The 34th International Conference on Distributed Computing Systems, ICDCS 2014,
June 30 - July 3, 2014, Madrid, Spain.

Will They Blend?: Exploring Big Data Computation atop Traditional HPC NAS Storage

Ellis H. Wilson III and Mahmut T. Kandemir
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
Email: {ellis,kandemir}@cse.psu.edu

Garth Gibson
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: garth@cs.cmu.edu

Abstract

The Apache Hadoop framework has rung in a new era in how data-rich organizations can process, store, and analyze large amounts of data. This has resulted in increased potential for an infrastructure exodus from the traditional solution of commercial database ad-hoc analytics on network-attached storage (NAS). While many data-rich organizations can afford to either move entirely to Hadoop for their Big Data analytics, or to maintain their existing traditional infrastructures and acquire a new set of infrastructure solely for Hadoop jobs, most supercomputing centers do not enjoy either of those possibilities. Too much of the existing scientific code is tailored to work on massively parallel file systems unlike the Hadoop Distributed File System (HDFS), and their datasets are too large to reasonably maintain and/or ferry between two distinct storage systems. Nevertheless, as scientists search for easier-to-program frameworks with a lower time-to-science to post-process their huge datasets after execution, there is increasing pressure to enable use of MapReduce within these traditional High Performance Computing (HPC) architectures.

Therefore, in this work we explore potential means to enable use of the easy-to-program Hadoop MapReduce framework without requiring a complete infrastructure overhaul from existing HPC NAS solutions. We demonstrate that retaining function-dedicated resources like NAS is not only possible, but can even be effected efficiently with MapReduce. In our exploration, we unearth subtle pitfalls resultant from this mash-up of new-era Big Data computation on conventional HPC storage and share the clever architectural configurations that allow us to avoid them. Last, we design and present a novel Hadoop File System, the Reliable Array of Independent NAS File System (RainFS), and experimentally demonstrate its improvements in performance and reliability over the previous architectures we have investigated.

I. Introduction

Cluster computing specialized for processing massive virtual and physical sensor data, one definition of the emerging Big Data vertical, arose from internet services computing, especially the MapReduce [16], Google File System [19], and BigTable [15] tools and their open source siblings, Hadoop [4], its distributed file system (HDFS) [11], and HBase [14], respectively. These systems were developed with a specific system model: identical cost-optimized nodes containing all the compute and storage available to the cluster, simplified semantics tailored to target applications, and the expectation of frequent failures [19]. With this heritage, interoperation with systems and tools from other environments such as high performance computing (HPC) can not be taken for granted, including traditional HPC storage. Because HPC computing systems are of comparable scale to Big Data clusters, it is particularly interesting to be able to support both types of applications using existing HPC storage for convenience and load sharing, if not consolidation for lower associated costs.

With the emergence of resource allocators like Mesos [21] and Yarn [2], a Big Data cluster can dynamically distribute resources between different parallel program schedulers such as Hadoop or HPC's ubiquitous Message Passing Interface (MPI) tools [18]. This enables a sharing of clusters arising from the needs of internet services and those arising from the needs of high performance computing. Switching a set of nodes from executing Hadoop programs to executing MPI programs is easy; its just stopping and launching a set of user-level binaries. However, storage solutions for Big Data frameworks differ widely from that of traditional HPC. For example, HDFS stores write-once files that can only have one writing process, while HPC parallel file systems such as PVFS [22], Lustre [6], GPFS [26], and PanFS [28] support concurrent writes to the same file from thousands of processes. Further, HDFS was designed to store all data in the local disks of compute nodes, using replication for fault tolerance, and to interface to its servers through library (Java class) plugins. Parallel file systems, on the other hand, typically store all data in external storage systems, using RAID erasure coding for fault tolerance, and access servers through a Virtual File System (VFS) kernel module in each host. Therefore, if data is stored in the native format of one, it is not easily accessible to the other and copying is required between the storage mediums; with terabytes to petabytes of data the copy operation itself, not to mention the egregious amounts of wasted capacity, becomes prohibitively expensive.

A. The NAS and HPC Narrative

Nevertheless, due to the attractiveness of the solutions in the Big Data space, many organizations have acquired or put aside a separate set of nodes for exclusively Hadoop compute and storage and have suffered through the cost of capacity waste and expensive copies. Doing so may not be economically or tractably possible due to the huge datasets in traditional

HPC NAS storage, but even if it were, the scattering of storage throughout the compute nodes comes with a number of drawbacks, which motivate our effort to seek consolidated compute for HPC and Big Data computing atop a NAS system:

- *Loss of Infrastructure Consolidation:* Unlike traditional POSIX filesystems, it is non-trivial to execute a variety of applications on HDFS without adapting them to its specific semantics and interface.
- *Forced Import/Export:* Sharing data between HDFS and traditional storage requires an import or export, wasting storage and network resources.
- *I/O Performance Degradation:* For more typical workloads I/O performance degrades since Hadoop is tuned for performance on large datasets.
- *Loss of High-Availability:* The Hadoop NameNode is a single point of failure and requires administrative intervention when downed.
- *No Modification of Files:* It is impossible to modify previously written data, as HDFS is a write-once-read-many distributed file system.
- *Inefficient Compute-Storage Coupling:* When an HDD fails, system administrators may be forced to power down the node or at least restart Hadoop services, resulting in loss of computational resources.

While we argue these reasons give credence to even considering NAS for use under MapReduce, we admit there are other approaches possible in exploring solutions to this problem, such as adapting Hadoop to more efficiently handle scientific problems as recent research has attempted [12], [13]. Therefore, it is important to pause here and clarify that we attack it from the very specific narrative that existing traditional HPC architecture is in-place already. Specifically, we make the following key assumptions: First, the HPC compute is already in place and tuned to efficiently operate with discrete, high-performance NAS storage. Second, the HPC applications developed over many years, which run on these systems have been written in a combination of C or Fortran and MPI to maximize performance for their extensive executions and presume POSIX or near-POSIX semantics, making wholesale transition to MapReduce and a pure Hadoop environment highly intractable. We argue these assumptions reflect the reality of most supercomputing centers in the world today.

