Aalto University
School of Science
Master's Programme in Mobile Computing – Services and Security

Zhen-Huan Hwang

Benchmarking of IP-based Network Storage Systems

Master's Thesis
Espoo, October 20, 2014

Supervisor: Professor Antti Ylä-Jääski
Advisor: Doctor Zhonghong Ou

Most materials produced in the course of this thesis are available at

`https://github.com/zwuh/iotest/`

Typeset in Latin Modern (LM) fonts by the author using LaTeX on Linux.
The TeX template used is available at `https://github.com/zwuh/aalto-thesis/`

Printed in Finland.
Bound by Kirjansitomo V. & K. Jokinen in Helsinki.

Aalto University
School of Science
Master's Programme in Mobile Computing – Services and Security

Aalto University
School of Science

ABSTRACT OF
MASTER'S THESIS

Author:	Zhen-Huan Hwang	
Title:	Benchmarking of IP-based Network Storage Systems	
Date:	October 20, 2014	**Pages:** x + 66
Major:	Mobile Computing – Services and Security	**Code:** T-110
Supervisor:	Professor Antti Ylä-Jääski	
Advisor:	Doctor Zhonghong Ou	

Mobile platforms with access to high speed wireless network have become ubiquitous. Advancements in network technology and consumer electronics have brought traditional storage systems into offices and homes. Services based on cloud technologies, including object based storage, have gained popularity among both private users and enterprises. However, there is still a lack of systematic evaluation of both traditional storage systems and cloud based object storage in a mobile and wireless context.

In this thesis, we evaluate the performance of three drastically different storage systems, namely NFS, iSCSI, and OpenStack Swift, which can potentially be used by mobile platforms over wireless network. We build a testbed and an in house, ad hoc microbenchmark to study the impact of various network complexities and different access behaviours of application. In addition, we employ two widely used macrobenchmarks – PostMark and FileBench – to simulate the workloads of typical applications.

We find that: (1) iSCSI excels in networks whose condition is as good as LAN; (2) NFS and Swift are more suitable for complex networks such as wireless network and WAN; (3) Swift is a viable replacement for NFS in all scenarios; and (4) System configuration on the client side impacts storage performance significantly and deserve adequate attention. Furthermore, we make several recommendations to practitioners and point out numerous future research directions.

Keywords:	network storage, benchmarking, performance, throughput, cloud computing, energy efficiency, Linux, NFS, iSCSI, OpenStack Swift
Language:	English

Contents

List of Figures

List of Tables

Acknowledgements

The work reported in this thesis was conducted in cooperation with Intel Labs, Hillsboro, Oregon, United States of America. I would like to thank Professor Feng Chen at Louisiana State University, United States of America, Doctor Ren Wang and others at Intel Labs for their inspiring discussions and suggestions.

In addition, I would also like to thank the reviewers of 2014 ACM Symposium on Cloud Computing (SoCC'14) who provided us with precious comments.

I am grateful to Doctor Zhonghong Ou and Professor Antti Ylä-Jääski for their guidance and insightful comments. In particular, thanks to Dr. Ou for his involvement at various stages of the study.

Thanks to my fellow students and colleagues for making the days at Aalto enjoyable.

Helsinki, October 20, 2014

Zhen-Huan Hwang

感謝家宜的陪伴與支持。

Abbreviations and Acronyms

API	Application Programming Interface
BDP	Bandwidth Delay Product
CFQ	Completely Fair Queuing
CPU	Central Processing Unit
DAS	Direct Attached Storage
FIFO	First-In First-Out
FUSE	File System in User Space
GiB	Gigabytes (2^{30} bytes = 1024 MiB)
HSPA	High Speed Packet Data Access
HTTP	Hypertext Transfer Protocol
I/O	Input and/or Output
IOPS	I/O Operations Per Second
IP	Internet Protocol
iSCSI	Internet Small Computer System Interface
KiB	Kilobytes (2^{10} = 1024 bytes)
LAN	Local Area Network
LBA	Logical Block Address
LTE	Long Term Evolution
LWP	Light Weight Process
MiB	Megabytes (2^{20} bytes = 1024 KiB)
NAS	Network Attached Storage
NFS	Network File System
NFSv4	Network File System version 4
POSIX	Portable Operating System Interface
RPC	Remote Procedure Call
RTT	Round Trip Time
SAN	Storage Area Network
Swift	OpenStack Swift
TCP	Transmission Control Protocol
VFS	Virtual File System
WAN	Wide Area Network

Chapter 1

Introduction

Network storage systems have been widely deployed in enterprises and organisations since their introduction in 1980s [60]. Their performance and reliability have been proven in practice. Network storage was developed as a solution to enable sharing of storage devices attached directly to computers, which were traditionally private and not shared. Traditional network storage is dominated by two major families – network attached storage (NAS) and storage area network (SAN).

SAN shares the storage device itself and provides block level access. The actual organisation of data stored in the device and the synchronisation between concurrent access are the duty of the clients. On the other hand, NAS provides access to file systems deployed on the storage devices. The interface of access is file oriented and provides a subset of the features of a local file system. The server handles the physical organisation of data and coordinates concurrent access. A set of storage providers and a set of clients then form a "storage network".

Traditional NAS and SAN are often deployed over wired and dedicated networks. Moreover, SAN systems are generally deployed in data centres. The emergence of storage appliances and developments in consumer network electronics have led to network storage systems deployed at homes and in offices.

Mobile platforms, such as laptop, smart phone, and tablet, with increasing computing power have become ubiquitous. Combined with the advancements in wireless network technologies and growing accessibility to high speed wireless network, such as IEEE 802.11n [33], HSPA [69], and LTE [70], the current abundance of applications is a natural result [54]. Besides being deployed in offices and at homes, network storage systems begin to be accessed by mobile platforms through wireless networks.

1.1 Motivation

The computing power of mobile platforms, the speed of wireless networks, and the complexity of applications have grown substantially. However, battery capacity has seen only limited improvements. Mobile platforms are powered by batteries with limited capacity. Energy efficiency has thus become the salient concern in mobile computing.

Storage subsystem is key to virtually any kind of application or service based on information technology. Energy efficiency of the storage subsystem is therefore technically and economically paramount for mobile applications. Re-

search indicates that energy consumption of mobile platforms greatly depends on computation intensity and wireless network throughput [54, 55]. With the computing power of modern mobile platforms, storage activities over wireless network are generally transmission bound. The performance of storage subsystem thus depends on the throughput of transmission over wireless network. It is therefore possible to investigate the energy efficiency of storage subsystem by measuring transmission throughput. Moreover, performance itself is of enough interest to users of storage systems.

Services based on cloud computing, including storage, have gained popularity in recent years [15]. Cloud computing features scalability, elasticity, easy to use, and pay per use [42], which make cloud based object storage services a viable choice for both individual and enterprise users. Emerging object based [3, 43] cloud storage services have gained popularity with its attractive features among both individual users and enterprises. Moreover, road warriors, working from home, and high speed wireless network in office are all common in contemporary business. The energy efficiency and performance of using cloud based object storage on mobile platforms therefore worth investigation.

There are abundant studies on traditional storage systems. However, few of them studied in the context of a mobile platform accessing storage services through wireless links. Furthermore, there is no systematic comparison of contemporary IP-based storage systems that included both traditional systems and cloud based object storage.

1.2 Problems and Scope

In this thesis we evaluate the performance of three drastically different network storage technologies in different client side environments and under emulated network conditions. We try to gain insights on the following questions:

- Are traditional NAS and SAN suitable in mobile and wireless scenarios?
- Is cloud based object storage a viable replacement for NAS and SAN?
- If the suitability and viability are not universal, what are the strengths and the limitations?

Furthermore, performance evaluation also allows us to indirectly infer the energy efficiency of the network storage technologies.

The user story we assume and simulate is that of an individual user. There are read and write access of files and metadata operations. Furthermore, we assume exclusive access to the storage service from a single device. That is, no concurrent access of stored objects or files from other sessions, no need of consistency across devices, and no security concerns.

We choose Network File System (NFS) version 4 [56, 65] and Internet Small Computer System Interface (iSCSI) [10, 45] as the representatives of NAS and SAN, respectively. NFS is famous and widely used by organisations to share

file systems located on their storage servers. It is built into most popular open source operating systems and is readily available. SCSI is the standard protocol of storage hardware such as disks. iSCSI allows SCSI commands to be exchanged over TCP/IP network. It is widely used to allow access to hard disks attached to remote hosts or specialised storage devices.

Open source implementations as well as commercial implementations for both NFS and iSCSI are available. For example, Linux kernel is bundled with an open source implementation of NFS and there are open source implementations of iSCSI available as optional packages for most Linux distributions. Commercial implementations and storage appliances for NFS and iSCSI are also available in the market.

Popular cloud storage services are often proprietary (see Appendix A), meaning that it is usually impossible to set them up in a laboratory environment. Among those who are open source, we choose OpenStack Swift [53] ("Swift" hereafter) as the representative for cloud based object storage. It is an integral component of the OpenStack cloud computing suite [52] which has gained popularity among enterprise users. There are even companies dedicated to providing services for Swift[1]. This entails that by investigating its performance, it is possible to benefit a wide range of users and applications.

We investigate the performance of the three storage systems in three main directions. Specifically, we investigate the impact of:

- different access approaches employed by applications
- client environment
- network condition

on network storage performance in this thesis.

It should be noted, nevertheless, that the coverage of this thesis is limited. We focus on application throughput and not other aspects, such as CPU usage and number of packets. More detailed discussion is provided in Chapter 7.

1.3 Methodology

The primary methodologies employed in this thesis are experimenting, measurement, and data analysis. We use an in house, ad hoc microbenchmark [73] and two popular macrobenchmarks – PostMark [36] and FileBench [67] – to evaluate the performance.

The microbenchmark conducts file read and write operations through POSIX file API[2] and metadata operations via command line. For file operations we evaluate the impact of file size, access unit size, different forms of access, and the number of worker threads. Other aspects such as traffic overhead and operating system specific behaviours are also examined.

