Is Cloud Storage Ready? A Comprehensive Study of IP-based Storage Systems

Zhonghong Ou, Zhen-Huan Hwang
Antti Ylä-Jääski
Aalto University, Finland
Email: firstname.lastname@aalto.fi

Feng Chen
Louisiana State University, USA
Email: fchen@csc.lsu.edu

Ren Wang
Intel Labs, Portland, USA
Email: ren.wang@intel.com

***Abstract*—Traditionally, network storage systems have mainly been dominated by two IP-based storage technologies, i.e., Network Attached Storage (NAS) and Storage Area Network (SAN). In recent years, cloud based storage (e.g., Amazon S3) has gained growing popularity for its high flexibility and cross-platform compatibility. Many enterprises are considering to replace traditional storage systems with cloud-based systems. Evaluating such a transition demands a systematic study on understanding the performance behaviours of the emerging cloud storage. To fill in this gap, in this paper, we conduct a comprehensive study on the three storage systems with realistic network conditions. Specifically, we select one representative from each category for comparison, i.e., Network File System (NFS) from NAS, Internet Small Computer System Interface (iSCSI) from SAN, and OpenStack Swift from cloud storage. We build a testbed and develop a suite of micro-benchmarks to study the impact of network complexities. Through a set of experiments and detailed analysis, we make several key observations: (1) iSCSI excels under good network conditions, e.g., in local area networks (LANs) where network delay and packet loss are trivial; (2) NFS and Swift are more suitable for complex networks such as wireless networks and Internet environment; (3) Swift is a viable replacement for NFS in all scenarios we investigate; and (4) system configuration on the client side impacts storage performance significantly and deserves adequate attention. We hope our findings can not only shed light on storage service design and optimizations, but also encourage more research on emerging storage technologies.**

***Keywords*-Cloud Storage; Swift; iSCSI; NFS; Performance Evaluation.**

I. Introduction

Enterprise storage systems have traditionally been dominated by two major technologies, i.e., Network Attached Storage (NAS) and Storage Area Network (SAN). Both technologies have been widely deployed in enterprise environment in the past decade [1], and have proved their performance and reliability over time.

In recent years, cloud computing paradigm has gained significant popularity, and starts to replace the traditional computing model. As a critical component, the emerging cloud storage (e.g., Amazon S3) provides a highly promising solution to enable a transition from dedicated storage to a more platform-independent IP-based storage. Many enterprise IT departments are considering to replace traditional network storage services with private or public cloud-based storage services [2]. Nevertheless, without a thorough understanding of such a new storage model comparing with traditional ones, it is challenging to realize such a transition efficiently and perform further optimizations.

Firstly, in today's enterprise environment, end users usually rely on wireless networks to gain mobility and flexibility. Unfortunately, such a practice makes it particularly challenging to ensure the quality of storage services. What's worse, users on mobile often need to access storage services through the unpredictable **Internet** (e.g., work from home or work on travel). These issues together inevitably introduce significant network complexity, and thus pose strong interference with the user-perceivable storage performance. Secondly, although NAS, SAN, and cloud storage are all important in practical environment, there still lacks a systematic study on understanding the three drastically different storage systems in a comparative way. It is important to understand their performance behaviours and relative strengths and weaknesses. Such insights will not only assist identifying the most suitable storage solution for different scenarios, but also provide useful hints to improve the emerging cloud storage design further.

To fulfil the purposes mentioned above, we conduct an experimental study striving to understand the intrinsic characteristics of NAS, SAN, and cloud storage, and investigate their implications in different scenarios. As the first attempt to provide such a study, we primarily focus on the **performance** aspect in this paper.

We selectively choose Network File System (NFS) [3], Internet Small Computer System Interface (iSCSI) [4], and OpenStack Swift[1], as the representative of each technology. In order to provide a fair comparison, we run the experiments on the same hardware setup, integrate the same Ext4 file system for each system, and access them all through standard POSIX APIs. To provide a controlled wireless environment, we use the Wide Area Network emulator (WANem) to emulate various network scenarios. We also design a set of experiments to cover different aspects of performance analysis.

With all the efforts mentioned above, we strive to answer the following questions:

- Is cloud storage a viable replacement in scenarios that

[1]https://wiki.openstack.org/wiki/Swift

are traditionally dominated by NAS and SAN?
- Is cloud storage universally better than NAS and SAN? If not, in what scenarios is cloud storage better?
- How much does network conditions and application behaviours impact the performance of each technology?

Through systematic analysis, we make several important observations: (1) We find that under ideal network conditions, SAN performs the best, while NAS slightly outperforms cloud object storage; under network conditions similar to the Internet, the performance of SAN declines the most, NAS sits in between, while cloud object storage remains relatively stable (unaffected). (2) From performance perspective, we conclude that object-based cloud storage is a viable replacement for NAS in all network scenarios we investigate, including both LAN and WAN environments. (3) We discover that the capability of utilizing multiple TCP connections affects performance significantly in a positive way, especially under realistic network conditions that involve nontrivial network delay and packet loss. (4) We notice that access behaviours have a remarkable impact on storage performance. Different access forms (e.g., `Direct` vs. `Sync` I/Os) should be chosen carefully, which is a tradeoff between performance and consistency.

As the first comparative study of its kind, we hope this study can shed light on understanding the intrinsic characteristics and system implications of each solution. Furthermore, we hope our study can inspire a series of similar work focusing on other aspects, e.g., consistency issues, of the systems.

II. Background

In this section, we briefly introduce the three technologies in general, and then select one representative from each family for detailed description.

A. Network Attached Storage

NAS provides access to file systems deployed on remote storage server via a file based interface. The server handles physical organization of data and coordinates concurrent access. Clients mount a NAS volume and integrate the shared namespace into the local file system and access data stored on the remote server in the same way as local files.

Network File System (NFS) [3] is a representative NAS protocol, and has been widely used since 1980s. Built on top of the host file system, NFS exposes a portion of the server file system to the clients. The clients access the exported namespace through Remote Procedure Calls (RPCs) [5]. File operations from the clients are converted into RPCs to the server, where the RPCs are further converted into local file system operations accordingly.