This narrative and its assumptions are critical to point out so that it is clear experimentally comparing MapReduce atop NAS against traditional Hadoop installations would be both inappropriate and discordant with our theme – we are not attempting to prove traditional HPC nor a converged architecture is the way of the future for HPC. *Our goal in this work is to show, if you presume the existence of high-performance NAS storage and compute nodes with limited storage, that using a Big Data computation framework such as Hadoop atop this NAS storage is not only possible, but, if cleverly configured, can be efficient and thereby expedite time-to-science for post-execution analytics.*

B. Contributions

In this work we seek to explore possible architectures that allow Hadoop MapReduce to run alongside traditional POSIX applications and against a consolidated storage system provided via NAS. Our **first contribution** towards this effort is a thorough exploration of the following three architectural arrangements that use existing software solutions to accomplish the aforementioned goal:

1) *HDFS as a Client Service*: As HDFS employs node-local VFS to store chunks, reconfiguration can replace node-local storage with remote NAS storage.
2) *HDFS as a Wire Protocol*: Alternatively, because HDFS is a client-server model with a network protocol for all communication, it could also be treated as a Network-Attached Storage (NAS) protocol like NFS [17] or CIFS [20], with the server running within the NAS system itself.
3) *No HDFS*: While HDFS requires MapReduce applications to efficiently operate on its data, MapReduce will efficiently operate on non-HDFS data if configured correctly, so skipping HDFS entirely and going directly to NAS is possible.

In each of the above architectures we thoroughly analyze the impact the arrangement has on *reliability* and experimentally demonstrate the effect that each has on *application performance*. Finding the above solutions satisfactory yet suboptimal, our **second contribution** is the design and development of a new Hadoop FileSystem interface class. Note that Hadoop applications perform I/O via a FileSystem class which can be replaced. Further, alternative FileSystem implementations to HDFS are available for communicating to Amazon S3 [1], CloudStore [5] and PVFS [27].

Our novel file system, the *Replicating Array of Independent NAS File System (RainFS)*, overcomes all of the major issues we discovered. As we had done with the three more standard approaches above, we analyze and evaluate RainFS along the dimensions of reliability and performance. We experimentally demonstrate the performance advantages of RainFS to be as high as 127% for write-intensive workloads and 217% for read-intensive workloads when solely utilizing erasure-coding reliability mechanisms. This superiority continues when we combine replicating and erasure coding reliability mechanisms, demonstrating as high as 254% for write-intensive workloads and 210% for read-intensive workloads. In all tests performed, RainFS performed as well or better than all other architectures tested, while providing reliability guarantees exceeding any of the architectures which perform comparably to it.

II. Background

A. Overview of HDFS

When Google introduced its MapReduce framework [16] in 2004, a restructuring of large scale data analysis was spurred with the most notable open-source implementation of Google's framework being Hadoop [4]. The Hadoop sub-projects we utilize in this work are Hadoop MapReduce, Hadoop YARN, and the Hadoop Distributed File System (HDFS). MapReduce allows users to write parallel programs that are automatically broken into Map and Reduce tasks and executed in parallel within a distributed environment. YARN allows multiple resource management and scheduling mechanisms to co-exist (MapReduce, MPI, etc.) in one managed cluster. HDFS, taking

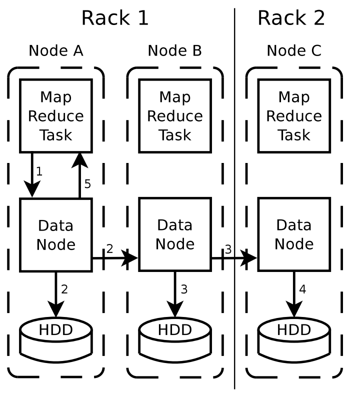


Fig. 1: Flow of I/O from MapReduce tasks, through HDFS, and eventually to each nodes local HDD.

its initial specification from the Google File System [19], makes distributed storage accessible to MapReduce programs, and is tuned to perform well for local hard-disk-drives (HDDs) in the same cluster as the computation. This architecture is now referred to as *Converged Storage*. Note that converged storage breaks from previous architectures that separated processing from storage, normally taking the form of commercial RDBMS or MPI atop NAS systems.

Hadoop MapReduce and HDFS have been shown to scale to thousands of nodes, millions of files and petabytes of storage [8]. HDFS provides double-disk failure tolerance via file replication, out-of-band metadata access, is designed in Java for platform independence, supports Unix-style file permissions, and automatic capacity balancing between nodes. Hadoop has been adopted by many organizations, some of the most notable being Yahoo!, Facebook, Twitter, and LinkedIn.

1) Replication in HDFS: HDFS achieves double-disk failure tolerance by replicating a file across multiple nodes (and therefore distinct HDDs). The process, shown in Figure 1, proceeds in the following manner: First, contact the NameNode for file creation (for clarity, communications with the NameNode are excluded from all diagrams). A response will provide the locations to write each copy and requesting Node A will begin concurrently writing its first copy to the local disk and its second copy over the network to Node B. The NameNode will attempt to specify a rack-local Node B to take advantage of higher-performance intra-rack communication, and specifies a rack-remote Node C to provide rack-failure tolerance. Node B concurrently writes to its local disk and pipelines that replica to the last destination, Node C, referred to as *replication pipelining* in HDFS. It is intended to share the replicating work with another node (B) and decrease the total time to replicate.

III. Architectures Explored

We evaluate three architectural options regarding where the DataNode (DN) daemons are located; we examine running them on the client node as in typical Hadoop usage, running them on the nodes within the NAS system (as if HDFS were a wire-protocol like NFS or CIFS), and bypassing DN daemons altogether, as shown in Figure 2.

The simplest manner in which one can utilize the Hadoop framework with NAS is to specify NAS mount points instead of paths to local storage. As can be seen in Figure 2a, this

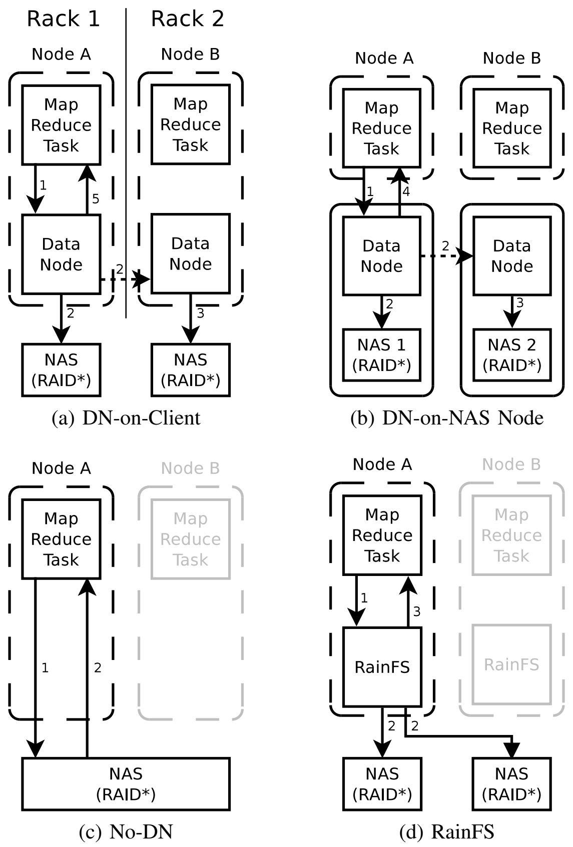


Fig. 2: Flow of writes in the different architectures we explore, shown with a replication level of 2 for all but Figure 2c, which relies on internal redundancy. Dashed arrows signify communication that is purely network overhead, and grayed out nodes indicate that they have no role in the writing of the file from Node A.

