Performing Cloud Computation on a Parallel File System

Ellis H. Wilson III
Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
ellis@cse.psu.edu

1. 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 [11], Google File System [15], and BigTable [10] tools and their open source siblings, Hadoop [3], its distributed file system (HDFS) [8], and HBase [9], 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 [15]. With this heritage, interoperation with systems, storage, and tools from other environments such as high performance computing (HPC) can not be taken for granted. 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 to lower costs of storage systems.

With the emergence of cloud resource allocators like Mesos [17] and Yarn [1], 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 [14]. 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 [18], Lustre [5], GPFS [22], and PanFS [24] 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, while parallel file systems 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.

1.1 Motivation: Why NAS?

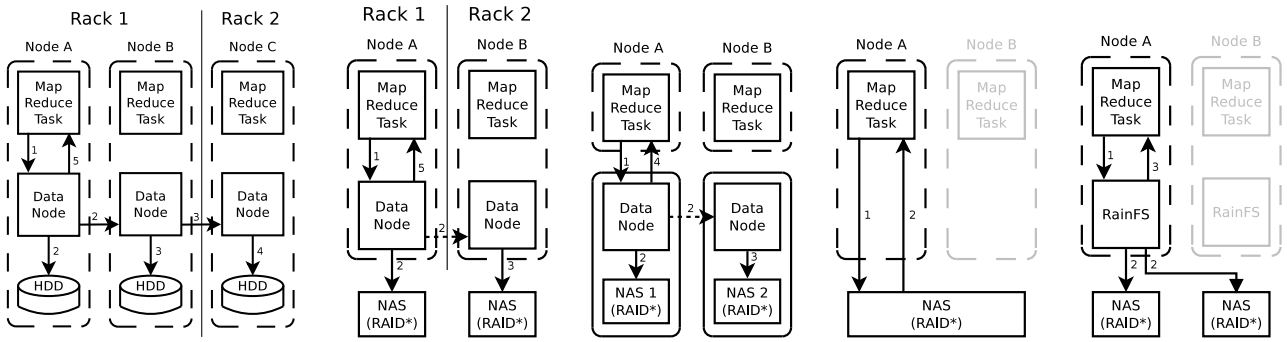
Nevertheless, due to the attractiveness of the solutions in the Big Data space, many organizations have acquired or put aside a separate set of nodes for exclusively Hadoop compute and storage and have suffered through the cost of capacity waste and expensive copies. However, with the scattering of storage throughout the compute nodes come a number of drawbacks, which give particular pause to moving away from NAS systems for HDFS. The following issues provide motivation for my effort in this work to seek consolidated compute for HPC and cloud computing, all atop a consolidated NAS system.

- *Loss of Infrastructure Consolidation:* In HDFS one loses the ability to run various application types; everything must be a MapReduce application.
- *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 some workloads I/O performance degrades since Hadoop is so tailored for performance on large datasets.
- *Loss of High-Availability:* The Hadoop NameNode is a single point of failure. On failure administrative intervention is required.
- *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:* If 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.

2. BACKGROUND AND RELATED WORK

2.1 HDFS Overview

When Google introduced its MapReduce framework [11] in 2004, a restructuring of large scale data analysis was spurred with the most notable open-source implementation of Google's framework being Hadoop [3]. MapReduce allows users to write parallel programs that are automatically broken into Map and Reduce tasks and executed in parallel within a distributed environment. HDFS, taking its initial specification from the Google File System [15], 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.



(a) Traditional Hadoop (b) DN-on-Client (c) DN-on-NAS (d) No-DN (e) RainFS

Figure 1: How "writes" proceed in the different architectures. Dashed arrows signify network overhead communication, and grayed out nodes indicate they have no role in writing the file from Node A.

