5.7% on average).

References

[1] Alluxio. http://alluxio.org/.

[2] Alluxio bug fix of the memory usage statistics. https://github.com/Alluxio/alluxio/pull/354.

[3] Alluxio scalability issue in reading from under fs. https://alluxio.atlassian.net/browse/ALLUXIO-257.

[4] AMPLab Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[5] GlusterFS. http://www.gluster.org/.

[6] HBase. http://hbase.apache.org/.

[7] HDFS Architecture. http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[8] HiBench. https://github.com/intel-hadoop/HiBench.

[9] Mahout. http://mahout.apache.org/.

[10] Parallel filesystem I/O benchmark. https://github.com/chaos/ior.

[11] Solaris FileBench. http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[12] Storage Performance Council. http://www.storageperformance.org/home.

[13] Why use Lustre. https://wiki.hpdd.intel.com/display/PUB/Why+Use+Lustre.

[14] YCSB. https://github.com/brianfrankcooper/YCSB/.

[15] R. Bodnarchuk and R. Bunt. A synthetic workload model for a distributed system file server. In *Proceedings of ACM SIGMETRICS*, 1991.

[16] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system, 2002.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[18] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC, Indianapolis, Indiana, USA, June 10-11*, pages 143–154, 2010.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation, OSDI, San Francisco, California, USA, December 6-8*, pages 137–150, 2004.

[22] 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, Stevenson, Washington, USA, October 14-17*, pages 205–220, 2007.

[23] P. M. Dickens and J. Logan. Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC, Garching, Germany, June 11-13*, pages 31–38, 2009.

[24] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP, Bolton Landing, NY, USA, October 19-22*, pages 29–43, 2003.

[25] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of ACM SIGMETRICS*, 1997.

[26] Gluster. An Introduction to Gluster Architecture. White Paper, 2011.

[27] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A facebook messages case study. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST, Santa Clara, CA, USA, February 17-20*, pages 199–212, 2014.

[28] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE, Long Beach, California, USA, March 1-6*, pages 41–51, 2010.

[29] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC, Seattle, WA, USA, November 3-5*, pages 1–15, 2014.

[30] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[31] M. Noll. Benchmarking and stress testing an Hadoop cluster with TeraSort, TestDFSIO & Co. `http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-clust` 2011.

[32] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Providence, Rhode Island, USA, June 29 - July 2*, pages 165–178, 2009.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST, Lake Tahoe, Nevada, USA, May 3-7*, pages 1–10, 2010.

[34] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST, San Jose, CA, USA, February 14-17*, page 22, 2012.

[35] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):5:1–5:56, 2008.

[36] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA, Orlando, FL, USA, February 15-19*, pages 488–499, 2014.

[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, USA, November 6-8*, pages 307–320, 2006.

[38] N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of application workloads from network file traces. In *8th USENIX Conference on File and Storage Technologies, FAST, San Jose, CA, USA, February 23-26*, pages 183–196, 2010.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI, San Jose, CA, USA, April 25-27*, pages 15–28, 2012.

[40] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, Farmington, PA, USA, November 3-6*, pages 423–438, 2013.

[41] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. T. Foster. MTC envelope: defining the capability of large scale computers in the context of parallel scripting applications. In *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC, New York, NY, USA, June 17-21*, pages 37–48, 2013.