architecure is the most similar to the traditional HDFS setup shown in Figure 1 since the DN daemon still runs within a client node. Hadoop will attempt to write to and read from these paths and, finding that it can do so, will happily begin to use it as if it were a local drive. However, as a side-effect of HDFS believing this is a local drive, one will have to make sure to provide unique paths within the mount-folder for each node to use or else nodes may accidentally corrupt each other's files. As we later show, while easy to setup and get started with, this method has serious reliability and performance short-comings.

The second manner we explore using Hadoop on NAS is to move the DN out from each client and to run it directly on NAS nodes. In this architecture we specify, within the NAS nodes, paths to the mount-point(s) for the remaining NAS slaves so that incoming data to the node can be sent via these mounts to the individual storage nodes. This data flow is depicted in Figure 2b. However, forcing all of the data both for reads and writes through these nodes creates serious performance bottlenecks, as we experimentally demonstrate later.

Our third option is to completely avoid using HDFS, directing MapReduce tasks to access the underlying NAS mounts as if locally available to the system. One large caveat is demarcated in Figure 2c – the underlying NAS storage must either be a single system or multiple systems that provide federating services, such that the exposed namespace is unified and the NAS systems transparently achieve striping among themselves. This requirement is an artifact of MapReduce jobs being designed to operate on a single path, rather than a series of paths because HDFS is presumed to be in use. In order to fully remove HDFS, these NAS systems must expose a single namespace, provide reliability mechanisms, and provide load and capacity balancing.

IV. Reliability Analysis

Reliability guarantees and the means by which these guarantees are kept vary between NAS systems and HDFS. In this work we consider the following fault model:

- **Failure of a Disk**: The disk is the most basic unit of storage, and is the most common part to fail.
- **Failure of a Rack**: Inability to get to an entire rack (e.g. top-of-rack network failure) or destruction of an entire rack of storage leads to the transient or permanent failure of an entire rack, and storage systems must provision for such failures.

To ease discussion, we use the term *failure domain* to refer to the range of physical resources that lose data upon failures greater than what it can tolerate. The trade-offs when considering the size of a failure domain tends to be a performance/reliability trade-off; large domains (across many NAS systems) allows for high-bandwidth access to single files since many disks can simultaneously help in the access, whereas small domains (perhaps one domain per NAS) reduces the chances of concurrent failure but have segmented namespaces and therefore fewer disks can help for a single file access.

A. Failure in NAS

To continue to operate upon failure of one or more disks in a NAS system, RAID (i.e. erasure coding) is typically employed, which can recover from a defined maximum number of concurrent disk failures based on the RAID level.

To handle failure of network and power components of the storage system that would otherwise result in inaccessibility of an entire rack, NAS systems employ redundant hardware (e.g., redundant network interface cards and power supply units).

B. Failure in Hadoop

HDFS provides tolerance to individual disk failure in a similar but nuanced manner when compared to traditional erasure coding done in NAS systems. In effect, HDFS fault tolerance mimics declustered RAID1 with copies total. By replicating every file created on distinct HDDs, this assures that upon failure of a drive another copy will remain. The location and health of all replicas is managed by the NameNode.

To handle failures of an entire rack (caused by inaccessibility or physical damage) HDFS still employs replication, but it relies on knowledge of the topology of the HDDs to assure resilience. By assuring that at least one replica exists in a separate rack than the original copy, HDFS provides single-rack failure tolerance. The specific layout and flow of these replicas was diagrammed and can be referenced in Figure 1.

C. Combining the Architectures

We now consider how each of the proposed architectures placed atop NAS handle failure. When we report "failure tolerance of X disks", we are referring to the *maximum X which can be tolerated, no matter which specific X disks fail.*

Let us first consider placing the DN on the client node and configuring paths to NAS mounts. As mentioned, HDFS assumes each time it copies a file, it is in a totally separate disk (and therefore, failure domain), which will not be true if each client node sees paths to all NAS systems. When given multiple paths a DN will randomly select one of the paths for writing the file to load balance as this normally doesn't matter (typically each path is a discrete, local HDD). However, when DN-on-client points at NAS paths, we risk sending multiple replicas to the same NAS.

However, we discovered that achieving a replication level of two is safely possible by giving all of the clients in a given rack access solely to the single NAS system in the same rack (and thereby a single failure domain), and providing HDFS the topology of the system such that it knows all those clients are in the same rack. HDFS will therefore immediately attempt to make the second replica outside of the rack, assuring that the two copies are in two separate NAS systems. However, at triplication and beyond, Hadoop will attempt to create a rack-local second replica, and therefore duplication is the highest level of replication possible when using HDFS on NAS. This limits the possible reliability schemes as presented in Table I.

Moving to the DN on the NAS node architecture, one will see that this no longer suffers from the replication level ceiling restriction as discovered with the last architecture. This is a result of placing a single DN on each NAS node; since we only have a single NAS system for each rack, there is a one-to-one mapping of DN to rack and therefore accidental duplication to the same rack becomes impossible. This allows for replication up to the number of NAS systems, achieving a comparatively wider range of reliability scenarios, shown in Table I.

Last, examining the impact of guiding MapReduce to operate directly on the underlying NAS mount-point, we are faced with a much different situation since the NAS systems must be federated in order for this architecture to work. Because the NAS units are federated, RAID-5 will only provide single-disk tolerance across all of them, and RAID-6 similarly only provides double-disk tolerance. Further, without HDFS we lose replication, so we have no way to tolerate rack failure or ensure availability in the face of network, power, or some other fault in a rack that causes it to go offline.