To date, NFS has experienced several generations. We focus on the latest version, i.e., NFSv4, in this paper. Compared with previous versions, NFSv4 introduces several new features. Firstly, NFSv4 is stateful, and the client issues OPEN and CLOSE operations before and after accessing a file. Secondly, it introduces the COMPOUND operation, where several individual operations coalesce to form a complex request to reduce the number of RPC calls (and the round trips) required for file system operations. Thirdly, NFSv4 adopts TCP as the transport protocol, which offers improved communication reliability. Finally, it introduces the `delegation` concept to temporarily transfer certain responsibilities from the server to the client to allow more aggressive caching for performance purposes. For its cross-platform popularity in practice, in this paper, we use NFSv4 as a representative NAS solution.

B. Storage Area Network

SAN exposes physical storage devices to the clients and provides block level access to them. Unlike NAS, SAN does not provide file system abstraction, but rather raw storage devices. Thus, the client itself needs to build a file system on the exposed SAN device. The client OS manages the file system directly, like any other direct-attached device. Data organization on the device and synchronization among concurrent access are handled on the client side. For this reason, I/O operations on SAN storage are subject to effects of generic block I/O mechanisms such as caching, prefetching, and scheduling schemes that are applicable to any local file system.

Internet Small Computer System Interface (iSCSI) [4] is a widely adopted SAN protocol in enterprise environment. Its client-server architecture consists of two major components: the client is called the *initiator*, while the server counterpart is called the *target*. With iSCSI, the block-level SCSI commands from the initiator are encapsulated into TCP/IP packets and transmitted over the Internet to the target. The target then unpacks the received packets and extracts the SCSI commands for execution. The iSCSI protocol utilizes the concept of *session* to keep track of communication streams between the initiator and the target. All I/O operations occur in block level. For its high popularity in reality, we use iSCSI as a representative SAN protocol in this paper.

C. Object-based Cloud Storage

Cloud storage emerges as a new paradigm to provide users with storage service of great flexibility. Users access data in unit of *object*, which is similar to the file in a conventional file system. Objects can be logically organized into *containers*, which are akin to directories in file systems. Unlike file systems, object-based cloud storage usually provides a flat namespace, which means containers cannot be nested. Besides executing HTTP requests manually, many cloud storage services also provide tools to integrate a cloud storage repository into the local file system.

Among the cloud storage solutions that are openly accessible, OpenStack Swift has gained wide adoption because of its technical maturity. OpenStack Swift is a distributed object storage system, which aims to emulate the behaviours of Amazon S3. There are several components involved in the Swift architecture. Herein we only introduce the components that are relevant to our experiments. Swift consists of four server processes, i.e., proxy, account, container, and object. The proxy server process is responsible for handling incoming requests, looking up locations of the storage nodes, routing requests accordingly, and returning responses back to the clients. The account server process is responsible for maintaining metadata of individual accounts and the listing of containers within each account. The container process is responsible for maintaining container metadata and the listing of objects within the container. The object server is responsible for storing, retrieving, and deleting objects. Objects are stored as binary files in the host file system.

CloudFuse[2] is a Swift client that provides file system interface for cloud users. Using a File System in User Space (FUSE) implementation, CloudFuse bridges the gap between HTTP based Swift API and the Virtual File System (VFS) of Linux OS. CloudFuse communicates with Linux kernel through the assistance of a FUSE library, which passes file system operations and responses as messages via a virtual device. The FUSE library supports multithreaded FUSE implementation. By default, CloudFuse enables this feature and is capable of utilizing multiple TCP connections. After mounting Swift to the local file system, file operations are automatically converted to HTTP requests.

Because of the technical maturity, we use OpenStack Swift as the representative of object-based cloud storage. Furthermore, to provide a fair comparison with NFS and iSCSI, we utilize CloudFuse to integrate a file system on top of Swift to provide the same file operations.

III. Environment and Methodology

In this section, we describe the methodology and environment setup for our experiments. The experiments are conducted in a custom testbed illustrated in Figure 1.

A. *Experiment Methodology*

As stated previously, the three storage systems are drastically different in nature, we strive to provide a fair comparison with the following efforts: (1) we use the same set of hardware for the three systems to avoid potential difference arising from hardware (cf. Figure 1); (2) we deploy the same file system (Linux Ext4) on the storage systems; (3) we access the three systems through the same file operations via the standard POSIX API; for Swift, we integrate CloudFuse on the client side to enable such operations; (4) we design the experiments in such a manner that they can largely separate performance differences arising from protocol design and implementation details. For that, we develop a series of custom-made micro-benchmarks, which consist of access

[2]http://redbo.github.io/cloudfuse/

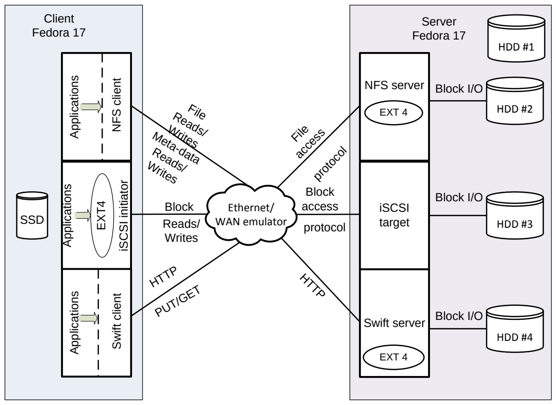


Figure 1: Testbed overview.

unit size, single-file operations, batch operations (single-threaded/multithreaded), and amplification effect.

B. *Network Environment*

To provide a controllable network environment, we use a WAN emulator, i.e., WANem[3], to achieve the desired wireless conditions. We choose 100 Mbps (12.5 MiB/s) as the network bandwidth, which is around the peak data rate from a commercial wireless access point using 802.11n technique. The network parameters we investigate include network delay, packet loss, and packet jitter. For network delay, the values we select are based on real measurements from Amazon EC2 data centers. We choose 50 ms for WAN Internet scenarios, which is the Round Trip Time (RTT) between our test site and the nearest Amazon EC2 data center. It also represents the RTT between the US east-coast and west-coast data centers. We choose 0.1% packet loss rate for the Internet environment. Packet jitter means latency variation, which is common in the Internet environment. We choose 10 ms as the jitter value when applicable, which follows normal distribution. In the rest of the paper, "ideal network" means a near-perfect network condition (12.5 MiB/s bandwidth, no network delay, packet loss, or jitter), and "realistic network" refers to a suboptimal network condition (12.5 MiB/s bandwidth, 50 ms latency, 0.1% packet loss, and $\pm$10 ms jitter).