[1]Such as SwiftStack (`http://swiftstack.com/`).

[2]Official website of POSIX working group: `http://www.opengroup.org/austin/`

Specifically, we simulate applications which cache files locally and synchronise with remote storage service periodically. Microbenchmarking experiments mainly focus on batch operations which is similar to the access pattern of such applications. The metric in these experiments is the transmission throughput for file operations and completion time for metadata operations.

Macrobenchmarking experiments, on the other hand, cover applications which access the storage service interactively. Macrobenchmarks are configured to simulate the behaviours of some typical applications which an individual user might use. The metric in macrobenchmarking experiments is completion time for PostMark and IOPS for FileBench.

The benchmarks are run on a custom built testbed which consists of three computers and an Ethernet switch. One of the computers is the network emulator which is capable of simulating network complexities often seen in the Internet, such as latency, loss, and jitter. The other computers are the client machine and the server machine.

System states, such as time and network interface counters, are collected from numerous sources as applicable. Collection method includes invoking C library routines and accessing Linux kernel counters, either through `/proc` pseudo file system or through system utilities such as `date`. In addition, `tcpdump`[3] is used in metadata experiments to observe network traffic.

We rely on MySQL[4], PHP[5], and Octave[6] for data analysis.

1.4 Thesis structure

In Chapter 2 we briefly introduce the storage technologies studied in this thesis and provide an overview on related work.

Detailed description of the testbed, including hardware and software configurations and settings of various experimental variables related to our emulated network, is provided in Chapter 3.

Chapter 4 and 5 are devoted to our custom microbenchmark. Design of the microbenchmark, the set of different access approaches, and experiments related to file operations are first discussed in Chapter 4. Metadata operations are discussed in Chapter 5.

Macrobenchmarking experiments, which evaluates the performance of the storage technologies under workloads similar to typical applications, are discussed in Chapter 6.

Based on the results we obtained, we compare the three storage technologies investigated and discuss some future research directions in Chapter 7. Finally, we finish this thesis with conclusions in Chapter 8.

[3]Official website: `http://www.tcpdump.org/`
[4]Official website: `http://www.mysql.com/`
[5]Official website: `http://php.net/`
[6]Official website: `https://www.gnu.org/software/octave/`

Chapter 2

Background

In this chapter, we briefly introduce the three storage technologies investigated in this thesis. One example is selected from each of the three families of storage – NFS from NAS, iSCSI from SAN, and Swift from object based cloud storage.

NFS is a file access protocol and requires at least one message exchange per access. On the other hand, iSCSI is a block access protocol. Its behaviour depends on the file system deployed on the drive. For some file systems, such as ext4, it is possible to aggregate multiple file access into fewer iSCSI command exchanges. Swift is similar to NFS in the regard of file access but is based on the concept of objects and supports only full object access.

We do not intend to provide a complete coverage of the storage technologies but to mention only parts which are relevant to this thesis. For detailed information, please refer to their respective specifications [10, 72] and documentation [53].

Finally, we present a survey on related work at the end of this chapter.

2.1 Network File System

Network File System (NFS) [56] is a file access protocol based on Remote Procedure Call (RPC) [5, 72], originally developed at SUN Microsystems [60]. Modern POSIX compatible operating systems are often bundled with an implementation of NFS.

NFS provides access to parts of local file systems of the server machine to client machines. These shared parts of remote file systems are called "exports" and are mounted into local file system hierarchy of the client machine. A mounted export can then be accessed through ordinary system calls for file I/O. Random access within a file is supported and each file operation may require several NFS operations to complete.

One thing to note is that the client does not do any organisation work which an ordinary file system does. Rather, this is the responsibility of the local file system on the server machine. An overview of NFS storage system architecture is illustrated in Figure 2.1.

During the years, there are numerous revisions to the protocol – version 2, 3, and 4. Previous studies mostly focused on version 2 and 3 [19, 58]. We note that these studies were conducted ten years ago, when version 4 implementations were still immature and were not widely deployed. In this thesis, we investigate the latest version, version 4 ("NFSv4" hereafter).

Compared to previous versions there are numerous changes and new features

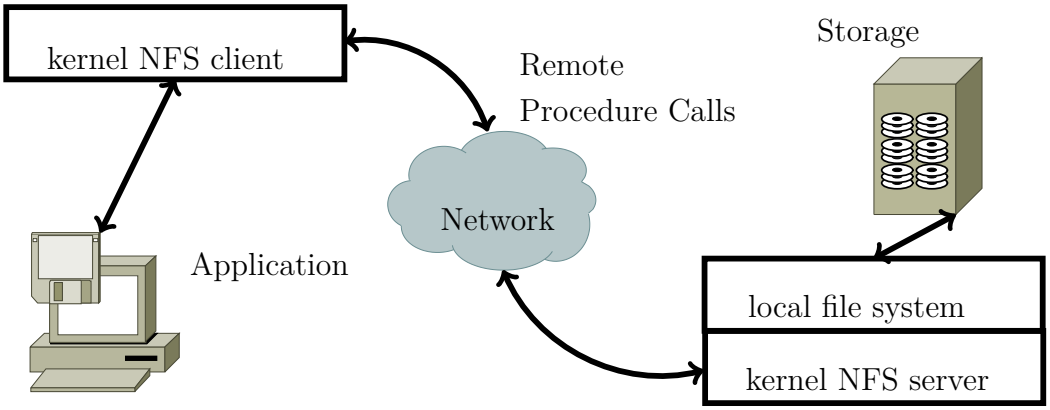


Figure 2.1: NFS architecture.

in NFSv4. NFSv4 mandates the use of TCP as the transport protocol to provide congestion control and to adapt to network unreliability. It is stateful and clients are required to perform OPEN operation before and CLOSE operation after accessing a file to provide enhanced share semantics.

A new COMPOUND operation is introduced which allows coalescing multiple RPCs required for a file access operation into a single one. This reduced the number of round trips required for a file operation and thus the access latency. The concept of open delegation and leases alleviates the coordination burden of both clients and the server by temporarily transferring control of a certain file to a client. Numerous security improvements were also made, such as the use of textual user identifiers instead of integers, and mandatory support of GSS-API [18].

2.2 Internet Small Computer System Interface

Small Computer System Interface (SCSI) is a family of protocols designed for communication between host computer and peripheral devices. It is widely used by storage devices, such as hard disks, for block level access and is practically the common language to communicate with such devices. Internet Small Computer System Interface (iSCSI) [10, 45] is a protocol which allows the encapsulation of SCSI commands into TCP/IP packets for exchange over the Internet.

iSCSI follows the client-server model. The server is called the “target” and the client is called the “initiator”. The initiator sends SCSI commands to request service from a device attached to the target. This device can be physical or logical, such as a part of a hard disk, and is identified by its logical unit number (LUN). iSCSI utilises the concept of sessions to manage communications between client and initiator. Figure 2.2 illustrates the architecture of an iSCSI based storage system.

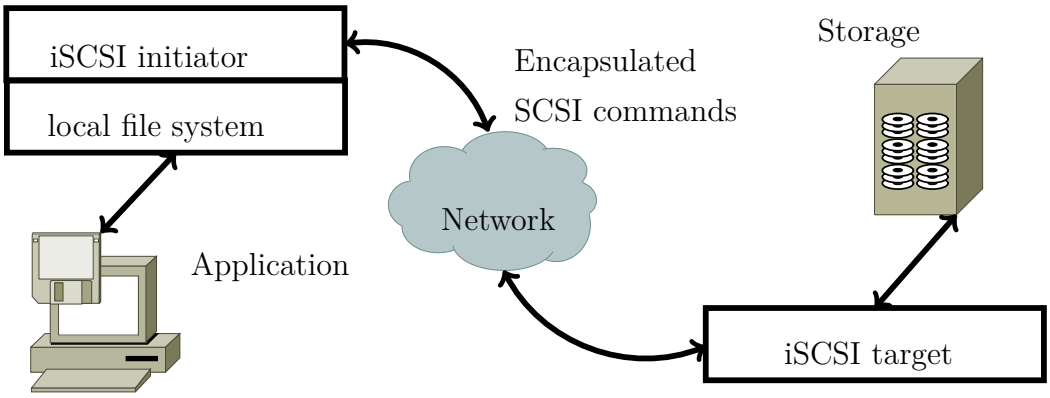


Figure 2.2: iSCSI based storage architecture.

A wide range of scenarios have been considered during the design of iSCSI, including remote access, lossy links, slow storage targets, such as tape. Acceleration hardware for iSCSI is also commercially available. Control and data traffic share the same communication channel for the ease of hardware implementations. It is possible by design to use multiple network channels, such as concurrent TCP connections, with a session. Nevertheless, such feature is usually available only in commercial implementations bundled with enterprise class storage appliances.

2.3 OpenStack Swift

OpenStack [52] is an open source cloud computing platform. It consists of various components, including the object based storage service – Swift [53]. Swift is one of components of the initial release of OpenStack and is of highest degree of maturity.

Swift aimed to provide an open source alternative to Amazon S3. It is implemented in Python and runs in the user space. The object storage service is accessed through an HTTP [22] based RESTful [21] API. Swift itself is composed of several components: Proxy Server, Object Server, Container Server, and Account Server.

Proxy Server is the front end of the whole Swift architecture. All communications between Swift and a client are handled by the Proxy Server. It is also responsible for locating other components responsible for each request and route the request accordingly.

Object Server stores objects as binary blobs on local devices. Local path for an object is derived from the hash of the name of the object. Metadata are stored as extended file attributes (xattr) which is not universally available in file systems. Permissible candidate for underlying file system is therefore limited.

Container Server handles the listing of objects. Each object belongs to a specific container. The list is stored as an SQLite database and does not include

the location of the objects. Account Server is similar to the Container Server, except that it handles the listing of containers.