D. Why Not Just NAS?

Finally, we recognize that many supercomputing systems may (and should) be architected to be reliable based on the guarantees provided by the parallel file systems and NAS alone. While this may be the case, and in which case utilizing the No-DN architecture may be the best choice, two potential use-cases exist for layering Hadoop replication on existing NAS RAID: First, doing so dramatically increases the reliability and up-time for files in an existing system that may only otherwise provide RAID5; higher reliability may be desired for the post-processing analytics so that chance of data loss

	RAID 5	RAID 6	RAID 5	RAID 6	RAID 5	RAID 6
	Repl. 1	Repl. 1	Repl. 2	Repl. 2	Repl. 3	Repl. 3
DN-on-Client	1 / 0	2 / 0	3 / 1	5 / 1	– / –	– / –
DN-on-NAS Node	1 / 0	2 / 0	3 / 1	5 / 1	5 / 2	8 / 2
No HDFS	1 / 0	2 / 0	– / –	– / –	– / –	– / –
RainFS	1 / 0	2 / 0	3 / 1	5 / 1	5 / 2	8 / 2

TABLE I: The number of concurrently failed disks or racks (shown in disk / rack format) that a given architecture can tolerate *without data loss*. Considered for all combinations of typical replication and RAID levels, and dashes (– / –) used to indicate an architecture cannot operate at this reliability level.

for these final results is not only absolutely minimal, but also not tethered to the capabilities of the system to rebuild quickly. Second, layering Hadoop with replication on NAS also may make sense for smaller supercomputing centers whom may utilize NAS systems from multiple vendors. With such discrete namespaces, this would prevent the No-DN architecture from operating, and therefore layering Hadoop in-between would enable one to execute a MapReduce task to concurrently utilize all of the discrete NAS pools.

V. Data Locality and Transport

In this section we consider the impact of locality changes when moving away from a traditional node-local Hadoop configuration to a remote storage solution.

A. *Write Transport*

Figure 2 provides an overview for each of the considered architectures on how writes flow from MapReduce tasks to NAS. When the DN runs on the clients as shown in Figure 2a, performance suffers significantly due to the errant network transits. Because HDFS assumes that it is working with individual HDDs, it believes there is no other way to get data to that HDD besides going over the network to the client that supposedly contains it. However, when all of these local HDDs are replaced with paths to remote storage, the storage becomes equally close to all of the clients, and therefore an additional transit through some other node is complete overhead. It would be better for the DN in client Node A to write both replicas to separate NAS systems itself, but there is no way to effect this in HDFS without significant changes to its codebase that handles topology. This inefficient behavior is experimentally validated in Figure 3a, which shows that while writes should solely result in network send-bandwidth from the clients to the NAS, high receive-bandwidth is also occurring. In short, when we use HDFS with a replication level of 2 (original plus one copy) atop NAS, we should expect one out of three of the total network transits of the file to be overhead.

Moving the DN onto the NAS head nodes does not remove this errant pass-through behavior for writes, as shown in Figure 2b, but it does have the potential to alleviate it, depending on the relative link sizes available to nodes and connected to the NAS systems. Specifically, while the client machines in our experiments solely have a single Gigabit Ethernet link, each NAS system has two bonded 10 Gigabit links available. Nevertheless, it is quite computationally expensive to manage many tens, if not hundreds, of high-bandwidth, busy streams. This is a key issue we run into; even with only 10 clients per NAS system stream management overhead outweighs any benefits we gain from a higher theoretical peak performance.

In the third architecture, without HDFS shown in Figure 2c, there is no potential for any misunderstanding on how the storage is actually laid out – MapReduce is operating directly on normal files served via NAS. This enables all accesses to incur the fewest transits to maximize performance.

B. *Read Transport*

Reads seem considerably simpler than writes since no replication is performed – one imagines that reads should go directly through the local NAS mount and therefore avoid any pass-through situations. While this is true for our DN-on-NAS node and No-DN architectures, this intuition was found to fall flat for the simplest architecture of DN-on-Client.

We experimentally document the problem witnessed with our NAS on Client architecture in Figure 3b, which shows that while reads from NAS storage via HDFS should only show large amounts of received data, our per-node aggregation shows significant amounts of sends as well. This phenomenon is the result of a task being placed on a client node that the HDFS NameNode does not believe to have the file that task operates on stored locally. Misplacement results in a request to one of the other nodes that the NameNode does believe to have the file locally, which starts a pull-through from the NAS system to the requestee and finally to the requester. These sends reach as high as around 33% of the bandwidth of the receives as shown in the middle of Figure 3b, indicating about a third of the tasks were misplaced.

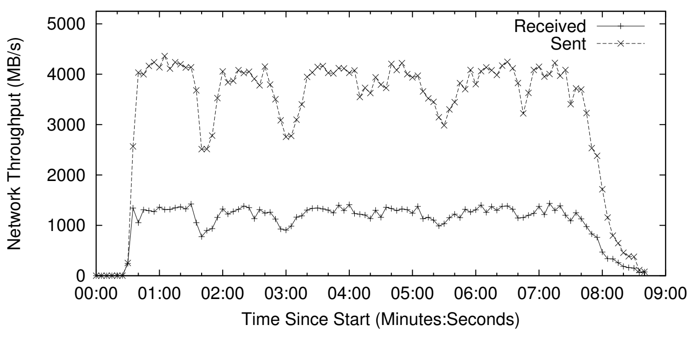
VI. RainFS

As we have shown, there are considerable overheads when utilizing HDFS to access NAS storage, yet bypassing HDFS entirely removes many of the reliability boons we had previously enjoyed. Further, bypassing HDFS requires that the NAS systems are capable of federation, which is not always the case, particularly with systems from different vendors. Therefore, we decided to implement a solution that attempts to avoid these overheads while concurrently retaining the ability to replicate over discrete failure domains and provide client-level federation. To that end, we present the Reliable Array of Independent NAS File System (RainFS), an intermediate file-system to replace HDFS for MapReduce on NAS applications.

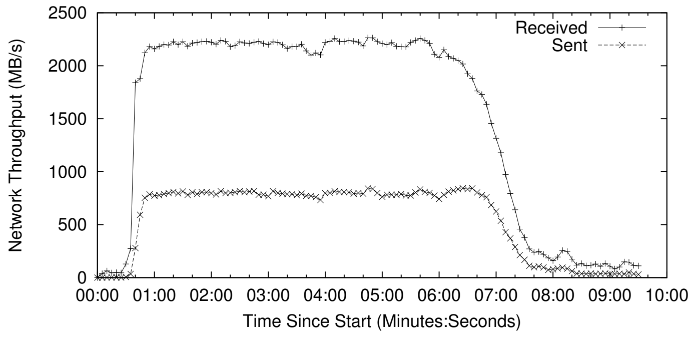
A. *Design Desirata*

The goals in the design of RainFS were four-fold:

1) *Client-Level Federation of NAS Systems:* Enable MapReduce to take advantage of the performance of all of the available NAS systems concurrently *and* maintain discrete failure domains.