C. *Hardware Configuration*

The client machine is an Intel's Shark Bay development board (B0 step) with a quad core IvyBridge CPU clocked at 2.6 GHz, 4 GiB of memory, on-board Gigabit Ethernet interface, a Samsung MZ-7PD128 SSD as system disk and log storage. The server machine is a Supermicro server with an Intel Xeon X5570 quad core processor clocked at 2.93 GHz, 6 GiB of memory, on-board Gigabit Ethernet interface, four Western Digital WD2502ABYS 250 GB SATA HDDs. One HDD serves as the system disk, and the rest three each

[3]http://wanem.sourceforge.net/

backs one storage system. The WANem is deployed on a Dell laptop with an Intel Core2 Duo P8400 dual core processor clocked at 2.26 GHz, 4 GiB of memory, and on-board Gigabit Ethernet interface. The network switch is an ASUS RT-N66U home router with 250 MiB of memory. All Ethernet interfaces on the switch and network cables are capable of achieving 1 Gbps bandwidth despite the 100 Mbps limit configured in WANem.

D. Software Configuration

Both client and server machines run Fedora Core 17 with a patched Linux kernel version 3.2.1 [6]. The WAN emulator software is WANem 3.0 Beta 2. The network switch runs Tomato version 1.28[4]. The TCP congestion avoidance algorithm is CUBIC. To capture the number of packets accurately, all TCP offload engines on the client machine are disabled.

For NFS, we use the Linux kernel implementation and increase the number of `nfsd` threads on the server machine to 32. The NFS mount is a directory in an Ext4 partition and is exported with `sync` and `no_subtree_check` flags set. For iSCSI, we use the implementation by Intel [6] on both the client (the initiator) and server (the target) machines. We use the default block I/O scheduler in Fedora Core 17, i.e., Completely Fair Queuing (CFQ), if not stated otherwise. For Swift, we use the latest version on GitHub with Python 2.7.3. We configure it as in the official Swift All In One (SAIO) configuration, and use Ext4 as the host file system for storage. In order to provide the same operations as with NFS and iSCSI, we use CloudFuse on the client machine.

IV. Microbenchmark analysis

In this section, we analyze the performance of the three storage systems in fine granularity through a series of well-designed micro-benchmarks. We first analyze the effect of access unit size and forms of access on the performance. Based on the analysis, we choose a subset of access forms for further investigation. We then study the performance of single file access, which closely imitates the scenario where a small number of files are accessed. Thereafter, we analyze batch operations, which imitate the scenarios where a large number of files need to be synchronized with the remote storage. Both single-threaded and multithreaded batch operations are investigated to reveal performance variations from different degrees of parallelism. Finally, we examine the amplification effect on the performance of the systems. It is also worth noting that our results can be readily deduced to apply on energy-efficiency analysis of the the three systems, as the power consumption of wireless communications demonstrates good linear relationship with network throughput [7].

[4]http://www.polarcloud.com/tomato/

A. Forms of Access

When a file is accessed via the `open()` system call, a flag indicating the desired form of access can be passed. The most common flags are O_DIRECT, O_SYNC, and no flag, which specifies direct, synchronous, and default I/O operations, respectively.

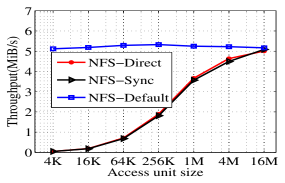
Direct I/O: The flag O_DIRECT informs the kernel to minimize cache effects of I/O operations. Such I/O operations bypass the caching facilities, e.g., page cache, along the I/O path in the kernel. Bypassing I/O cache entails that data must be read from or written to remote storage upon request. Direct I/Os enable a direct control on raw storage devices; nevertheless, no read-ahead or bundling is possible due to the absence of page cache. On the other hand, bypassing I/O cache can effectively decrease time-consuming memory copy operations and improve data reliability, which are desirable in certain scenarios, e.g., database operations.

Synchronous I/O: The flag O_SYNC causes `write()` operations to be blocked until dirty data are flushed to persistent storage. In the context of network storage systems, the physical storage is the storage service on the server machine. Whether the data are truly written to the physical medium on the server is another issue and depends on the server implementation. In this regard, synchronous I/Os and direct I/Os are similar for write operations; nonetheless, synchronous reads benefit from read-ahead and cache effects, which bring performance advantages for read operations.

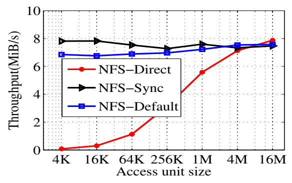
Default I/O: If no flag is set (`Default`), storage accesses are subject to the standard procedures for handling I/Os in OS kernel. System optimizations, e.g., caching and read-ahead, are in effect. Dirty data are periodically flushed to the persistent storage asynchronously, and power outage may result in data loss, which is a tradeoff for performance. Note that for Swift/CloudFuse, although Linux FUSE supports O_DIRECT and O_SYNC flags, CloudFuse does not take them into account as it operates on the local temporary file. Our preliminary measurements show that forms of access do not make any difference for performance. Thus, we only present results from `Default` access form for CloudFuse.

B. Access Unit Size

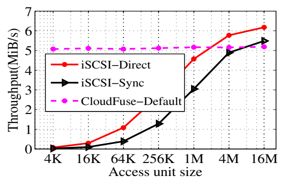
Besides the access forms, when a file (especially a large file) is accessed, the access unit size also plays an important role on performance. The smaller the access unit size is, the more I/O requests are generated, and accordingly, the more round trips are needed to access the same file. It is the opposite when bigger access unit sizes are applied. The experiments are conducted by accessing a relatively large 16 MiB file in different unit size, varying from 4 KiB through 16 MiB. We analyze all the three access forms. Note for CloudFuse, as stated in Section IV-A, the three access forms do not present performance difference; thus, only the `Default` access form is analyzed in this paper. The network configurations for the experiments are 12.5 MiB/s speed and 50 ms delay. The results are illustrated in Figure 2.