Hadoop MapReduce and HDFS have been shown to scale to thousands of nodes, millions of files and petabytes of storage [7]. HDFS provides double-disk failure tolerance via replication, out-of-band metadata access, is designed in Java for platform independence, supports Unix-style file permissions, and automatic capacity balancing. Hadoop has been adopted by many organizations, some of the most notable being Yahoo!, Facebook, Twitter, and LinkedIn.

2.1.1 Replication in HDFS

HDFS achieves double-disk failure tolerance by replicating a file across multiple nodes. The process, shown in Figure 1a, proceeds in the following manner: First, contact the NameNode to let it know you wish to write a file (for clarity, communications with the NameNode are excluded from this and subsequent diagrams). Upon a response indicating success and locations to write each copy to from the NameNode, the requesting Node A begins concurrently writing its first copy to the local disk and its second copy over the network to Node B. NameNode attempts to select Node B such that it is located within the same rack as Node A to take advantage of higher-performance intra-rack communication, and select Node C as a node outside of that rack to provide tolerance to the failure of an entire rack. As Node B begins to receive its replica, which it stores it to local disk, it concurrently begins pipelining that replica to the last destination, Node C. This process is called *replication pipelining*, and is intended to share the replicating work with another node (B) to decrease the total time between the start of the write and completion of three replicas being entirely on disk.

2.2 Related Work

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

In the first work, MixApart [19], 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, their scheme does not appear to operate without powerful local storage, and therefore one of the major incentives for my 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 [12], the authors perform a comparative study of unmodified Hadoop MapReduce on HDFS versus a modified version of GPFS. Unlike my work, where I 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 my explored architectures. In the former case, EMC Isilon provides their solution, OneFS [2], which exports a wire-protocol version of HDFS but that translates HDFS commands extensively into Isilon-specific data movement in the back-end. My wire-protocol architecture (DN-on-NAS node) has extensive high-level parallels with this setup, although the exact implementation is not equivalent. In the latter case, NetApp provides their OpenSolution for Hadoop [6], 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 my DN-on-Client setup, except I utilize NAS storage rather than direct-attached storage. These commercial implementations of architectures similar to mine were part of my 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.

3. MY APPROACH

3.1 Architectures Explored

In this work I seek to explore possible architectures that allow Hadoop MapReduce to run alongside traditional POSIX applications and against a consolidated storage system provided via NAS. My **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:

HDFS as a Client Service: As HDFS employs node-local VFS to store chunks, reconfiguration can replace node-local storage with remote NAS storage. As can be seen in Figure 1b, this architecture is the most similar to the traditional HDFS setup shown in Figure 1a 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.

	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 1: 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.

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 I will later show, while easy to setup and get started with, this method has serious reliability and performance short-comings.

HDFS as a Wire Protocol: 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 [13] or CIFS [16], with the server running within the NAS system itself. In this architecture I 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 1c. However, forcing all of the data both for reads and writes through these nodes creates serious performance bottlenecks.

No HDFS: While HDFS requires MapReduce applications to efficiently operate on its data, MapReduce will operate on non-HDFS data if configured correctly, so skipping HDFS and going directly to NAS is possible. One large caveat is illustrated in Figure 1d – 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 amongst 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 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.

In each of the above architectures I 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, each having their own failing points, my **second contribution** is the design and development of a new Hadoop FileSystem interface class.

My novel file system, the *Replicating Array of Independent NAS File System (RainFS)*, overcomes all of the major issues I discovered. As I had done with the three more standard approaches above, I analyze and evaluate RainFS along the dimensions of reliability and performance. In all tests performed, RainFS performed as well or better and provided reliability as good as or better than all other architectures.

3.2 Reliability in NAS vs. Hadoop

To continue to operate during the failure of one or more HDDs 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 chosen RAID level. To handle failure of network and power components of the storage system that, upon such failure, would result in inaccessibility to an entire rack, NAS systems provide redundant hardware. Specifically, two or more Network Interface Cards (NICs) and Power Supply Units (PSUs) are made available in the NAS system so that if one fails there is automatic fail-over to the redundant component.