(a) Errant Pass-Through Write Transport Behavior



(b) Errant Pass-Through Read Transport Behavior

Fig. 3: Write-intensive and read-intensive benchmarks on 50-node cluster and 5 NAS shelves, demonstrating poor write and read transport behaviors for DN-on-Client architecture.

2) *Full Replication:* Restore the ability to replicate files written in MapReduce.
3) *No Data Pass/Pull-Throughs:* Neither writes nor reads should ever go through another client node on its way to or from the NAS systems.
4) *A Fair Namespace:* Create a framework-agnostic namespace where no imports or exports are required.

B. Implementation Overview

RainFS employs two key mechanisms to achieve the aforementioned goals. The first is utilizing symbolic links (symlinks) and hidden folders in order to allow a single NAS system to manage the remaining NAS systems and present a fair, federated namespace to MapReduce and POSIX applications. The second mechanism is providing metadata management via hidden metadata files beside the symlinks.

RainFS allows the NAS systems to be unfederated at the storage-level and yet, exposes a unified namespace, by maintaining that namespace on one of the many NAS systems available to the administrator. In that managing-NAS namespace, instead of storing the data directly when written from MapReduce, it stores symlinks that lead to the data files in one of a number of distributed, hidden directories. It is important to emphasize that the symlinks nor hidden metadata files are not on individual, local clients, but on a single managing-NAS system such that all clients see them in a unified manner. This enables concurrent reads on a set of files in the visible RainFS file system to be load-balanced across many or all of the NAS systems, depending on how many files are accessed at once. Further, this allows an MPI or other POSIX application to read a MapReduce-generated file directly from the symlink in the visible namespace of the managing NAS or to write its own set of files and allow MapReduce to turn around and process those files directly without import or export.

The second mechanism, metadata management via hidden files beside the symlink, allows RainFS to manage these many hidden files and folders that the symlink points to. As we will discuss in the following sections, there are numerous consistency issues that must be addressed on file create, delete, and move, and RainFS achieves all of this without its own centralized metadata manager.

C. File Operations

To enable easy adoption and code-reuse, RainFS extends from the abstract base class FileSystem as provided in the Hadoop code. This makes it a sibling to other FileSystem-extended classes like FTPFileSystem, where FTP servers are used for storage, S3FileSystem, where Amazon S3 is used as the backing store, and even HDFS, which also extends from the FileSystem class. Furthmore, for operations that do not require special handling of symlinks or hidden file metadata (e.g., a mkdir), RainFS re-uses code from the RawLocalFileSystem class to provide a similar range of operations available in other Hadoop filesystems. The operations which do starkly differ in RainFS from those available in RawLocalFileSystem are file creation, deletion, and move, and we detail them at length in this section.

Algorithm 1 File Creation

```
1: procedure CREATE($filepath, replication$)
2:     LOCK(FILEPATH)
3:     $rndID \leftarrow$ RNDGEN.NEXTLONG
4:     $nasStart \leftarrow rndID\ mod\ nasCount$
5:     $bucket \leftarrow rndID\ mod\ 2^{bucketPower}$
6:     for $i \leftarrow 0, replication - 1$ do
7:         $files[i] \leftarrow$ BUILDPATH($(nasStart + i)\ mod\ nasCount, bucket, rndID$)
8:         FILE.CREATE($files[i]$)
9:     end for
10:    CREATEMETASTORE($filepath$)
11:    SYMLINK($filepath, files[0]$)
12:    UNLOCK(FILEPATH)
13: end procedure
```

1) Create: Algorithm 1 creates a file by taking as parameters the path to the file and the replication level desired. First, we utilize NAS file locking to create and lock a hidden locking file beside the path of the symlink we are seeking to create. Since MapReduce is designed for Big Data (and therefore fewer, larger files), we do not believe a basic locking approach should have notable performance impacts on any realistic workloads. We then generate a psuedo-random long integer and identify both the first NAS system the symlink should point to as well as the *bucket* in use. Since the first NAS is randomly chosen and then subsequent replicas are just round-robined around available other NAS systems, the replicas will be uniformly distributed. We utilize the concept of *buckets*, which are truly just folders, and a configurable $bucketPower$, to ensure that it is unlikely that any single folder becomes overwhelmed with a huge number of files. Without buckets, depending on the NAS system in use, folders with a huge numbers of files could very well suffer performance degradation.

Then we create each replica, iterating through the available NAS systems as specified in the RainFS configuration file in round-robin fashion. Changing available NAS systems is as simple as altering the configuration file, as it is read upon every call to the RainFS library; no restart required. However, in similar nature to HDFS files needing to be read and rewritten if the block size is desired to be changed, RainFS does not yet provide capacity rebalancing if the replication level changes. One will need to read, rewrite, and delete the old file to achieve rebalancing. Following creation of the replicas the symlink to the first NAS is created and the metadata information about the file is stored in a hidden file beside it.

Once the file is unlocked this procedure returns a new output stream for subsequent writes. Writes occur simultaneously to all replicas via threading, one thread per replica. This process is identical for subsequent writes sometime later after the file creation. On read access, only the data file that the symlink points to will be actually read from – reads are not performed concurrently from all replicas available. However, MapReduce programs frequently work with multiple files concurrently, so it should still achieve load balancing on read-intensive workloads across all NAS systems. In future work we are considering improving the read function so even single, huge files, are served from all available replica locations. This achieves client-level federation of the NAS systems, allows for replication all the way up to the number of NAS systems available, avoids transit overheads for both writes and reads since each client directly contacts the NAS systems for each of its I/Os, and maintains a namespace on the primary NAS system that non-Hadoop applications can access.

Algorithm 2 File Deletion

```
 1: procedure DELETE($filepath$)
 2:     LOCK(FILEPATH)
 3:     $replicas \leftarrow$ GETREPLICAS($filepath$)
 4:     FILE.DELETE($filepath$)
 5:     for $i \leftarrow 0, replication - 1$ do
 6:         FILE.DELETE($replicas[i]$)
 7:     end for
 8:     FILE.DELETE($filepath.metadata$)
 9:     UNLOCK(FILEPATH)
10: end procedure
```

2) Delete: File deletion as shown in Algorithm 2 works in nearly reverse order as create, and is done so with good reason: if intermediate failure occurs partway through a creation or deletion, these routines are built to maintain a reasonably sane environment even in the face of failures. Further, partial creates or deletes should be easy to clean-up with this ordering by an independent, scanning RainFS checking daemon. In the delete routine the file destined for deletion is locked via NAS file locking as done with create and all of the replicas are determined from the metadata file. Then, the symlink is removed such that any subsequent operation should recognize that the left-over replicas should also be deleted. Finally, the replicas are removed followed by the metadata file. The metadata file is left until last in order to preserve information and expedite clean-up in the event of an interrupted delete.