In addition to the aforementioned components, there are also the Replication processes, Updaters, and Auditors. Storage Policies can also be configured to provide differentiated service.

2.3.1 CloudFuse

As mentioned earlier, all experiments in this thesis are conducted via POSIX file API. We therefore need an adaptor to bridge Swift API and Linux virtual file system (VFS) interface. CloudFuse [4] is a file system in user space (FUSE) implementation of Swift API client.

Linux FUSE consists of two components: a kernel module and a user space library. The kernel module interfaces with VFS like other file systems do. FUSE implementations, such as CloudFuse, connects to the kernel module through the assistance of the library. The two components communicate by passing file system operations and responses as messages via a virtual device. FUSE library supports multithreaded FUSE implementations. CloudFuse by default enables this features and is capable of utilising multiple TCP connections.

Objects corresponding to a file are retrieved in their entirety when the file is opened and then stored in a local temporary file. All subsequent operations are conducted on this temporary file. Finally, when the file is closed, the content of the local temporary file is uploaded to Swift to update the objects.

There are two exceptions. The first is when the file is truncated or opened solely for writing, then the existing objects are not retrieved. The second is when the file is opened only for reading, then the local temporary file is not upload when the file is closed.

The storage architecture of Swift/CloudFuse combination is illustrated in Figure 2.3.

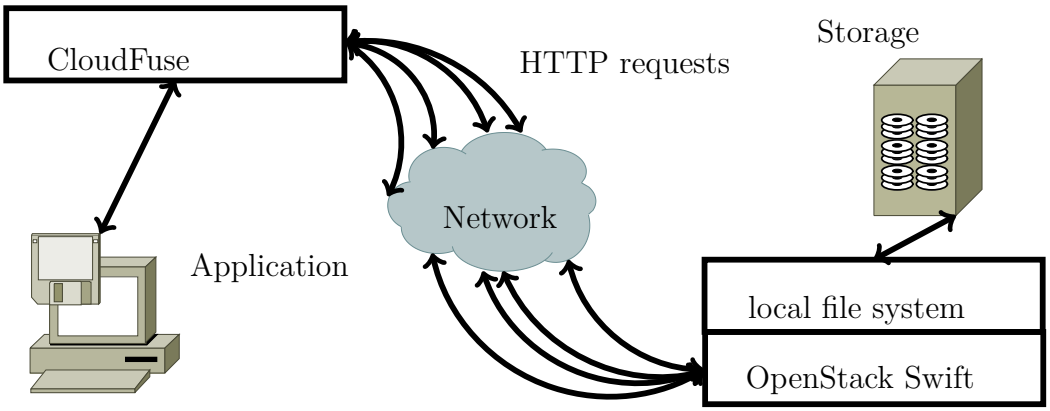


Figure 2.3: Swift/CloudFuse architecture. CloudFuse is capable of utilising multiple TCP connections.

2.4 Related work

A comprehensive comparison of the performance of NFS and iSCSI is provided by Radkov et al. [58] They focused on metadata operations via POSIX system calls. In addition to metadata operations, they conducted single threaded file I/O experiments using an 128 MiB file. This is perhaps more similar to the access pattern of a database management system than to an ordinary user. Drago et al. conducted an in depth study of one commercial cloud storage service [15] and compared several of them [16]. Some of our experiments were inspired by their results and some of our findings agree with theirs. Gauger et al. [23] derived a mathematical model for iSCSI throughput in different network environments and they focused on low latency networks.

Numerous studies aimed to improve storage performance under limited connectivity. Roselli et al. [59] found that small files are common in daily usage environments and reads are more common than writes. Large files tend to be accessed randomly than sequentially. They also found that most blocks have a short lifetime, suggesting that using a large and non-volatile buffer and longer write-back delay may greatly improve storage performance.

Muthitacharoen [47] proposed an approach to reduce the traffic of NFS by employing a local cache and transmits only parts of a file which the other party does not have already. Kroeger et al. [37] proposed an approach to intelligently prefetch files from remote storage server to increase performance. Nightingale et al. [49] suggest that by relaxing synchronous semantics from application's perspective to that of users, it is possible to achieve both the durability and ordering guarantees of a synchronous file system and the performance of an asynchronous one. One effect is that small write requests which are not read immediately after being written can be bundled.

Hardware acceleration for network tasks has been studied and become commercially available. Modern operating systems by default leverage the TCP offload engines [14] on network interface. Moreover, there are a variety of dedicated iSCSI host bus adaptors which implement full iSCSI and TCP stack available in the market. However, as Sarkar et al. [61] indicated, the capability of acceleration hardware must be at least on the same level as the host to have positive effect.

Various approaches exist for storage in distributed systems or over wide area network. Hildebrand et al. [29, 30] proposed an extension to NFSv4 which employs a parallel file system architecture to achieve high performance for large file access. Small file and metadata access still go through legacy NFSv4 I/O path. Google File System [25] is the storage system used by the network giant Google. Wang et al. [74] proposed a distributed file system which supports random inserts and random truncates with a file.

Coda [62], the successor of Andrew [31, 46], is a network file system which supports offline operations under the assumption that there are few shared files between users and relaxed consistency requirements. In addition to Coda,

GPFS [63], Lustre [51], and Ceph [75] are distributed file systems bundled in modern Linux kernel source tree. Henschel et al. [28] deployed and tested Luster over an 100 Gbps trans-continental wide area network. They found that applications are able to run over such deployments. However, the performance does not scale with the increase in bandwidth due to complications in the network core.

Cloud storage and local storage can supplement each other. Livenson et al. [38] implemented a prototype of a proxy which provides a unified interface for accessing heterogeneous storage services, including local file systems and cloud storages. Zhang et al. [79] proposed integrating local file system with cloud storage service to improve resilience against data corruption and inconsistency. FUSE implementations for other cloud storage exist, such as dropfuse[1] for Dropbox and S3QL[2] for Amazon S3, Google Drive, and OpenStack Swift.

Cabrera et al. [7, 8] proposed an architecture which stripes files over multiple storage nodes, which is also named Swift. We conjecture that OpenStack Swift is probably inspired in part by their approach.

A survey on storage studies which involved benchmarking is provided by Traeger et al. [73]. We categorised the experiments conducted in this thesis according to their definitions. Tarasov et al. [68] tried to spur debates on how file systems should be evaluated by showing that simply running the benchmark which others ran not necessarily produces meaningful results. Chen et al. [12] proposed an approach to adapt workloads of a benchmark according to the capabilities and the capacity of the platform under test. Moreover, they proposed an approach to use obtained measurements to predict the performance of untested workloads. Shivam et al. [66] proposed an approach to automate benchmarking in order to explore the test space with less effort.

[1] Available at `https://github.com/arekzb/dropfuse`
[2] Available at `https://bitbucket.org/nikratio/s3ql`

Chapter 3

Environment

The storage systems investigated in this thesis are deployed in a custom testbed. In this chapter, we describe the configuration of our testbed. Figure 3.1 provides an overview of it.

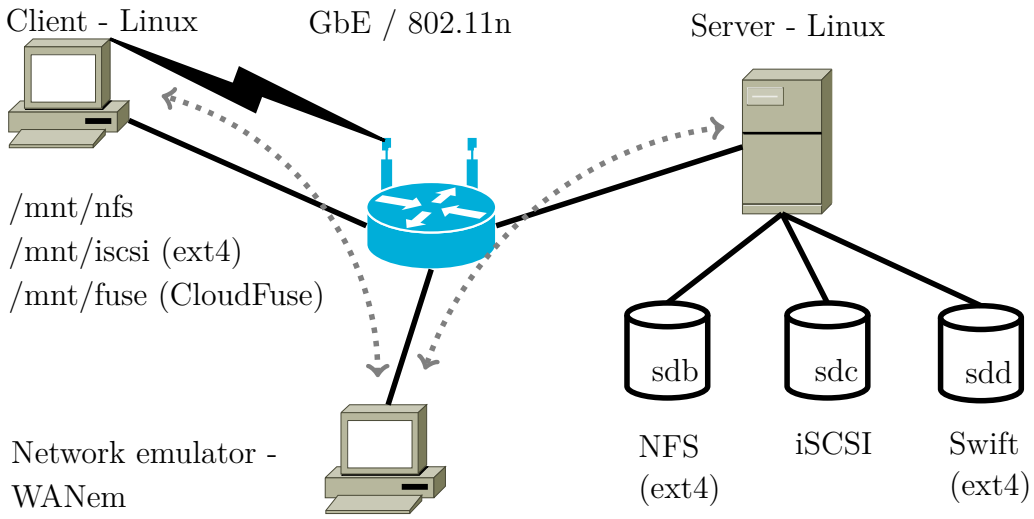


Figure 3.1: Testbed overview. Wireless network is unused in this thesis.

3.1 Hardware

The testbed consists of four components: the client machine, the server machine, the network emulator, and the network fabrics.

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 Intel 82579LM Gigabit Ethernet interface, a Samsung MZ-7PD128 Solid State Drive as system disk and log storage. It is also equipped with an Intel Centrino Advanced-N 6205 wireless interface but unused in the experiments.

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 Intel 82576 Gigabit Ethernet interface, four Western Digital WD2502ABYS 250GB SATA hard drives. One of the hard disks serves as the system disk and one for each of the storage systems we investigated.

The network emulator is a Dell laptop with an Intel Core2 Duo P8400 dual core processor clocked at 2.26 GHz, 4 GiB of memory, on-board Intel 82567LM Gigabit Ethernet interface.

The network switch is an ASUS RT-N66U home router with a 600 MHz Broadcom BCM5300 chip and 250 MiB of memory. All Ethernet interfaces on the switch and cables used are capable of operating 1 Gbps bandwidth.

3.2 Software

Both client and server machine run Fedora[1] 17 with a patched Linux kernel version 3.2.1 [44]. The default TCP congestion avoidance algorithm is CUBIC [26]. The WAN emulator software is WANem[2] 3.0 Beta 2 and is run using live image from a USB flash memory stick. It is capable of emulating various network complexities, including network latency (delay), packet loss, and jitter [24, 35], which are the parameters used in our experiments. The network switch runs Tomato[3] 1.28. Inherent latency of such configuration is less than 1 ms.