The Hadoop Distributed File System provides tolerance to individual disk failure by replicating every file created on distinct HDDs. This assures, in a very simplistic yet effective manner, that even upon failure of one of the drives another copy will remain. To handle the more extreme 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.

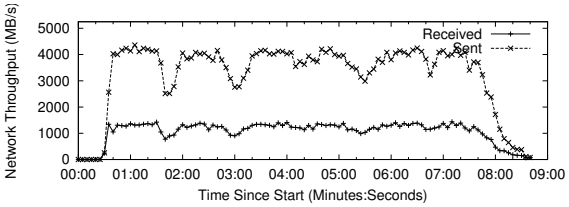
3.2.1 Combining for Hadoop on NAS

I now consider how each of the proposed architectures placed atop NAS handle failure. It should be noted that when I report "failure tolerance of X disks", I am referring to the *maximum X which can be tolerated, no matter which specific X disks fail.*

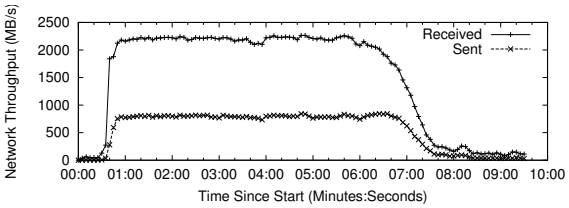
Therefore, let us first consider placing the DN on the client node and providing paths to NAS mounts. As mentioned, HDFS assumes each time it copies a file, it is in a totally separate disk (and therefore, failure domain), which will not be true if each client node sees paths to all NAS systems. Because a given DN, when given multiple paths, will randomly select one of the paths for writing the file, there is no promise that the DN that receives the responsibility for writing the second replica of a file will not pick the same path, and therefore the same NAS system as the first one.

However, I discovered that achieving a replication level of two is safely possible by giving all of the clients in a given rack access in HDFS 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. When running at a replication level of 2 (duplicates), HDFS immediately attempts to make the second replica outside of the rack, so this guarantees that the two copies are in two separate NAS systems. However, at triplication and beyond, Hadoop will attempt to create the second replica within the same rack, thereby running into reliability issues.

Moving to the DN on the NAS node architecture, 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 there is only have a single NAS system for each rack, there is a one-to-one mapping of DN to rack and therefore duplication to the same rack becomes impossible. This allows for high replication



(a) Errant Pass-Through Write Transport Behavior



(b) Errant Pass-Through Read Transport Behavior

Figure 2: Write-intensive and read-intensive benchmarks on 50-node cluster and 5 NAS shelves, demonstrating poor write and read transport behavior for DN-on-Client.

and reliability guarantees as shown in Table 1.

Last, examining the impact of guiding MapReduce to operate directly on the underlying NAS mount-point, I am 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, there is no way to perform replication, so I have no way to tolerate rack failure of unavailability.

3.3 Data Locality and Transport

3.3.1 Write Transport

Figure 1 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 1b, 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, despite the remote storage being "equally close" to all of the clients. 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 2a, which shows that while writes should solely result in network send-bandwidth from the clients to the NAS, high receive-bandwidth is also occurring.

Moving the DN onto the NAS nodes does not remove this errant "pass-through" behavior for writes, as shown in Figure 1c, 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 my experiments solely have a single Gigabit Ethernet link, each NAS system has two bonded 10 Gigabit links available, however, my setup may not be typical. Nevertheless, it is quite computationally expensive to manage many tens, if not hundreds, of high-bandwidth, busy streams. This is a key issue I run into; even with only 10 clients per NAS system stream management overhead outweighs any benefits I gain from a higher theoretical peak performance.

In the third architecture, without HDFS shown in Figure 1d, there is no potential for overhead or a misunderstanding on how the storage is actually laid out – MapReduce is operating directly on plain POSIX files served via NAS. This enables maximum performance for all accesses.