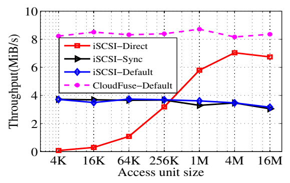
(a) NFS – Write.



(b) NFS – Read.



(c) iSCSI + CloudFuse – Write.



(d) iSCSI + CloudFuse – Read.

Figure 2: Impact of access unit size and form of access on throughput. Note that the curve for iSCSI `Default Write` operations is missing from subfigure (c), because it only writes to local cache and the throughput is of several hundred MiB/s.

From the figure, we can see that for `Direct` I/O operations (red curves), the overall trend is that the throughput increases along with the access unit size. This trend applies to NFS and iSCSI, including both `Write` and `Read` operations. Furthermore, from Figure 2a and 2c, we can see that `Direct` I/O operations perform slightly better than `Sync` operations. The trend is more evident in Figure 2c. This is mostly due to the different cache implications of the three access forms, as explained in Section IV-A.

For `Sync` I/O operations (black curves), as shown in Figure 2a and 2c, the performance of `Write` operations is similar to that of `Direct` I/O operations. While for `Read` operations (cf. Figure 2b and 2d), `Sync` can benefit from read-ahead effect; thus, the throughput remains unaffected despite the access unit size. The abnormally low throughput for iSCSI `Read` operations with `Sync` (and `Default`) access form is because of the access pattern. The read request is issued only after the response for the previous request has been received; thus, network delay dominates the overall completion time and becomes the bottleneck.

For `Default` I/O operations (blue and pink curves), since they are not obliged to bypass caches or to sync up with remote storage, they can benefit from bundling (write-back) for `Write` operations and also read-ahead for `Read` operations. The overall trend is that the throughput is not affected by the access unit size, which is applicable to both NFS and iSCSI. For iSCSI `Write` operations, the I/O returns immediately after the dirty data is written to the local cache, which is effectively at memory speed and thus not shown in Figure 2c. For CloudFuse, although the behaviour is similar to NFS and iSCSI (i.e., also benefit from write-back and read-ahead), the underlying cause is different. CloudFuse always retrieves the entire file into a temporary file in advance, or uploads the entire temporary file after all local file operations are completed. Thus, it can achieve the same effect as the other two systems. Note that commercial cloud storage clients (e.g., DropBox) also adopt similar policies (retrieve a complete copy of remote objects and store it locally for accesses) to boost performance.

Table I summarizes the behaviours of the access forms for the three systems.

Storage system	**Form of access**	**Write**	**Read**
NFS	`Direct`	write-through	read-through
	`Sync`		read-ahead
	`Default`	write-back	
iSCSI	`Direct`	write-through	read-through
	`Sync`		read-ahead
	`Default`	buffered	
Swift + CloudFuse	`Direct`	write-back	read-ahead
	`Sync`		
	`Default`		

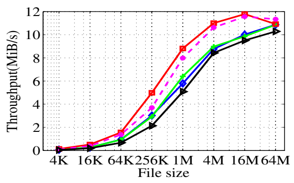
Table I: Writing/reading policies of different access forms.

C. Single File Access

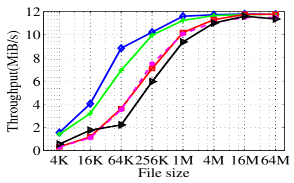
In the previous section, we analyzed the impact of access unit size within large files. Another common practice is to access files in their entirety, especially for small files that are dominant in modern file systems. Certain systems also bear size limits for the files allowed. For example, the object sizes of Dropbox vary between 4 KiB and 4 MiB; files larger than 4 MiB are sliced into 4 MiB chunks before uploading to the cloud [8]. In the following sections, we will investigate accessing files in their entirety with various sizes, ranging from 4 KiB to 64 MiB (stepping at powers of 4).

In the face of potential interference from bundling or parallelism, in order to gain insights into the intrinsic protocol design, we design our experiments of single file access to imitate the scenario where an end user has a small number of files to access. Thus, potential bias from bundling or parallelism is effectively isolated. Figure 3 illustrates the results. Note for NFS and iSCSI, we only present results from `Direct` and `Sync` access forms, which will remain the same for the remaining figures in this paper, to provide a clear presentation. This is because when the access unit size equals the file size, the performance of NFS `Default` operations follows closely with that of `Sync` access form; while for iSCSI, its performance of `Default Read` operations follows that of `Sync` operations, and `Default Write` still completes to local buffer (thus of no interest).

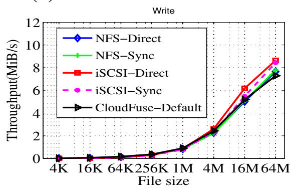
From Figure 3, we can see that the common trend for the three systems is that throughput increases with file sizes. Performance of small files suffers because their transmission completes when the TCP connection is still in slow start phase. Comparing the three systems, it is clear that under ideal network, for `Write` operations (cf. Figure 3a), iSCSI



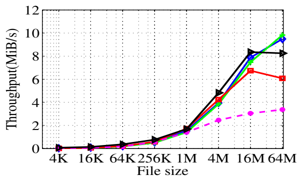
(a) Write – ideal network.



(b) Read – ideal network.



(c) Write – added 50 ms delay.



(d) Read – added 50 ms delay.

Figure 3: Single file access in different network conditions.

outperforms NFS and CloudFuse slightly; while for `Read` operations (cf. Figure 3b), NFS performs the best, and iSCSI is comparable with CloudFuse in throughput. When 50 ms delay is introduced, the overall trend is that the throughput of iSCSI deteriorates faster than the other two systems. In consequence, the head start of iSCSI `Write` operations disappears (cf. Fig. 3c); for `Read` operations iSCSI lags behind the other two systems with a greater gap (cf. Fig. 3d).