Algorithm 3 File Move

```
 1: procedure MOVE($filepath, newpath$)
 2:     if $filepath > newpath$ then
 3:         LOCK(FILEPATH)
 4:         LOCK(NEWPATH)
 5:     else
 6:         LOCK(NEWPATH)
 7:         LOCK(FILEPATH)
 8:     end if
 9:     UPDATEMETASTORE($newpath, filepath$)
10:     FILE.MOVE($filepath, newpath$)
11:     FILE.DELETE($filepath.metadata$)
12:     UNLOCK(FILEPATH)
13:     UNLOCK(NEWPATH)
14: end procedure
```

3) Move: Last of the common file operations, file move, would be a tricky routine if it were not for locks provided via NAS file locking. As we mentioned, providing locks around the entire routine is a reasonable approach for creates and deletes – as a Big Data framework, MapReduce was never designed for rapid creation or deletion of huge numbers of tiny files. We find moves to also be satisfactory for such routine-encompassing locks because *we do not actually move the data, we only move the symlink and the metadata file*. This allows for a very short time to be spent in the locked section compared to a full data copy and subsequent deletion of the old data. Therefore, in the move routine, we lock both the target and the source files in order to prevent any concurrency issues from arising. Once both are successfully locked, we first update the metadata file at the target and then and only then move the symlink to that target. Updating the metadata file at the target first prevents the possibility of a partial move resulting in lack of a metadata file alongside the new target symlink. Once the move is successful (metadata and symlink), we then delete the old metadata file at the source location and unlock both the source and target.

D. Failure Handling

The last consideration before experimentally comparing the performance of RainFS against the previous architectures is reliability of RainFS in the face of failure of a disk or rack.

First, just as in the other architectures, since it utilizes NAS storage, it gets single- or double-disk failure tolerance based on the RAID level the NAS system employs *for every NAS system*. However, unlike the architecture that simply bypasses HDFS, RainFS achieves federation at the client-level and therefore the fault domains among NAS systems are not conjoined. Concurrent failures in distinct NAS systems will therefore not aggregate – for five NAS systems and RAID-5, five disk failures can be tolerated without data loss in any of the failure domains if they occur in separate systems.

Furthermore, if an entire rack becomes unavailable and RainFS is unable to contact it for a read, it will simply iterate through the remaining replicas to attempt transfer from a NAS system on another rack. A notable caveat here is that the master NAS is a single point of failure; if it fails the unified namespace will be unavailable until it is restored.

Component	Description	Per-VM
Processor	Intel Xeon E5240	2 Cores
Memory	DDR2 667 MHz ECC	3.8 GB
Hard Disk	Sata II 7,200 RPM	200 GB
NIC	Unknown Card	1 Gb/s

TABLE II: Hardware and VM resources

Beyond what RainFS can promise in the face of failure or unavailability of one of the NAS systems, we must also briefly consider how consistency is maintained. We make one important assumption in this work to simplify RainFS and keep it out of the critical path for non-MapReduce applications: if a file is created via Hadoop, the user should take care to also delete it or move it with Hadoop. Users can of course (as it was one of our goals) read from those files using any application they wish, be it MapReduce-based or not. The reason behind this assumption is that since RainFS is solely in Hadoop, if an external user or application moves a symlink in the unified namespace, RainFS will not be able to keep up and move the metadata file along with it. Similarly, external deletions of symlinks will leak storage since the metadata files and replicas still remain in hidden folders on the distributed NAS systems. This RainFS checker similarly attempts to complete or roll-back partial creates, deletes, and moves after a specified time-out. Improving the RainFS checker to be more robust beyond these capabilities is a target in future work.

VII. Evaluation

We now describe our experimental environment, the benchmarks we used to tease out differences between architectures, and provide results for and discussion on our experiments.

A. Experimental Setup

To experimentally validate our expected overheads and proposed optimizations we used a medium-sized cluster of 50 nodes, which utilize five shelves of Panasas ActiveStor 12. The nodes have Gigabit NICs, which are connected to one of four Force10 S50n Gigabit switches. These switches are in turn connected by dual 10-Gigabit Ethernet uplinks to a single Force10 S4810p 10-Gigabit switch, which our NAS system is also connected to via two, 10 Gigabit Ethernet links bonded together per shelf. Each node is running KVM with a virtual machine image of CentOS 5.5, which is the environment for all of our experiments. Further specifics regarding the hardware is listed in Table II. We required virtualization in order to take advantage of the Panasas DirectFlow client module, which at the start of this work was solely available for RedHat-based distributions and our hosts run Debian.

B. Benchmarks

Perhaps the most ubiquitous macro-benchmark in the Hadoop space is the TeraSort benchmark, which is a suite of MapReduce applications designed by Yahoo! in 2008 [24] designed to compete in (and enabled them to win) the terabyte sort competition [25] that year. The three major components of the benchmark are:

TeraGen: The first component of the suite is TeraGen, an application that utilizes MapReduce to automatically divide the work of generating a configurable number of rows of key/value pairs over available clients. This benchmark is almost exclusively write-intensive and therefore, is used in this work as our write-throughput microbenchmark.

TeraSort: The second component is the most complex and performs the sort itself. TeraSort incorporates a custom partitioning algorithm that uses a sorted list of $N - 1$ keys. This enables a simpler, nearly embarrassingly parallel sort that correspondingly scales well. It has a read- and CPU-intensive Map-phase, a network- and memory-limited shuffle phase, which shares the now-partially-sorted data, and last performs a write-intensive reduce phase, where the data is merged and outputted to disk fully sorted. Because it exhibits a full spectrum of MapReduce application behavior, we consider this component of the suite representative of more compute-heavy applications.

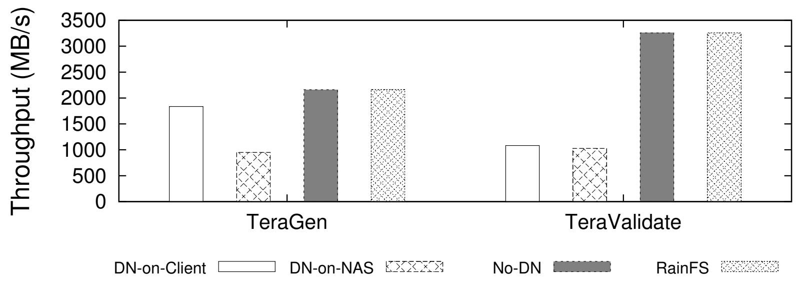
TeraValidate: The last component of the TeraSort benchmark suite is the validation application, which simply reads through the entire set of data and makes sure each key is sorted properly (it is less than or equal to the one previous). This benchmark, barring any errors (we encountered none in any of our tests), is completely read-intensive and therefore is used as a read-throughput microbenchmark in this work.