3.3.2 Read Transport

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

I experimentally document the problem witnessed with my NAS on Client architecture in Figure 2b, which shows that while reads from NAS storage via HDFS should only show large amounts of received data, my 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 type of data transit from the NAS system to the requestee and finally to the requester.

3.4 Design of RainFS

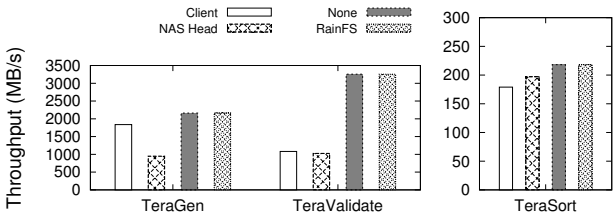
As discussed, there are considerable overheads when utilizing HDFS to access NAS storage, yet bypassing HDFS entirely removes many of the reliability boons previously enjoyed. Further, bypassing HDFS requires that the NAS systems are capable of federation, which is not always the case. Therefore, I decided 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, I have implemented the Reliable Array of Independent NAS File System, or RainFS, an intermediate file-system to replace HDFS for those looking to use MapReduce atop NAS. There is not sufficient space to describe the many facets of RainFS, but the four-fold goals were: • *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. • *Full Replication:* Restore the ability to replicate files written in MapReduce to as many NAS systems as are available. • *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. • *A Fair, Federated Namespace:* Create a framework-agnostic (i.e. accessible via MapReduce, MPI, RBDMS, etc.) namespace where no imports or exports are required.

4. RESULTS

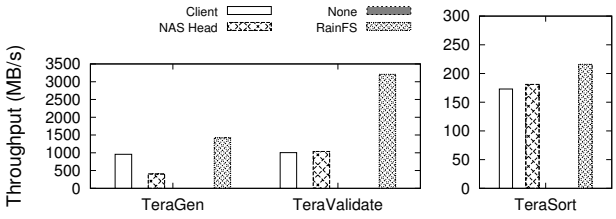
I now present my experimental environment, benchmarks employed to tease out differences between the architectures, and results from those tests.

4.1 Experimental Setup and Benchmarks

To experimentally validate the expected overheads and proposed optimizations I used a cluster of 50 nodes, which



(a) Replication Level 1: TeraSort Suite



(b) Replication Level 2: TeraSort Suite

Figure 3: 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 "None" architecture, or where neither HDFS nor RainFS is used at all, cannot replicate and therefore is missing from the bottom graphs.

utilize five shelves of Panasas ActiveStor 12.

Further, I have tested using the TeraSort benchmark, which is a suite of MapReduce applications designed by Yahoo! in 2008 [20] designed to compete in (and enabled them to win) the terabyte sort competition [21] that year. The suite is composed of:

1. **TeraGen**: First phase, an almost exclusively write-intensive application that utilizes MapReduce to automatically divide the work of generating a configurable number of rows of key/value pairs.
2. **TeraSort**: Second phase, that sorts TeraGen's output. 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.
3. **TeraValidate**: Last phase of the TeraSort benchmark suite is the almost exclusively read-intensive validation application, which simply reads through the entire set of data and makes sure each key is sorted properly.

4.2 Results

In these experiments I first write 0.5 Terabytes of data using TeraGen and then sort that data using TeraSort, which generates a separate 0.5TB of data. Finally, I read in and validate that the second 0.5 Terabyte dataset was truly sorted using TeraValidate. Each benchmark is executed three times in order to find a reasonable average. Cache-effects are avoided by flushing the Linux buffer-caches between runs.