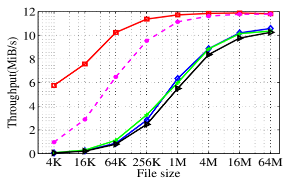
Furthermore, in Figure 3d, we observe the same behaviour for iSCSI `Sync Read` operations as shown in Section IV-B (cf. Figure 2d). For files larger than 1 MiB, they require multiple SCSI command cycles to complete. Thus, their performance is significantly impacted by network delay as the requests are issued in a sequential manner.

Summary: for single file access that imitates the scenario where there are a small number of files to access, the performance variation of the three systems is not significant; under ideal network, NFS and iSCSI lead the `Read` and `Write` performance respectively; when network delay is introduced, iSCSI deteriorates faster than the other two systems, especially for `Read` operations.

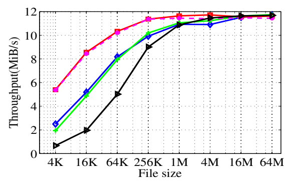
D. Batch Operation – Single-threaded

In this section we analyze batch operations, where a set of files of the same size is accessed in series, and the throughput is calculated from the steady state. Batch operations imitate the scenario where a large number of files need to be accessed, e.g., in backup services. Figure 4 illustrates the performance of single-threaded operations under different network conditions.

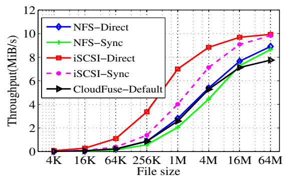
From Figure 4a and 4b, we see that iSCSI excels under ideal network and NFS performs slightly better than CloudFuse. Comparing Figure 4a with 3a, and Figure 4b with 3b, we can clearly see the benefits of block-level access protocol over the file-level and object-level protocols. iSCSI benefits from its internal bundling mechanisms, which improve the throughput by several orders of magnitude for small files.



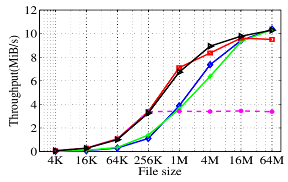
(a) Write – ideal network.



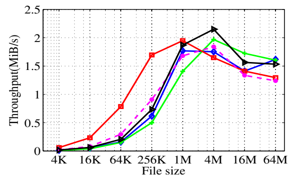
(b) Read – ideal network.



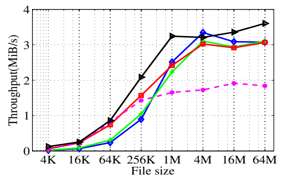
(c) Write – added 50 ms delay.



(d) Read – added 50 ms delay.



(e) Write – realistic network.



(f) Read – realistic network.

Figure 4: Performance of single-threaded batch operations. Note the different scales for (e) and (f).

When network delay is introduced (cf. Figure 4c and 4d), we can see that iSCSI is impacted severely; while CloudFuse is impacted the least. Thanks to the bundling effect, iSCSI `Write` operations (cf. Figure 4c) still outperform the other two systems in the face of performance deterioration; while for `Read` operations, CloudFuse starts to lead the three systems (from the worst in Figure 4b to the best in Figure 4d).

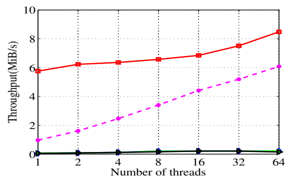
When packet loss and network jitter are introduced beyond the 50 ms delay, as shown in Figure 4e and 4f, the overall trend is that the three systems are all impacted significantly. This is because packet loss degrades TCP throughput remarkably when network delay exists. From our experiments, packet loss is the primary culprit for throughput degradation, while network jitter incurs performance variation instead. Comparing the three systems, we can see that except for small files `Write` operations where iSCSI continues to keep ahead, CloudFuse performs the best for the other cases, including `Write` operations for large files and `Read` operations for all files.

The peculiar leveling of iSCSI `Sync Read` performance in Figure 4d and 4f (when file sizes are larger than 256 KiB) is due to the way file requests are issued, as discussed in Section IV-B. One interesting observation from Figure 4a, 4c, and 4e is that iSCSI `Direct Write` operations greatly outperform `Sync Write` operations. This is primarily from the amplification effect, which will be analyzed in Section IV-F. Another cause is that `Direct` I/O operations bypass cache facilities; thus, it avoids the costly memory copy operations and improves performance accordingly.

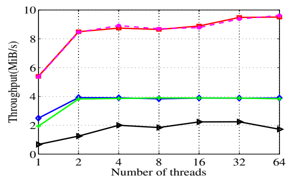
Summary: for single-threaded batch operations, iSCSI delivers the best performance for small file access under all network conditions; throughput of iSCSI and NFS is impacted by network delay and packet loss significantly, while CloudFuse is impacted the least; in realistic network, CloudFuse outperforms the other two systems for most scenarios except small file `Write` operations.

E. Batch Operation – Multithreaded

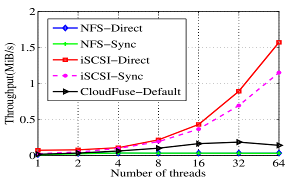
In this section, we investigate batch operations with multiple threads. File sets are accessed sequentially by multiple threads concurrently. This is to imitate applications that employ multiple threads (or processes) to service queued I/O requests. The number of threads ranges from 1 to 64, increasing by a power of 2. Note that after introducing the number of threads as the new dimension, we only select two representative file sizes for focus study, i.e., 4 KiB, and 16 MiB. 4 KiB represents small files, while 16 MiB represents large files.



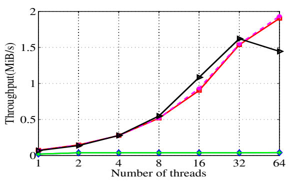
(a) Write – ideal network.



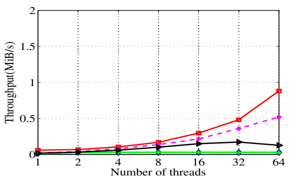
(b) Read – ideal network.



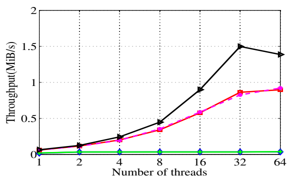
(c) Write – added 50 ms delay.