This benchmark exhibits the behaviors we expect most HPC post-processing analytics will exhibit. For instance, it is easy to imagine a scientist would first simulate a nuclear reaction using traditional HPC compute and storage. Then, they might seek to leverage a big data framework such as MapReduce as explored in this work to do a light filtering of the data, which might generate an only slightly smaller dataset (like the write-heavy TeraGen). Then they may sort to find areas with the highest or lowest temperatures (like TeraSort), and last look through all of the sorted data to verify no anomalies exist (like TeraValidate).

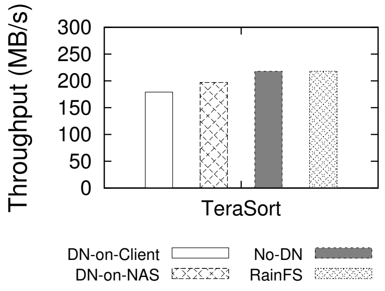
C. Results

Having laid out our experimental framework and the Hadoop TeraSort benchmark suite we utilize for this work, we begin our discussion of results by first presenting a basic overview of the parameters we configured for the TeraSort runs. In these experiments we first write 0.5TB of data using TeraGen and then sort that data using TeraSort, which generates a separate 0.5TB of data. Finally, we read in and validate that the second 0.5TB dataset was truly sorted using TeraValidate. Additionally, for all of our runs, we execute each benchmark three times in order to find a reasonable average. With half a terabyte, each machine is writing at least twenty gigabytes of data over the course of the benchmark suite and reading the same amount, making this a considerably out-of-memory dataset. Nevertheless, we take precautions against cache-effects by flushing the cache on all of our client nodes.

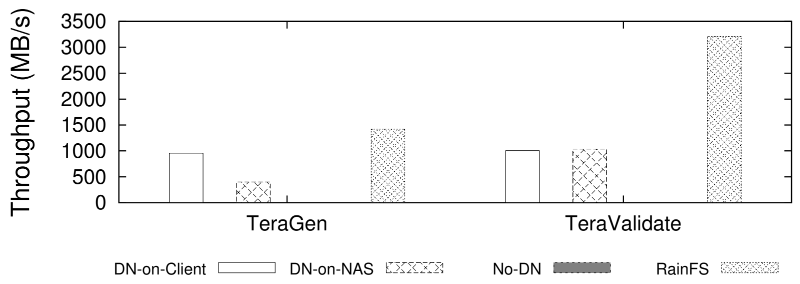
As a last consideration, we utilized the local HDD in each machine as the temporary storage space for MapReduce during its shuffle phases. This decision was made after thorough testing of entirely NAS-based setups, where no local disks were in use for the benchmark. In those we found performance to be much better when keeping temporary data local to the machine itself rather than pushing it out over the network to remote storage and subsequently pulling it back when needed (often very soon after the push). We believe utilizing a single



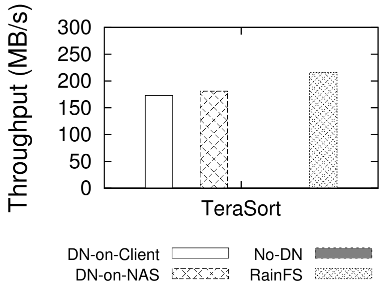
(a) Replication Level 1: Write- and Read-Intensive



(b) Replication Level 1: Mixed I/O



(c) Replication Level 2: Write- and Read-Intensive



(d) Replication Level 2: Mixed I/O

Fig. 4: User-perceived throughput of read-, write-, and compute-intensive benchmarks on a medium size cluster using all four architectures. Shown for replication level of 1 and 2 (where 1 means only the original data file exists, and 2 means two copies exist within the NAS systems). The No-DN architecture, or where neither HDFS nor RainFS is used at all, cannot replicate and therefore is missing from the bottom graphs.

local disk for temporary shuffle data does not jeopardize the intent of our exploration for two reasons: First, no permanent data stays on that disk and thus the reliability guarantees that HDFS and/or NAS promise are not in danger. Second, while not all HPC systems have a local disk (some boot simply over the network), many do, including current #1 Top500 Tianhe-2 [9], and almost all others provide some form of solution for a fast scratch space even if it is remote.

Moving to the results, let us first consider the write- and read-intensive benchmarks for replication levels of 1 and 2 as shown in Figures 4a and 4c. In the former, three main take-aways surface: First, DN-on-NAS runs into significant performance degradation for writes – it only begins to compete with any of the other architectures on reads, and that is because the DN-on-Client architecture begins to suffer from poor task placement (resulting in data pull-through). Second, and related, the impact of this errant data pull-through phenomenon resurfaces for TeraValidate in the DN-on-Client architecture. While performance improves for the No-DN and RainFS architectures when going from writes to reads, performance plummets for reads on the DN-on-Client setup. Third, the No-DN and RainFS architectures perform almost identically in all tests. This is a result of both of these architectures being based off of the same code-base, which performs I/O almost directly through standard Java I/O libraries.

In the latter set of read- and write-intensive benchmarks performing duplication, we note two points of interest: First, the DN-on-NAS node architecture performs even worse than in the previous case, achieving less than 10% of the theoretical throughput available to the NAS systems. This seems to make a fairly cogent case against pigeon-holing a distributed data framework like Hadoop MapReduce through a limited number of master nodes; it simply will not scale or achieve higher replication levels well once those nodes are overwhelmed. Second, since the No-DN architecture, or the architecture that skips both HDFS and RainFS and goes directly to NAS, cannot perform any replication, it is missing from this graph intentionally to accent the reliability/performance trade-offs.

Now examining the compute-intensive benchmark TeraSort as shown in Figures 4b and 4d, the findings are somewhat less striking but nevertheless fit intuition. First, because this benchmark is running on simply dual-core machines with limited main-memory, we should expect any improvements in storage access speed to only improve a limited fraction of the run time. This is demonstrated with much smaller swings from the worst to the best in the architectures. Further, DN-on-NAS node finally takes a win in this case because the DN responsibilities have been moved off of the compute nodes, allowing them to move faster, and I/O is not the bottleneck. Last, No-DN performs almost identically with RainFS for replication level of 1, and is excluded from replication level of 2 for the aforementioned reasons.