Let us first consider the write- and read-intensive benchmarks for replication levels of 1 and 2 as shown in Figures 3a and 3b. In the former, there are three main take-aways: First, placing the DN on the NAS node runs into significant performance degradation for writes. It only begins to compete with any of the other architectures on reads, and that

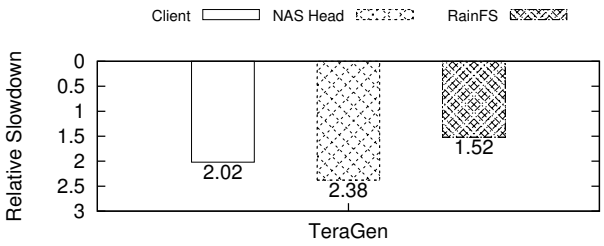


Figure 4: Throughput impact on write-intensive workloads when going from replication level of 1 to 2.

is because the DN-on-Client architecture begins to suffer from poor task placement (resulting in data "pull-through"). Second, and related, the similar "pull-through" phenomenon surfaces for TeraValidate in the DN-on-Client architecture. While performance improves when skipping HDFS and using the RainFS architecture when going from writes to reads, performance plummets for reads on the DN-on-Client setup. Third, the skipping HDFS and RainFS architectures perform almost identically in all tests. This is due to both architectures being based off of the same code-base, which performs I/O through standard Java I/O libraries.

In the latter set of read- and write-intensive benchmarks performing duplication, there are two points to note: First, the DN-on-NAS node architecture performs even worse than before, achieving less than 10% of the theoretical throughput available to the NAS systems available. This makes 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, the "None" architecture, or the architecture that skips both HDFS and RainFS and goes directly to NAS, cannot perform any replication, and therefore cannot be considered.

Now examining the compute-intensive benchmark TeraSort as shown in Figures 3a and 3b, the findings are somewhat less striking but nevertheless fit intuition. First, because this benchmark is running on dual-core machines with limited main-memory, storage access speed is not the bottleneck and therefore will only improve a limited fraction of the run time. 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 compute faster. Last, skipping HDFS performs almost identically with RainFS for replication level of 1, and is excluded from replication level of 2 for the aforementioned reasons.

Lastly, I analyze this performance data in Figure 4 to determine how the various architectures fair when increasing replication level. As the simplest architecture matches up with, my 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, coming in at 2.38 times slower and somehow RainFS fairs better, showing only a 1.52 times slowdown, due to the former being overburdened with high-throughput streams and the latter taking advantage of multi-threading to maximize throughput.

In conclusion, this work has shed light on the potential to combine new compute frameworks with traditional storage infrastructure to great effect in increasingly divergent times, and have detailed the numerous reliability implications, locality impacts, and caveats involved in utilizing four novel architectures to effect MapReduce atop NAS.

5. REFERENCES

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

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

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

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

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

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

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

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

[9] Fay Chang and Jeffrey Dean et. al. Bigtable: a distributed storage system for structured data. OSDI '06, 2006.

[10] Fay Chang and Jeffrey Dean et. al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[12] R. Ananthanarayanan et. al. Cloud analytics: Do we really need to reinvent the storage stack.

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

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

[15] Sanjay Ghemawat and Howard Gobioff et. al. The google file system. SOSP '03, 2003.

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

[17] Benjamin Hindman and Andy Konwinski et. al. Mesos: a platform for fine-grained resource sharing in the data center. NSDI'11, 2011.

[18] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. HPDC '96.

[19] M. Mihailescu, G. Soundararajan, and C. Amza. Mixapart: decoupled analytics for shared storage systems.

[20] O. O'Malley. Terabyte sort on apache hadoop. Technical report, 2008.

[21] O. O'Malley and A.C. Murthy. Winning a 60 second dash with a yellow elephant. Technical report, 2009.

[22] Frank Schmuck and Roger Haskin. Gpfs: a shared-disk file system for large computing clusters. FAST'02, 2002.

[23] Wittawat Tantisiriroj and Seung Woo Son et. al. On the duality of data-intensive file system design: reconciling hdfs and pvfs. SC '11, 2011.

[24] Brent Welch and Marc Unangst et. al. Scalable performance of the panasas parallel file system. FAST'08, 2008.