(d) Read – added 50 ms delay.



(e) Write – realistic network.



(f) Read – realistic network.

Figure 5: Performance of multithreaded batch access of 4 KiB files. Note the different scales for (a) and (b).

1) Small Files – 4 KiB: Performance of the three systems accessing 4 KiB files is illustrated in Figure 5. Note that the close to zero results for NFS and CloudFuse in certain subfigures are dwarfed by the relatively high readings from iSCSI. A general trend is that the throughput increases along with the number of threads, which is especially true for iSCSI and CloudFuse. iSCSI performance increases because access requests from different threads are merged to form larger SCSI accesses. CloudFuse performance grows up to 32 threads and decreases slightly at 64 threads. We conjecture the slight decrease is because of the overhead from coordinating multi-tasking and maintenance of multiple TCP connections. For NFS, we find that its throughput increases till 2 or 4 threads; after that, it remains at a stable value, which is clearly shown in Figure 5b.[5] This is likely from limitations of the NFS client, because the number of `nfsd` threads on the server is set to 32 in our setup.

Comparing the three systems, a similar trend to that in Figure 4 (for small files) is observed. For `Write` operations, iSCSI demonstrates dominating advantage over NFS and CloudFuse, because of its intrinsic block-level access schemes. The advantage is especially dominant for the `Direct` access form (cf. Figure 5a). Despite the large performance degradation from network delay and loss, its head start advantage makes it lead the three systems in realistic network still (cf. Figure 5e).

For `Read` operations, under ideal network condition, iSCSI still dominates, while NFS is twice as fast as CloudFuse (cf. Figure 5b). We find that the weak performance of CloudFuse under ideal network is because of the internal cooperation among the Swift servers. In Swift, multiple server processes are involved for each incoming request. The cooperation among the server components is around 30 ms for each request, which is not trivial under ideal network conditions. When 50 ms delay is introduced, the network becomes the dominating factor and CloudFuse starts to outperform NFS and iSCSI, as shown in Figure 5d. The outperformance of CloudFuse is even clearer in realistic network, as shown in Figure 5f. By monitoring the network traffic and examining the implementations, we find that both NFS and iSCSI multiplex the same TCP connection for multiple threads, while CloudFuse uses multiple TCP connections instead. Figure 5d and 5f clearly demonstrate the strengths of multiple against single TCP connection in adverse network, which will be revealed more in the subsequent section.

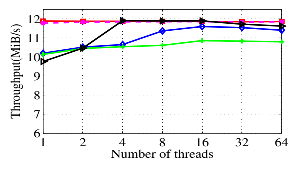
2) Large Files – 16 MiB: Figure 6 shows the performance of the three systems accessing files of size 16 MiB. In theory, similar trends from small files are expected for big files, because the protocols remain the same regardless of file sizes. From Figure 6, however, slightly different trends are observed compared with Figure 5. Under ideal network condition (cf. Figure 6a and 6b), we can see that the performance of the three systems is comparable, which is significantly apart from that in Figure 5a and 5b. This is because when the file size is large, i.e., 16 MiB in our experiment, network bandwidth becomes the bottleneck and constrains the capability of good-performing systems, such as iSCSI. Consequently, the three systems are on par with each other under ideal network.

When network delay is introduced, compared with small

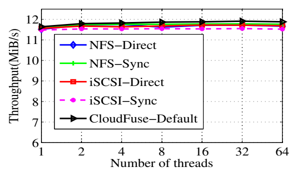
[5]This holds true for all the subfigures in Figure 5, although because of the presentation, it is not clearly visible in the other subfigures.

files, its impact is much smaller for large files. This can be explained by the fact that large files can still roughly saturate the link given the network delay (i.e., 50 ms). When packet loss is further added, we can see that the throughput of NFS and iSCSI is significantly degraded (cf. Figure 6e and 6f). On the contrary, CloudFuse still saturates the link capacity when more than 8 threads are used. Thus, its throughput dwarfs NFS and iSCSI clearly. As stated before, CloudFuse starts a new TCP connection for each thread. If one of the TCP connections is backed off due to packet loss, the other TCP connections can compensate for this. As a whole, the multiple TCP connections can make full use of the link capacity with great efficiency.

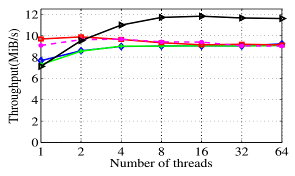
An interesting phenomenon from Figure 6d is the throughput of iSCSI `Sync Read` operations levels despite the number of threads. This peculiarity is due to block I/O schedulers, which is out of scope of this paper.



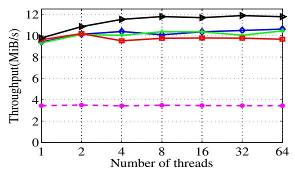
(a) Write – ideal network.



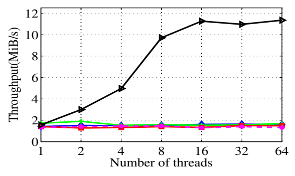
(b) Read – ideal network.



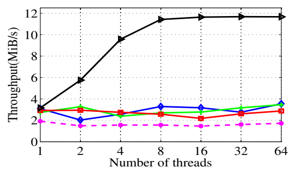
(c) Write – added 50 ms delay.



(d) Read – added 50 ms delay.



(e) Write – realistic network.



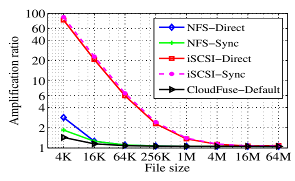
(f) Read – realistic network.

Figure 6: Performance of multithreaded access – 16 MiB.

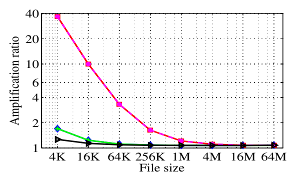
3) Summary: From the analysis of small and large files with multithreaded operations, we make the following observations: (1) the performance of iSCSI for small files demonstrates the great benefit of the intrinsic block-level access approach, which also shows the improvement headroom for NFS and CloudFuse by leveraging bundling schemes for small files; (2) the performance of CloudFuse for large files under realistic network illustrates the advantage of multiple TCP connections against single TCP connection design, which also indicates the optimization potential for NFS and iSCSI under adverse network conditions by using multiple TCP connections.