To allow high bandwidth experiments, we increased the initial size of TCP buffer to 4 MiB and the maximum size to 16 MiB on both the server and the client machine. In addition, we disabled TCP connection metrics caching in Linux kernel route cache[4]. The modifications to system configuration are shown in Listing 3.2. On the client machine, we disabled all TCP offload engines [14] on the Ethernet interface in order to accurately capture packets.

```
net.netfilter.nf_conntrack_max = 262144
net.core.wmem_max = 16777216
net.core.rmem_max = 16777216
net.ipv4.tcp_rmem = 1048576 4194304 16777216
net.ipv4.tcp_wmem = 1048576 4194304 16777216
net.ipv4.tcp_no_metrics_save = 1
net.core.netdev_max_backlog = 10000
```

Figure 3.2: /etc/sysctl.conf

The NFS implementation in Linux kernel was used for NFS experiments. We increased the number of `nfsd` threads on the server machine to 32. The NFS mount is a directory in an ext4 partition and was exported with `sync` and `no_subtree_check` flags set.

We used the iSCSI implementation by Intel [44] on both the client (the initiator) and the server (the target) with only basic iSCSI features utilised[5]. The iSCSI target software is configured to emulate 1024 bytes of sector size on top of 512 bytes hard disk sector size. Maximum data segment lengths are 64 KB and burst lengths are 64 MB. We formatted an ext4 partition from the client on the iSCSI disk. The partition is mounted with default options.

[1] Official website: `http://fedoraproject.org/`
[2] Official website: `http://wanem.sourceforge.net/`
[3] Official website: `http://www.polarcloud.com/tomato`
[4] Set `tcp_no_metrics_save` kernel parameter to 0.
[5] Available at `http://sourceforge.net/projects/intel-iscsi/`

The default block I/O scheduler of Fedora 17 is Completely Fair Queuing (CFQ). The other scheduler options are NOOP and deadline [2]. We note that CFQ is highly configurable. It is even possible to mimic the operation of deadline. In this thesis we left the parameters in default values.

For Swift we used the latest version on GitHub[6] with Python 2.7.3. We configured it as in the official "Swift All In One"[7] configuration. For the simplicity[8], we configured only one replica and one instance of each component, and used ext4 as the file system for underlying partition. We used CloudFuse [4] on the client machine to allow access to Swift API via POSIX file API.

3.3 Storage targets

There are totally four possible storage targets in our testbed. From the perspective of an application, a storage target is equivalent to a directory backed by a file system "mount".

There is one storage target for NFS and one for Swift/CloudFuse. There are two possible storage targets for iSCSI depending on the block I/O scheduler used on the client machine. In this thesis, we examine CFQ scheduler and deadline scheduler which are defaults for popular Linux distributions[9].

A reference storage target is a directory in the local file system of one of the disks on the server machine. This target serves a reference of a direct attached storage (DAS). All experiments related to this storage target were conducted on the server machine (see Appendix B).

The baseline throughput of the hard disks, measured by conducting sequential read and write using `dd` system tool with `direct` flag set[10], is 110 MiB/s for read and 100 MiB/s for write.

3.4 Network scenarios

Various parameters of the network emulator were varied to explore the impact of network condition. The parameters investigated are: latency[11], packet loss, and jitter.

We chose 100 Mbps (12.5 MiBps) as the bandwidth. Such access links are available to home users via digital subscriber lines (DSL) and IEEE 802.11n [33] wireless network, or through an LTE [70] subscriptions nowadays.

[6] Available at `http://github.com/openstack/swift`

[7] As described in `http://docs.openstack.org/developer/swift/development_saio.html`

[8] For example, to reduce the intensity of disk access, latency, and disk space consumption.

[9] For example, the default block I/O scheduler for Ubuntu (`http://www.ubuntu.com/`) is deadline.

[10] This causes `O_DIRECT` to be passed to `open()` internally.

[11] WANem allows configuring $RTT/2$ as its `delay` parameter. Values mentioned in this thesis are thus two times the values passed to WANem.

Latency values were chosen based on round trip times (RTT) to Amazon data centres, as depicted in Figure 3.4. For WAN scenarios, we chose 50 ms, 160 ms, and 250 ms. In particular, 50 ms is the RTT we measured from Finland to the data centre of Amazon in Ireland. Thus, it can be used to imitate the latency under regular Internet environment. Additionally, we chose 0 ms and 20 ms to emulate LAN and cross country network, respectively.

It has been noted that variations in latency are common in the Internet [24, 35]. This variation is called jitter. Based on our measurements, we chose ± 10 ms[12] as the jitter value when applicable.

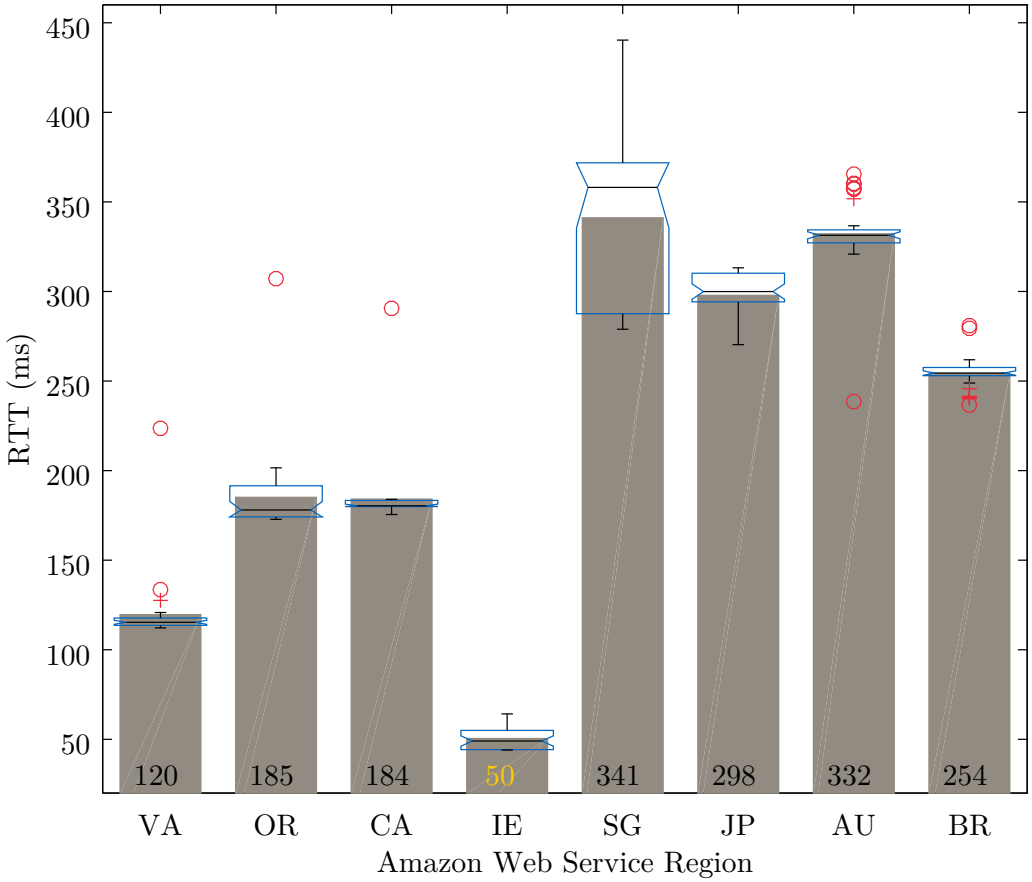
Packet loss rate was varied among 0%, 0.1%, 1%, and 2.5%.

Whenever we mention "realistic network" in the rest of this thesis, we mean 100 Mbps bandwidth, 50 ms latency, 0.1% packet loss, and ± 10 ms jitter, as shown in Table 3.1.

Bandwidth	**Latency**	**Packet loss rate**	**jitter**
100 Mbps	50 ms	0.1%	± 10 ms

Table 3.1: Realistic network condition – our emulated Internet.

[12]WANem relies on NetEm facility of Linux kernel (`tc-netem(8)`) to emulate network latency, loss, and jitter. The default distribution is normal distribution.



(a) Round trip time measurements.

Mark	VA	OR	CA	IE	SG	JP	AU	BR
Location	North Virgina	Oregon	North California	Ireland	Singapore	Tokyo	Sydney	São Paulo
Mean RTT	120	185	184	50	341	298	332	254

(b) Data centre locations and mean round trip time (in milliseconds).

Figure 3.4: Round Trip Time to Amazon Web Services data centres measured from test sites in Uusimaa region of Finland.

This page was intentionally left blank.

Chapter 4

Microbenchmarking

We designed a custom benchmark which simulates an application conducting file I/O via POSIX file API and a user giving commands via operating system shell to study the performance of the three storage systems. In this chapter, we introduce the design of the benchmark and discuss experiments related to file I/O. Discussion on experiments related to metadata operations is deferred to Chapter 5.

This chapter is organised as follows. We first introduce the design of the benchmark in Section 4.1, then discuss two important parameters to POSIX file API – form of access (Section 4.2) and access unit size (Section 4.3). Subsequently, the results of our experiments are discussed, in the order of: single file access (Section 4.4), file access by a single threaded application (Section 4.5), and file access by a multithreaded application (Section 4.6). Furthermore, we investigate the impact of Linux block I/O schedulers (Section 4.6.3), amplification caused by ext4 file system (Section 4.7), and the effect of increasing network bandwidth to 1 Gbps (Section 4.8). In addition, a short summary of our findings is provided in Section 4.9.