Lastly, we analyze this performance data in Figure 5 to determine how the various architectures fair when increasing replication level. As the simplest architecture matches up with, our intuition suggests if twice the amount of data is being written, then the slow-down should be two times. However, DN-on-NAS fairs worse than this number, coming in at 2.38 times slower and RainFS fairs better, showing only a 1.52X slowdown. In the former case, we believe this to be related to the already overburdened NAS node slowing down further as it continues to struggle with more high-throughput network

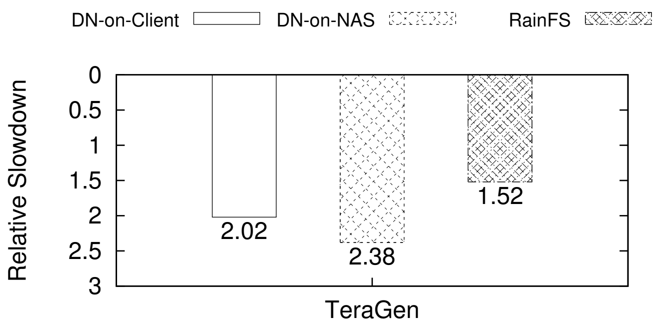


Fig. 5: Throughput impact on write-intensive workloads when going from replication level of 1 to 2.

streams. However, for RainFS we were initially quite unclear as to how it performed better than in the replication level 1 case. What we have found through microbenchmarks and careful observation of the test as it runs is that since RainFS makes use of threads to write both of the copies simultaneously when replicating, it is able to hide some of the overhead in the I/O path and the other stream of data fills in that space while the initial one idles.

VIII. Related Works

Despite the huge volume of work done in the cloud computation and parallel file system arenas, there are only two academic works to our knowledge that seek to find an efficient coexistence between them, and both are just short papers reporting research that is still in progress.

In the first work, MixApart [23], the authors create a new task scheduler and caching manager that relies on local HDDs to perform staging of data brought from shared storage. Their work does no exploratory research into whether standard incarnations of Hadoop are possible atop shared storage nor does it appear to operate without powerful compute-local storage, and therefore one of the major incentives for our work, infrastructure conslidation, becomes impossible. Last, there is no consideration of storage reliability or improving federation capabilities in their work; they rely on whatever the underlying shared storage can provide.

In the second work [10], the authors perform a comparative study of unmodified Hadoop MapReduce on HDFS versus a modified version of GPFS. Unlike our work, where we seek from the outset to use MapReduce on *function-specific dedicated storage*, this work attempts to retain the merged infrastructure of Hadoop where compute and storage share the same machines.

In the commercial space, EMC Isilon and NetApp have released products that perform, to one degree or another, actions similar to two of our explored architectures. In the former case, EMC Isilon provides their solution, OneFS [3], which exports a wire-protocol version of HDFS but that translates HDFS commands extensively into Isilon-specific data movement in the back-end. Our wire-protocol architecture (DN-on-NAS node) has parallels with this setup, although the exact implementation is not equivalent. In the latter case, NetApp provides their OpenSolution for Hadoop [7], which enables individual compute nodes in the Hadoop cluster with SAS-attached storage instead of their own commodity HDDs. This methodology has some parallels to our DN-on-Client setup, except we utilize NAS storage rather than directly-attached storage. These commercial implementations of architectures similar to ours were part of our motivation to explore this arena and try to bring some clarity about the benefits and pitfalls of the various approaches to integrating Hadoop MapReduce and shared storage solutions.

IX. Conclusion

As an increasing number of organizations and researchers in HPC begin to take stock of their I/O intensive workloads and consider a Hadoop environment, particularly for ad-hoc analytics and post-processing, we believe our work has shed light on the potential to combine new compute frameworks with traditional storage infrastructure. In this paper we have detailed the numerous reliability implications, locality impacts, and caveats involved in utilizing three different architectures to effect MapReduce atop NAS with standard software. We have further designed and presented RainFS, our custom Hadoop File System that works to overcome the many pitfalls observed in the previous architectures. Finally, we have compared these architectures along the dimensions of reliability and performance on a real cluster, and demonstrated performance improvements for RainFS as high as 127% for write-intensive workloads and 217% for read-intensive workloads for replication level 1, and as high as 254% for write-intensive workloads and 210% for read-intensive workloads when performing duplication to achieve higher reliability guarantees.

References

[1] Amazon s3. http://aws.amazon.com/s3/.

[2] Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] Emc isilon onefs. http://www.emc.com/domains/isilon/index.htm.

[4] Hadoop. http://hadoop.apache.org/.

[5] Kosmosfs. http://code.google.com/p/kosmosfs/.

[6] The lustre file system. http://wiki.lustre.org/index.php/Main_Page.

[7] The netapp opensolution for hadoop. http://www.netapp.com/us/solutions/big-data/hadoop.aspx.

[8] Poweredby hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[9] Tianhe-2 (milkyway-2) - th-ivb-fep cluster. http://www.top500.org/system/177999.

[10] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: Do we really need to reinvent the storage stack? In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[11] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.

[12] Joe Buck, Noah Watkins, Greg Levin, Adam Crume, Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn, Neoklis Polyzotis, and Aaron Torres. Sidr: Structure-aware intelligent data routing in hadoop. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 73:1–73:12, New York, NY, USA, 2013. ACM.

[13] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: Array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 66:1–66:11, New York, NY, USA, 2011. ACM.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[17] S. Shepler et. al. Network file system (nfs) version 4 protocol, 2003.

[18] William Gropp et. al. *MPI: A Message-Passing Interface Standard*. 2009.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[20] Christopher Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall Professional Technical Reference, 2003.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[22] . Iii Ligon, W. B. and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 471–, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Madalin Mihailescu, Gokul Soundararajan, and Cristiana Amza. Mixapart: Decoupled analytics for shared storage systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 133–146, Berkeley, CA, USA, 2013. USENIX Association.

[24] O. O'Malley. Terabyte sort on apache hadoop. Yahoo, 2008.

[25] O. O'Malley and A.C. Murthy. Winning a 60 second dash with a yellow elephant. Proceedings of sort benchmark, 2009.

[26] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 16–16, Berkeley, CA, USA, 2002. USENIX Association.

[27] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J. Lang, Garth Gibson, and Robert B. Ross. On the duality of data-intensive file system design: Reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 67:1–67:12, New York, NY, USA, 2011. ACM.

[28] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.