F. Amplification Effect

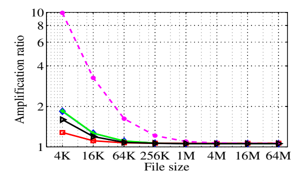
From previous sections, we see that for iSCSI `Write` operations, there exists a throughput gap between the `Direct` and `Sync` access forms. In addition, we also notice an interesting phenomenon during our experiments, which is the high volume of transmission incurred by iSCSI `Sync Write` operations when the file is small. In certain cases, it saturates the 12.5 MiB/s link while delivering very low throughput (around 1 MiB/s). This is due to the amplification effect caused by ext4 metadata and journaling. In this section, we analyze this amplification effect. To make a fair comparison among the three storage systems, we introduce a new metric, i.e., **amplification ratio**, to describe the effect. For single file access, the amplification ratio is calculated by dividing the total transmission volume (in both directions) by the file size; while for batch operations, it is computed by dividing the sum of transmission rates (in both directions) by the achieved throughput. Thus, the amplification ratio has an inverse correlation with **storage efficiency**. The amplification ratios for the three systems in ideal network are illustrated in Figure 7.



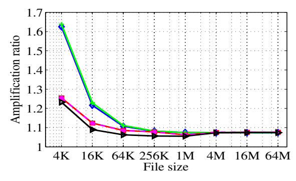
(a) Single file access – Write.



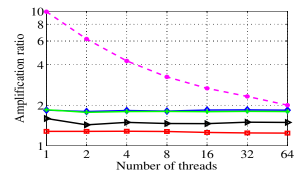
(b) Single file access – Read.



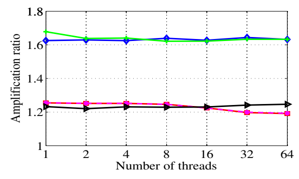
(c) Single-threaded – Write.



(d) Single-threaded – Read.



(e) Multithreaded access for 4 KiB files – Write.



(f) Multithreaded access for 4 KiB files – Read.

Figure 7: Amplification effect of file access under ideal network. The two curves from iSCSI overlap in almost all `Read` plots. Note the different scales in the figure.

From the figure, we can see that the general trend for NFS is that the amplification ratio slightly decreases from single file (cf. Figure 7a and 7b) to single thread access (cf. Figure 7c and 7d), while it remains stable when moving from single-threaded to multithreaded access (cf. Figure 7e and 7f). For CloudFuse, we do not find any perceivable difference for the three cases (single file $\rightarrow$ single-threaded $\rightarrow$ multithreaded access). This is because of the intrinsic behaviour of the protocols. As stated in Section II, NFS uses RPC while CloudFuse utilizes HTTP protocol for communications. Despite their different implementation, in essence, they share much similarity with each other – for each operation, at least one request/response pair is needed, and no bundling is used for either of them. Thus, the storage efficiency does not improve with the number of files transmitted concurrently (i.e., multithreaded), but the transmission throughput does increase (cf. Figure 5). Through packet capturing analysis, we find that NFS overhead consists of RPC protocol messages and is approximately several KiBs in size, while CloudFuse requires only a few HTTP requests and thus incurs the lowest overhead.

Interestingly, iSCSI demonstrates completely different behaviours, because of its fundamentally different structure. Its amplification ratio decreases significantly from single file (cf. Figure 7a and 7b) to single-threaded batch access (cf. Figure 7c and 7d), which applies to both `Write` and `Read` operations. Recall from Figure 3 and 4 that the throughput of iSCSI also increases remarkably when moving from single file to single-threaded operations. These two phenomena together clearly illustrate the great intrinsic benefits from the block-level access scheme, as both throughput and storage efficiency (the opposite of amplification ratio) increase side by side. When moving from single-threaded to multithreaded access, on the other hand, only the amplification ratio for `Sync Write` operations decreases (cf. the pink dashed line in Figure 7e), while the other operations remain relatively constant. Recall from Figure 5a that the throughput of `Sync Write` operations increases quickly along with the number of threads. We conjecture the constant amplification ratio for `Direct Write`, `Direct Read`, and `Sync Read` multithreaded operations is because one thread is sufficient to stuff one SCSI command with file operations. While for `Sync Write` operation, besides the file itself, it also needs to transmit the required metadata and journal at the same time. Thus, one thread is not enough to stuff the SCSI command; adding more threads in turn increases both the throughput and storage efficiency.

The following are quantitative results for iSCSI that we observe through packet capturing analysis. For single file access, we find that for iSCSI `Read` operations, in addition to the file itself, there are always additional $40+92$ KiBs data read from the iSCSI target. After the file is read, another $8+4+4=16$ KiBs data are written back. These are the Ext4 metadata. For iSCSI `Write` operations, besides the file, we find that there are $4*37=148$ KiBs data read from the target and $28+7*4=56$ KiBs written back. These are the associated metadata and journal entries for the files written to the target. We have the following equations for iSCSI amplification ratio:

$$A^r(s) = \frac{148+s}{s} \quad \text{and} \quad A^w(s) = \frac{204+s}{s} \tag{1}$$

for read and write, respectively, where s is the size of the file in KiBs. Note that these empirical values are a result of various factors. It is anticipated that different system configurations, and file system, may affect the values.

For batch file access (including single-threaded and multithreaded), we observe that the significant decrease of amplification ratio is due to the sharing of metadata (e.g., inode blocks, and dentry) of files. For iSCSI `Direct Write` operations, the extreme low overhead is because its metadata and journal are written in large batches, which is around 2 MiB per 10,000 files. On the contrary, for iSCSI `Sync Write` operations, each I/O data block incurs updates from the corresponding journal (usually with fixed size, e.g., 32 KiB) and metadata (e.g., 4 KiB). Thus, we can use the following equation to estimate the amplification ratio for single-threaded `Sync Write` operations:

$$A^w_s(s) = \frac{journal + metadata + s}{s} \tag{2}$$

where s is the file size in KiBs. For example, if the file size is 4 KiB, then the amplification ratio is $(32+4+4)/4 = 10$ (cf. Figure 7c). For multithreaded access, we find the following equation provides a close estimation:

$$A^w_m(s) = \frac{journal + metadata + n \cdot s}{n \cdot s} \tag{3}$$

where n is the number of threads, and s is the file size in KiBs. Note that the equations demonstrated above only account for the overhead contained in SCSI Command Descriptor Blocks (CDBs). When overhead from network protocols (e.g., TCP) are taken into account, the numbers are slightly larger, as shown in Figure 7.