The metric used in this thesis is the transmission throughput observed by an application. Previous study showed that the energy consumption of the wireless communication component constitutes a major portion of that of the whole mobile platform [9]. Others have shown that the energy consumption of communication is highly related to the transmission throughput [50, 55]. Although CPU may become the power hog under heavy load [54], the CPU utilisation remained below 10% in most of our experiments and we therefore focus only on the throughput.

We started the study with an established UNIX system tool – `dd`. It allows specifying the `open()` flags investigated in this thesis, file size, access unit size, and many others. For Swift, we used the official command line tool `swift`. With some scripting, we are able to mimic various sequential and parallel access patterns and to perform parameter sweep test. However, overheads such as process startup and process tear down turned out to be significant. Moreover, we would like to have a POSIX file interface for Swift API to allow the same benchmark to be used. The latter was solved by using CloudFuse.

We noticed that modern benchmarks are often complicated and some are even expensive. Those who are free are often insufficiently documented. Consequently, it is hard to control their exact access pattern for microbenchmarking. Furthermore, at some point we need server side assistance to speed up the benchmark.

4.1 Benchmark design

This led to the construction of a custom benchmark suite which is capable of generating sequential, parallel, and worker pool access patterns. Unlike our preliminary experiments, whilst we typed the commands one after another manually, our benchmark features certain degree of automation. It comprises a program written in C which simulates a pool of worker threads servicing a given set of file read or write operations and several of shell scripts which set up the environment and drive the tests. Due to the large amount of variables we would like to consider the complexity of the benchmark grew quickly. Considerable effort was spent on testing, verification, and debugging. The benchmark saw hundreds of revisions during the course of this study.

States such as current time, network interface counters, and CPU usage are collected from relevant Linux kernel counters through C library routines, `/proc` pseudo file system, or through system utilities such as `date`.

We do not restart the server or storage services, or re-mount file systems, as Radkov et al. [58] did because such a fresh server is highly unlikely in reality. In case that we have a newly connected storage target, we create, read, and then delete, 100 files which are 1 MiB in size to warm it up.

Potential bias exists in the benchmark as well as in the environment. The most prominent sources are caching in Linux kernel and TCP slow start [1, 26, 32, 34]. TCP slow start causes extremely low throughput if the time frame of measurement is too narrow. This problem mainly concerns batch experiments and we defer its discussion to Section 4.5.

Under the influence of the Linux kernel caching, we end up benchmarking the caching facility [40]. Since we would like to infer the energy consumption of wireless network hardware from the measured throughput, we therefore would like to have buffered operations transmitted over the network. This problem is solved by flushing and dropping local kernel cache (Figure 4.1) after running each test round.

```
$ sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"
```

Figure 4.1: Command to drop Linux kernel cache

While we assumed a cold cache on client machine, a warm cache on server machine is assumed. To ensure this, we "reheat" the cache on server machine by reading the same set of files involved in a reading round twice and measure the second run. This does not necessarily guarantee a cache hit on the server machine as the working set may be larger than the cache. The purpose is to simulate reading recently accessed files. During this reheat phase, the network emulator is disabled and the client connects to the server through 1 Gbps switched link.

4.2 Forms of access

When a file is opened via `open()` system call for access, a flag indicating the desired form of access can be passed. The flags of interest are O_SYNC and O_DIRECT, which specify synchronous I/O and direct I/O, respectively. According to Linux Programmer's Manual[1], setting O_DIRECT tells the kernel to try to minimise cache effects of I/O operations on the file being opened. It tries to bypass caching facilities, such as the page cache, along the I/O path as much as possible. Setting O_SYNC causes `write()` to block until relevant data have reached the physical storage. We define three different forms of access: direct, sync, and default. Their corresponding flags are listed in Table 4.1.

Form	direct	sync	default
Flags	O_DIRECT	O_SYNC	none

Table 4.1: Forms of access and `open()` flags applied.

In our environment, bypassing I/O cache entails that data must be read from or written to remote storage upon request. Furthermore, no read ahead or bundling is possible due to the absence of a buffer. The physical storage with regard to our environment is the storage service on the server machine. Whether the data are written to the physical storage on the server machine is another matter and is beyond the scope of this thesis. In this regard, synchronous I/O and direct I/O are similar for write operations. On the other hand, synchronous read enjoys read ahead which is employed by both NFS and ext4 file system to increase read performance.

A comparison of the performance of different storage systems using different forms of access by accessing a large file in units of different sizes is illustrated in Figure 4.2a and 4.2b. From the comparison we see that default read and sync read for NFS are similar – both benefit from read ahead – whereas default write is buffered and is only limited by link capacity. iSCSI behaves similarly except that default write only writes to local buffer on the client machine. Measuring iSCSI default write performance thus degenerates to measuring the performance of in memory cache. Hence, we focus on sync and direct access forms for NFS and iSCSI in the rest of this thesis.

Although Linux FUSE supports O_DIRECT and O_SYNC flags, CloudFuse does not take them into account when it operates on the local temporary file. Our preliminary measurements showed that forms of access are indifferent[2] for Swift/CloudFuse combination. We thus present only results of default access form for Swift/CloudFuse.

The case where O_DIRECT and O_SYNC are set simultaneously is not looked into in this thesis due to increased complexity. However, preliminary

[1] See manual page `open(2)`. Issue `man 2 open` at command line to show it.

[2] They *are* different, even for local disks, see Appendix B. However, our bottleneck turns out to be the network and the Swift service.

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 4.2: Writing and reading policies of different forms of access.

results suggest that read performance under such flag combination is similar to that of direct and write performance is similar to that of sync.

4.3 Access unit size

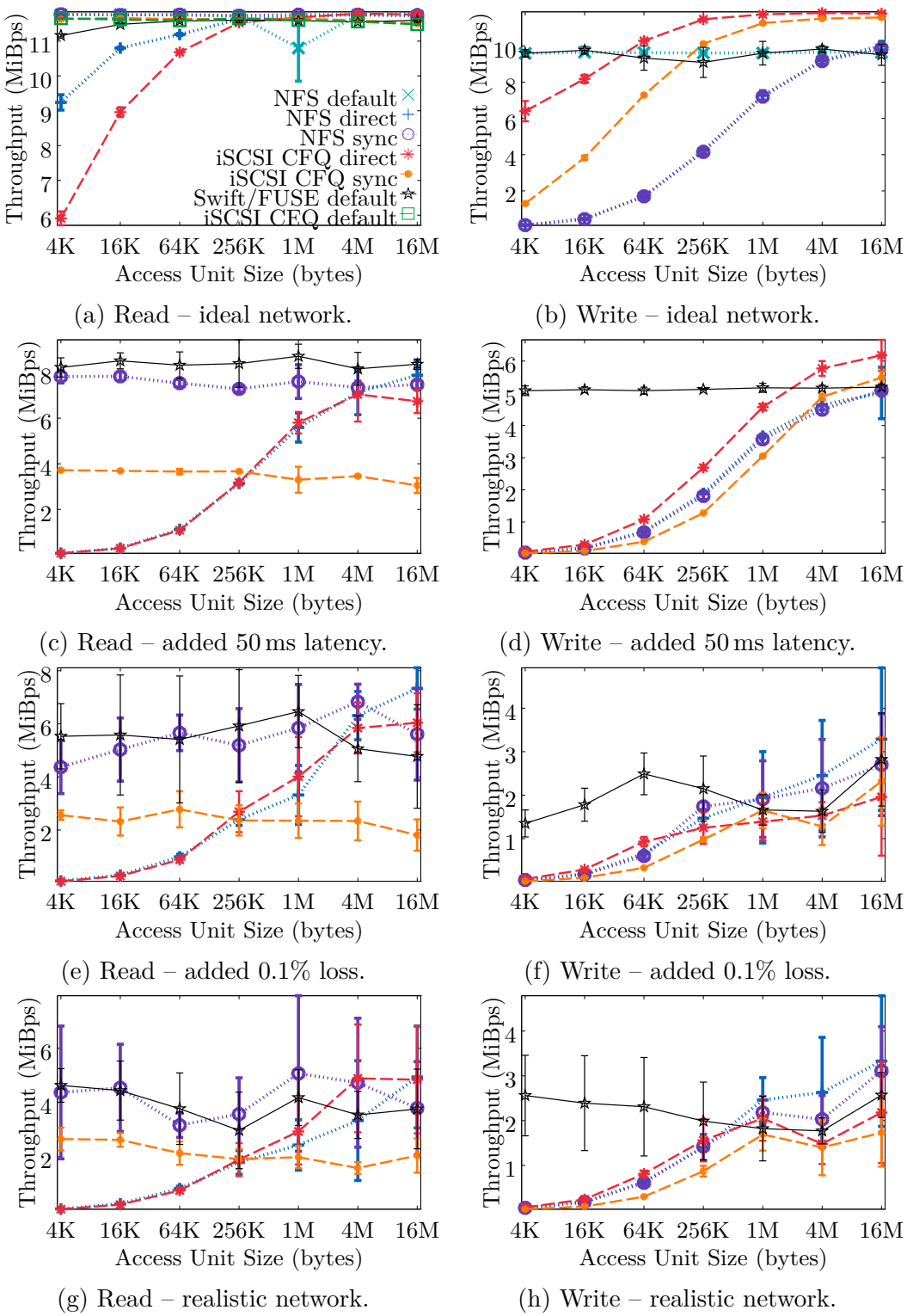
In this section we discuss the impact of different access unit sizes[3] which is an important parameter when invoking POSIX file I/O routines. The smaller the access unit size, the more the I/O requests generated. The experiment is conducted by accessing a file which is 16 MiB in size in units of 4 KiB through 16 MiB.

Before we conduct the write tests, we first write the whole test file once. Similarly, the test file is read in whole once before read tests. The purpose is to ensure a warm cache on the server machine and a warm file system cache on the client machine. Kernel cache on the client machine is dropped before each round of experiment.

A comparison of performance under different access unit sizes is illustrated in Figure 4.2. It is clear from the figure that Swift performance is unaffected by access unit size. This is because that Swift does not support random access within an object and CloudFuse always read the whole file into a temporary file in advance, or upload the whole temporary file after all local file operations.

Synchronised read for both NFS and iSCSI benefit from read ahead, as can be seen from Figure 4.2a. However, it is revealed under 50 ms network latency that iSCSI read commands are issued in a synchronised fashion – next command is issued only after the response for the previous command is received – which caused the constant low performance in Figure 4.2c. Direct I/O read

[3]Corresponds to `bs=`, or *block size*, parameter of `dd` utility. For `read()` and `write()` this corresponds to `count` argument.



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.2: Performance comparison of different access unit sizes under different network conditions. The test file is 16 MiB large. Note the different scales.

performance grows as the access unit size increases. This is because that local cache is bypassed in direct I/O and data are read from the server at requested size.

Write performance exhibits similar trend for both sync and direct access forms, with iSCSI slightly outperforming NFS. Under both access forms, data must be sent to the server before the operation completes. iSCSI direct write is slightly faster than sync write. We conjecture that the difference is due to the absence of local buffering.

From Figure 4.2e and 4.2f we see that packet loss has a significant impact on the performance due to the time required by TCP to recover from loss. This is especially prominent when the lost packet is the initiating command of some operation. Adding 10 ms of jitter to the network, which results in our emulated Internet condition, degrades the performance further, as illustrated in Figure 4.2g and 4.2h.

We note that small access unit may result in extremely poor performance under direct access form or under network conditions like the Internet. It is therefore suggested that the access unit be as large as possible. Hence, all experiments in the rest of this chapter are conducted with access unit size equal to the file size.

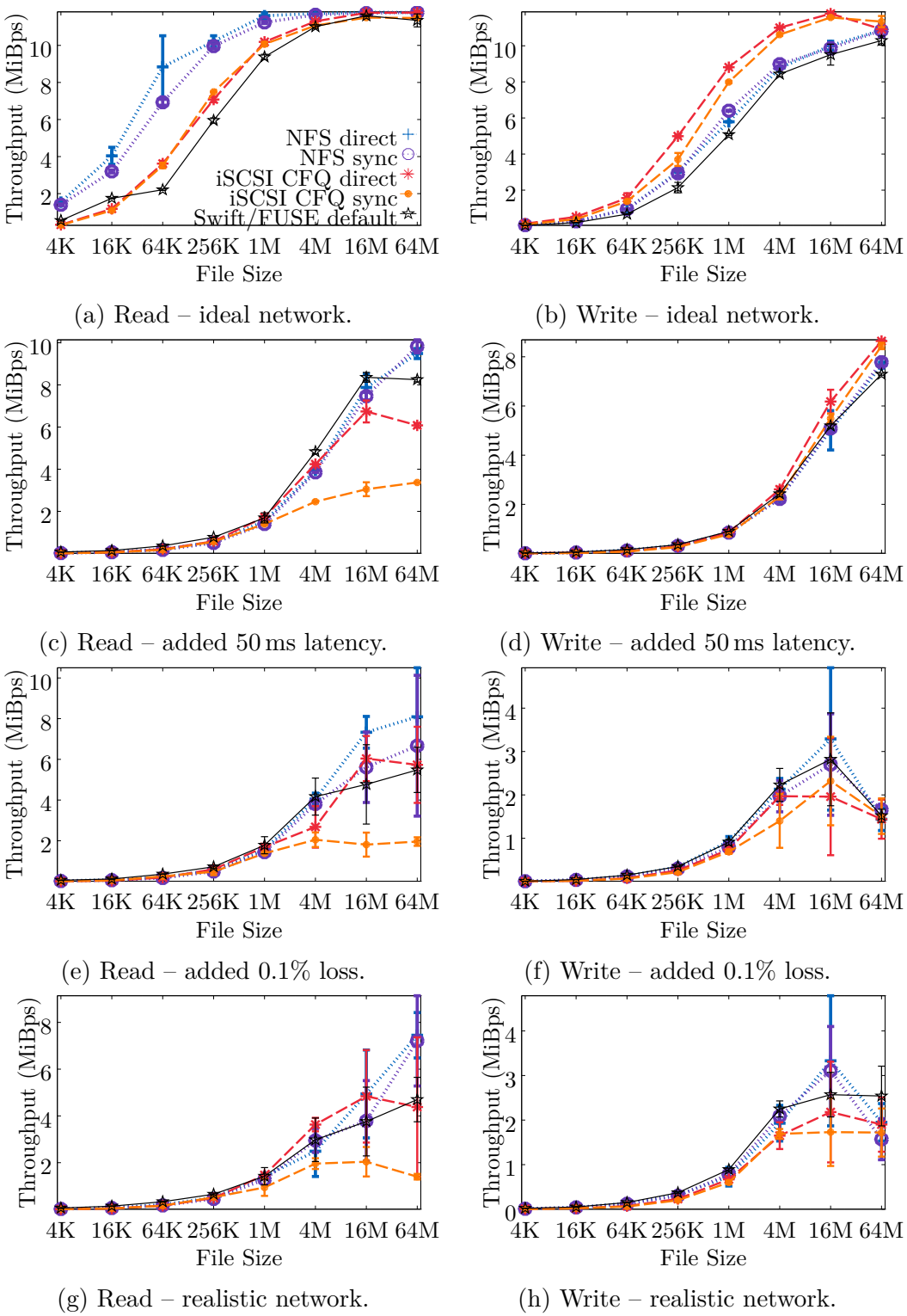
4.4 Single file access

In the preceding section we discussed the impact of access unit size on the performance of reading and writing a large file. It is noted, however, that small files are at least as common as large files in real life [15, 25, 59] and their access could impact energy efficiency significantly [32]. In this section we discuss the performance and other aspects of accessing one single file of different sizes. The file size we tested range from 4 KiB to 16 MiB.

In Figure 4.3c we observe the same behaviour of iSCSI synchronised reads as mentioned in Section 4.3 for files larger than 1 MiB which require multiple SCSI command cycles to read. Its performance is significantly impacted by network latency since it waits for the response of the previous requests before sending the next request.

From Figure 4.3e and 4.3f we see that loss impacted the performance of reading and writing files larger than 4 MiB significantly due to the increased number of message exchange rounds.

The general trend observed in Figure 4.3 is that the larger the file, the higher the throughput. Small files suffered because their transmission finished while still in TCP slow start phase [15].



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.3: Single file access under different network conditions. Note the different scales.

4.5 Batch operation

In this section we discuss batch operations where a set of files of the same size is accessed sequentially. The file sizes we chose are: 4 KiB, 1 MiB, and 16 MiB. This selection is based on observations made by previous studies which suggest that files of these sizes are most common in practice [15, 25, 59] or their transmission affects energy efficiency significantly [32].

As we mentioned earlier, if the file set is too small the resulting performance would suffer from TCP slow start. We are interested in steady state performance, or the highest sustainable throughput, of the whole system. To address this problem we need to make the benchmark run long enough, which means larger file set.

Preliminary experiments showed that reasonable throughput are obtained after running the benchmark for approximately 20 seconds. Since the highest sustainable throughput cannot be higher than the baseline throughput[4] of the underlying hard disk, we have

$$110\,\text{MiB/s} * 20\,\text{s} = 2200\,\text{MiB} \approx 2048\,\text{MiB} = 2\,\text{GiB}$$

and thus select 2 GiB as the total size of file sets.

This increase in file set size, however, raised two new problems. The first is that under certain network conditions some test rounds took excessively long to run. Consequently, we introduced a new parameter which limits the duration for which a test round may run. This parameter is set to 90 seconds for all batch experiments, including the multithreaded experiments discussed in the following section.

The second problem is that if a large number of small files are placed in a directory its relevant metadata grows enormously in size. Performance of accessing 4 KiB files thus suffer from amplification (see Section 4.7). Consequently, we introduced another arrangement in which we distribute test files into subdirectories each containing no more than 4096 files. In practice, only the 4 KiB file set which consists of 524288 files spanned multiple subdirectories.

We found from preliminary experiments that throughput of reading and that of writing are highly asymmetric, as can be seen from Figure 4.4. To address this asymmetry, we prepared file sets of different sizes on each storage system for reading experiments, instead of reading the files created in writing experiments.

It is also noted that there are occasionally failed operations, especially for Swift. The exact reason is beyond the scope of this thesis. We exclude failed operations from throughput calculation since they seldom happen and the portion of failed operations is generally less than 1%. That is, we compute throughput by dividing the total size of successfully transferred files by elapsed time.

Original CloudFuse backs off on most Swift errors (HTTP 4XX responses). However, we found that HTTP 404 responses are final and backing off would

[4]Refer to Appendix B for more details.

only stall the benchmark. We therefore modified CloudFuse to disable backing off and retrying on receiving HTTP 404 responses and report failure immediately[5].

For NFS and iSCSI, creating a new file involve different steps than overwriting an existing file. We delete all files created by previous writing test before running another writing test. However, deleting large amount of files from Swift is time consuming (see Chapter 5) and we find no noticeable difference between creating a new file and overwriting an existing file. Thus, we do not delete existing files for Swift and simply overwrite them instead.

The performance of of batch, single threaded access of file sets of different file sizes is illustrated in Figure 4.4.

From Figure 4.4a and 4.4b we see that iSCSI excels under ideal network condition and NFS performs slightly better than Swift. One interesting observation is that iSCSI direct write greatly outperforms sync write in Figure 4.4b. We conjecture that this is due to the fact that direct I/O bypasses cache facility and that there exist inherent latency in our testbed[6].