Summary: The write-back and caching effect with iSCSI makes it excel in storage efficiency for most batch operations, which is comparable to that of CloudFuse, while NFS is slightly worse than them. For `Sync Write` operations, iSCSI suffers from its high amplification effect especially for small files; on the other hand, `Sync Write` operations can guarantee their high consistency despite adverse network conditions. Thus, the tradeoff between storage efficiency and consistency needs to be considered carefully.

V. Related Work

IP-based storage systems have been introduced for a long time. Nevertheless, most of the existing studies have focused on analysing a single family of storage system, rather than comparing multiple ones. Aiken et al. [9] analysed the performance of iSCSI protocol under different configurations. Similarly, Xinidis et al. [10] evaluated the performance of commodity iSCSI storage systems in comparison to direct-attached storage. Zhang et al. [11] analysed the performance of an open-source iSCSI solution over high-latency network and proposed schemes to improve it.

On the other hand, as cloud storage systems start to gain popularity, several studies has been conducted. Drago et al. [8] presented a characterisation study of Dropbox by utilising data collected from four vantage points in Europe. Following the same line of study, they extended their methodology to cover another four cloud storage systems [12]. In contrast, Naldi et al. [13] compared the pricing plans of the major cloud providers. Zhang et al. [14] proposed integrating local file system with cloud storage to improve resilience from data corruption and inconsistency. Carpen-Amarie et al. [15] evaluated the feasibility of cloud storage on high performance computing workloads.

While the aforementioned studies presented good insights into understanding different aspects of storage systems, they are orthogonal to our study to a great extent. The most relevant work is from [1], where Radkov et al. provided a comparative study on NFS and iSCSI. Nevertheless, there exist two key differences that clearly differentiate our work from [1]: (1) Radkov et al. compared NFS and iSCSI, whereas our work provides a systematic study on NFS, iSCSI, and the emerging cloud storage; (2) Radkov et al. mainly focused on metadata, and provided only simple analysis on single-threaded file operations; while our work provides comprehensive analysis covering various aspects of the three storage systems, ranging from single file, single-threaded, and multi-threaded access; furthermore, different access forms, and varied file sizes are also compared.

VI. Conclusions and Future Work

The emerging cloud storage gains significant popularity in recent years. Certain enterprises have considered replacing conventional network storage systems with cloud storage solutions. Nevertheless, the lack of a thorough study among these systems makes it difficult to make an all-around decision. In this paper, we presented a systematic study on the three drastically different storage systems. Through a comprehensive set of experiments, we made several interesting observations: (1) iSCSI excels in typical LAN environment, whereas NFS and Swift are more suitable for complex networks such as wireless networks and WAN; furthermore, we find out that Swift is a viable replacement for NFS in all network scenarios. (2) Under realistic network condition, performance of iSCSI declines considerably, whereas Swift demonstrates high resilience. The key behind this is the capability to utilize multiple TCP connections.

In the future, we will evaluate the storage systems in Gigabit networks. Other metrics, such as performance per dollar, reliability, and elasticity, will be used to evaluate the viability of replacing NAS or SAN with cloud storage.

Acknowledgment

This work was supported by the Academy of Finland, with grant number 278207; and was supported in part by Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, U.S. National Science Foundation (NSF) under grant CCF-1453705, and generous support from Intel Corporation.

References

[1] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy, "A performance comparison of NFS and iSCSI for IP-networked storage." in *USENIX FAST '04*, pp. 101–114.

[2] A. Systems, "Avere public FlashCloud storage challenges traditional NAS," http://www.digitalmedia-world.com/Storage/avere-flashcloud-public-cloud-storage-challenges-traditional-nas.html/, 2014.

[3] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network file system (NFS) version 4 protocol," in *IETF RFC 3530*, 2003.

[4] M. Chadalapaka, J. Satran, K. Meth, and D. L. Black, "Internet small computer systems interface (iSCSI)," in *IETF RFC 3720*, 2004.

[5] R. Thurlow, "RPC: Remote procedure call protocol specification version 2," in *IETF RFC 5531*, 2009.

[6] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated storage services," in *ACM SOSP '11*, pp. 57–70.

[7] Z. Ou, J. Dong, S. Dong, J. Wu, A. Ylä-Jääski, H. Pan, R. Wang, and A. W. Min, "Utilize signal traces from others? A crowdsourcing perspective of energy saving in cellular data communication," *IEEE Transactions on Mobile Computing*, vol. 14, no. 1, pp. 194 – 207, 2015.

[8] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding personal cloud storage services," in *ACM IMC '12*, pp. 481–494.

[9] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke, "A performance analysis of the iSCSI protocol," in *IEEE/NASA MSST '03)*, pp. 123–134.

[10] D. Xinidis, A. Bilas, and M. D. Flouris, "Performance evaluation of commodity iSCSI-based storage systems." in *IEEE/NASA MSST '05*, pp. 261–269.

[11] Y. Zhang and M. H. MacGregor, "Tuning Open-iSCSI for operation over WAN links," in *The Ninth Annual Communication Networks and Services Research Conference (CNSR '11)*, pp. 85–92.

[12] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *ACM IMC '13*, pp. 205–212.

[13] M. Naldi and L. Mastroeni, "Cloud storage pricing: A comparison of current practices," in *HotTopiCS '13*, pp. 27–34.

[14] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ViewBox: Integrating local file systems with cloud storage services," in *FAST '14*, pp. 119–132.

[15] A. Carpen-Amarie, K. Keahey, J. Bresnahan, and G. Antoniu, "Evaluating cloud storage services for tightly-coupled applications," in *Euro-Par '12 Workshops*, pp. 36–46.