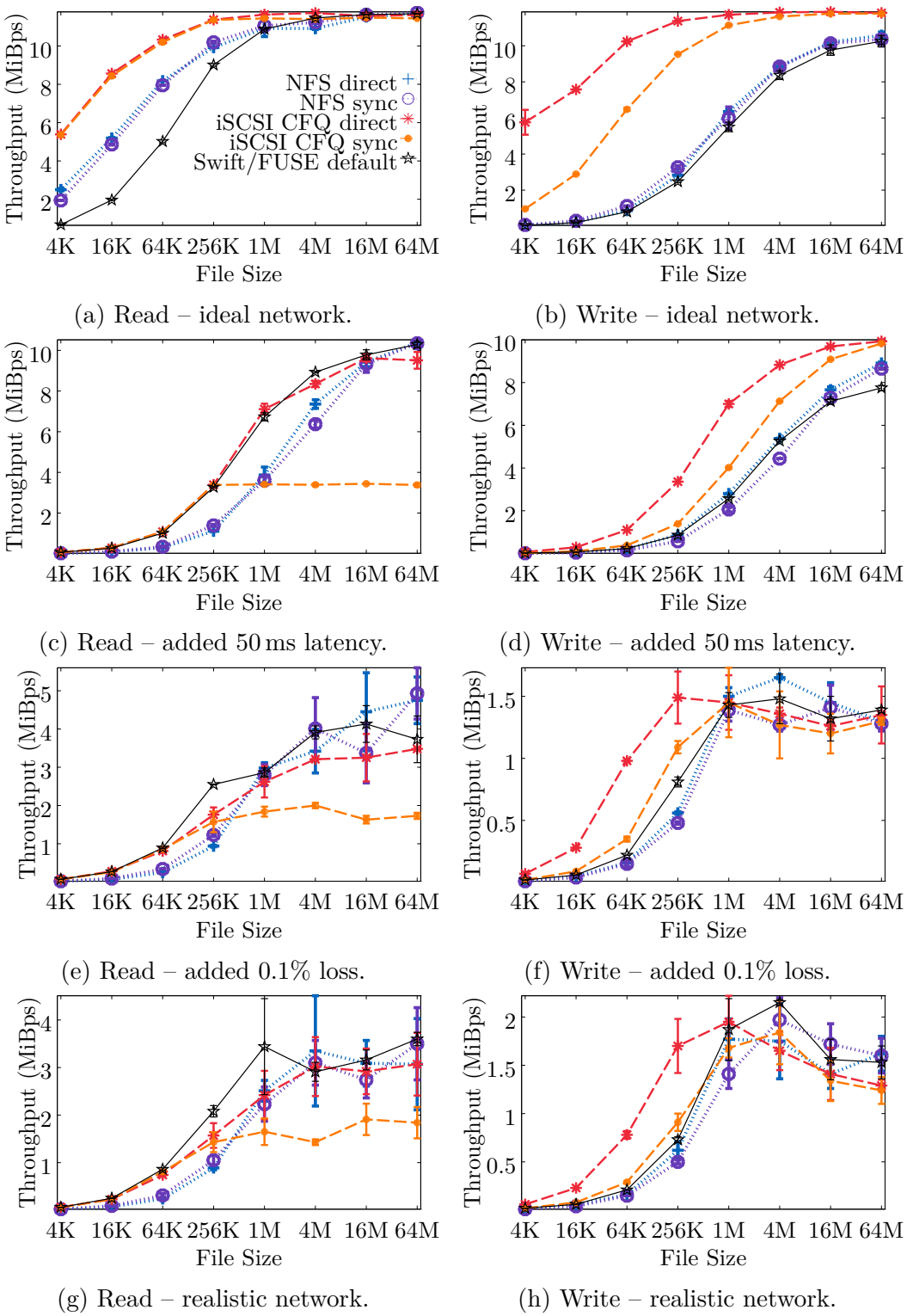
We see from Figure 4.4c and 4.4d that iSCSI and NFS are severely impacted by network latency. The peculiar levelling of synchronised iSCSI read performance is due to the way it works, as discussed previously in Section 4.3. Adding packet loss and jitter to the network further degrades the performance of all three systems and increases the variation.

For file set of 16 MiB files all three systems perform on the same level.

As a short summary, for single threaded application, iSCSI delivers the best performance for small file access and all three systems perform on the same level for large files.

[5]This modified version is available at `https://github.com/zwuh/cloudfuse/tree/timeout`

[6]For example, caused by the Ethernet switch and network emulator.



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.4: Performance of single threaded batch access. Note the different scales.

4.6 Multithreaded application

In this section we investigate multithreaded batch operations. File sets as described in Section 4.5 are accessed sequentially but, different from the previous section, file sets are accessed by multiple worker threads. This is to simulate applications which employ multiple threads or child processes to service queued I/O requests from the user. The number of worker threads range from 1 to 64, in steps of powers of 2. We divide the discussion into subsections for small and large files.

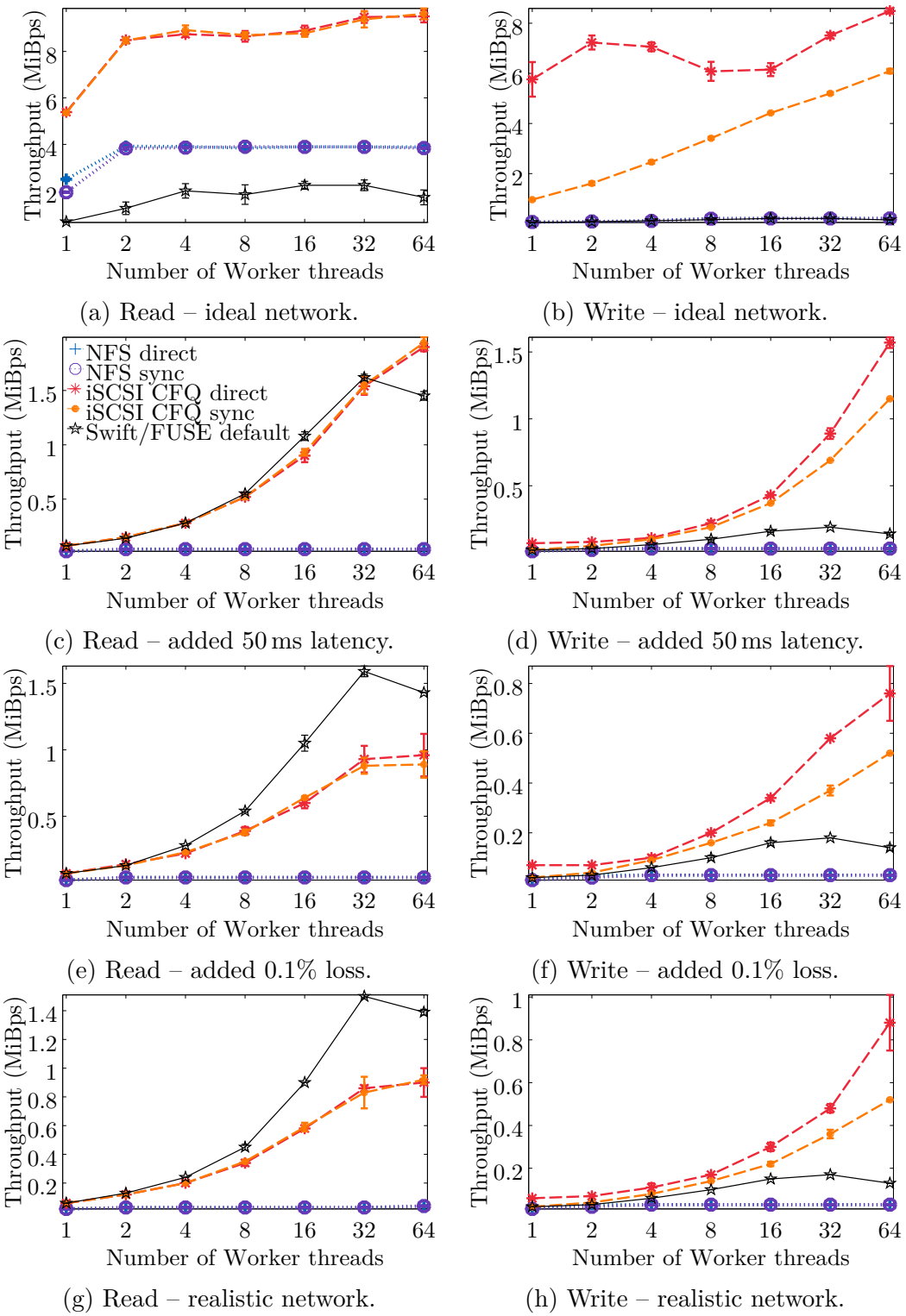
Generally, Swift is the best for read and iSCSI is the best for write due to the aggregation of ext4 file system. For large files, Swift delivers outstanding performance due to its capability of utilising concurrent TCP connections.

4.6.1 Small files

Performance of three storage systems accessing 4 KiB files under different network conditions is illustrated in Figure 4.5.

A trend similar to that in Figure 4.4 is observed. Under ideal network, iSCSI reads much faster than NFS which is twice as fast as Swift. Performance of NFS and Swift are limited by their protocol overhead. Latency impacts small file performance significantly. With 50 ms latency introduced, NFS performance degrades severely and iSCSI is on the same level as Swift. Loss impacts their performance further. Swift outperforms iSCSI after further adding loss. Adding jitter on top of latency and loss affects only slightly. For write operations, iSCSI excels under all network conditions and Swift delivers slightly better performance than NFS.

From the figure we see that Swift and iSCSI throughput increases as the number of threads increases. Swift performance grows up to 32 threads and decreases slightly at 64 threads. The reason is likely due to coordination overhead of multi tasking. iSCSI performance increases because access requests from different threads are merged to form larger SCSI access. We also see that iSCSI performance is strongly affected by network latency as it exhibits the largest performance drop.



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.5: Performance of multithreaded batch access of 4 KiB files. Note the different scales.

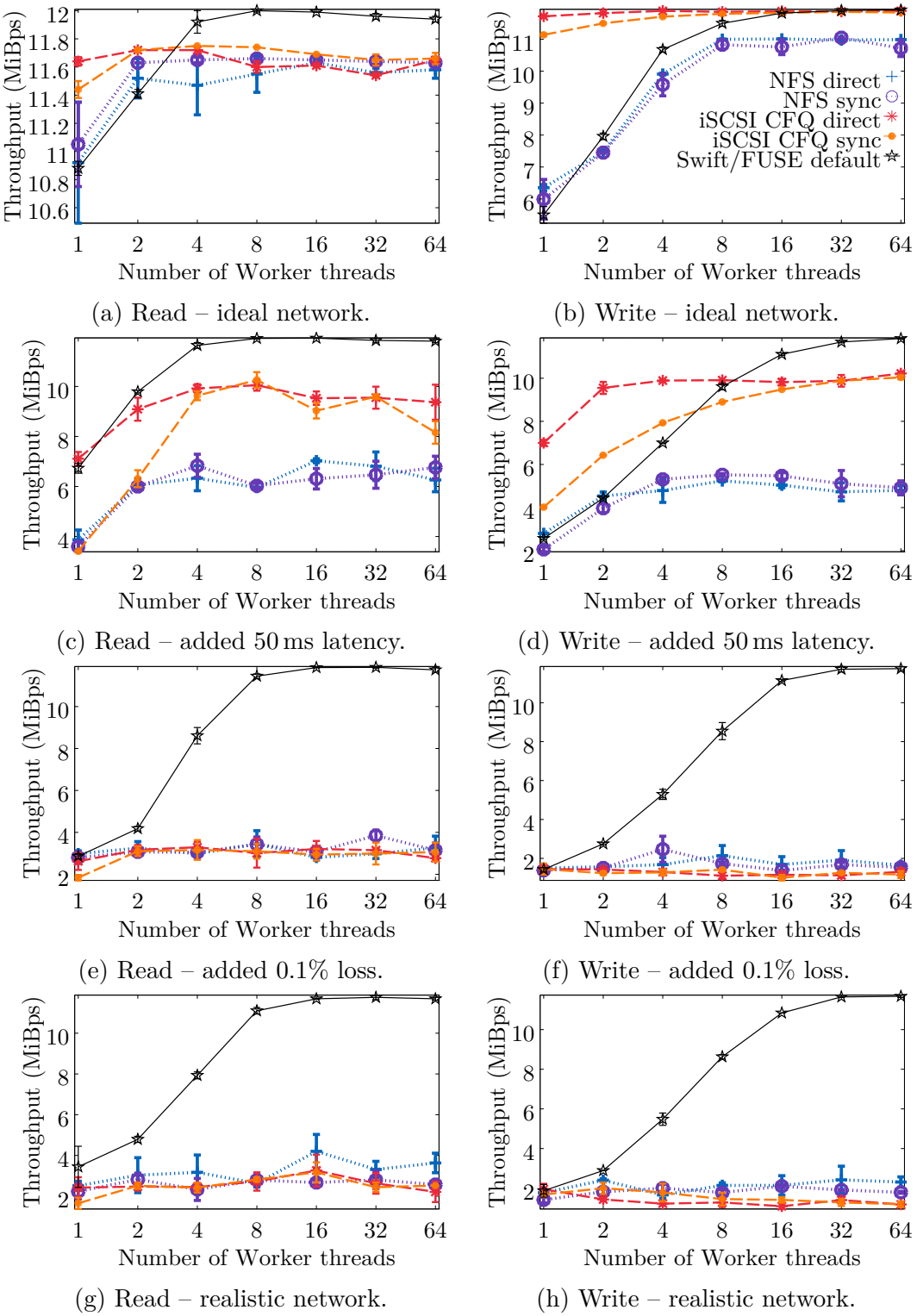
4.6.2 Large files

Performance of three storage systems accessing files of size 1 MiB and of size 16 MiB are illustrated in Figure 4.6 and 4.7, respectively.

From the figures we see that under ideal network condition Swift reads faster than NFS which is slightly faster than iSCSI when the file is very large (16 MiB, for example). iSCSI read performance is likely limited by the largest access unit size per SCSI command which is 1 MiB in our environment, whereas Swift transmits the file in its entirety. On the other hand, Swift and NFS are only comparable to iSCSI on write operation when the number of threads or the size of the files is large. One possible reason is the higher processing overhead on the server machine.

Adding 50 ms latency impacts the performance of iSCSI and NFS significantly. For reading of 1 MiB files, Swift saturates the link, iSCSI is slightly slower and NFS throughput nearly halved. Write operation showed similar trend, though the performance is lower when the number of threads is small. For reading of 16 MiB files, Swift still saturates the link, NFS and iSCSI direct are on par and slightly slower, and iSCSI sync delivers only one third of the others (Figure 4.7c). This peculiarity is due to Linux block I/O schedulers and is discussed in Section 4.6.3. For writing, iSCSI performance remained steady and NFS performance converges to that of iSCSI. Swift performance rises to link capacity as the number of threads grows.

While operated under loss and jitter, Swift delivers outstanding performance, saturating the link capacity, whereas others achieve less than 4 MiB/s. Compared to small files, latency has a smaller impact. The TCP window is able to enlarge and there are more packets in transit. On the contrary, loss has a higher significance. By monitoring the network traffic and examining the implementations, we found that iSCSI and NFS use only one TCP connection, whereas CloudFuse, our VFS adaptor to Swift, is capable of using multiple TCP connections. If one of the TCP connections backed off after loss, others can still flow through the link as normal. Jitter affects performance only slightly.



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

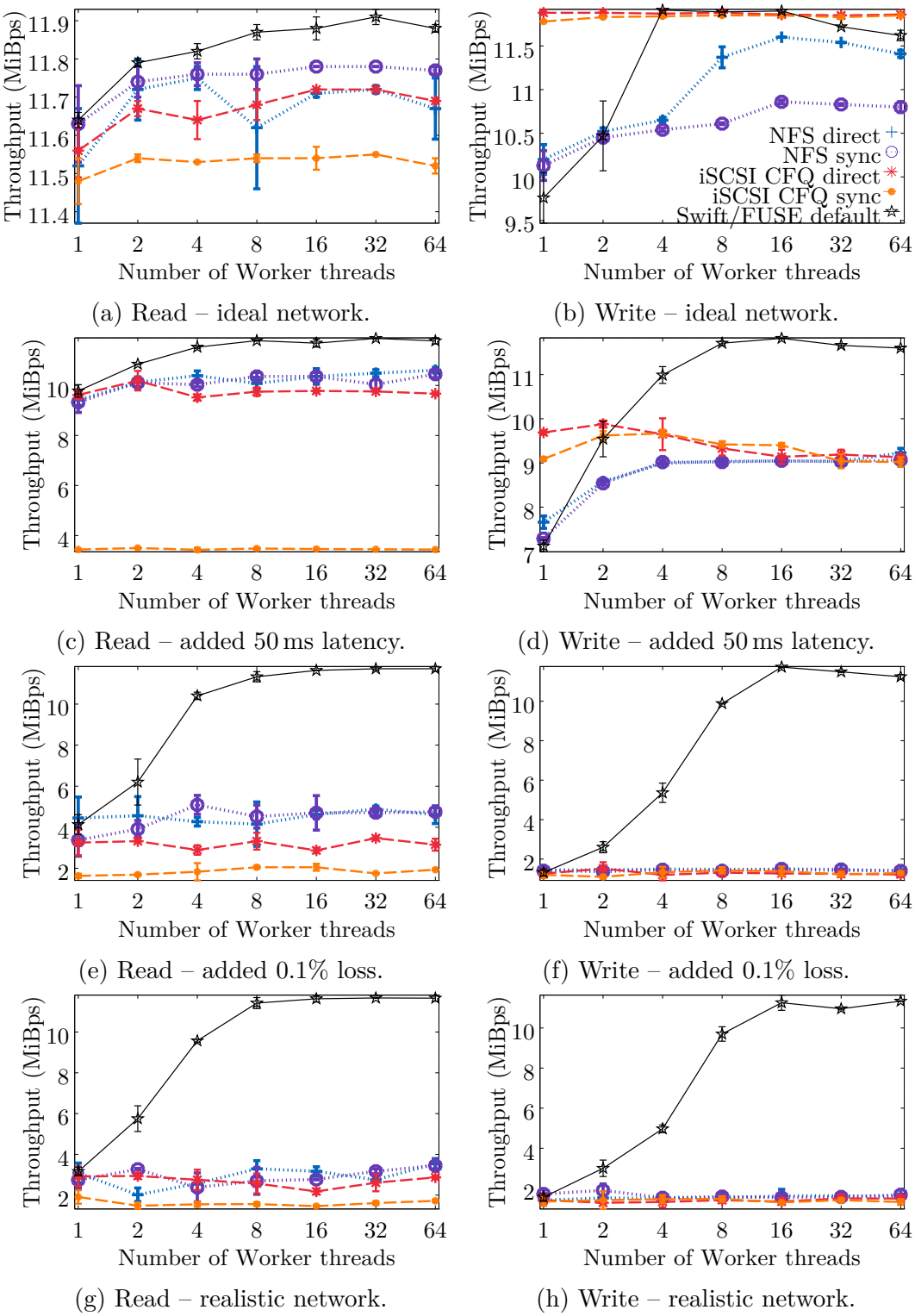
(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.6: Performance of multithreaded batch access of 1 MiB files. Note the different scales.



(a) Read – ideal network.

(b) Write – ideal network.

(c) Read – added 50 ms latency.

(d) Write – added 50 ms latency.

(e) Read – added 0.1% loss.

(f) Write – added 0.1% loss.

(g) Read – realistic network.

(h) Write – realistic network.

Figure 4.7: Performance of multithreaded batch access of 16 MiB files. Note the different scales.

4.6.3 Linux block I/O scheduling

As mentioned in the previous section, an abnormally low throughput for large file reads is observed for iSCSI sync under network latency (Figure 4.7c). We found that Linux block I/O scheduling [2, 39, 57] has an essential influence. In this subsection, we briefly introduce two commonly employed block I/O schedulers of Linux kernel – deadline and CFQ – and briefly compare their effect on iSCSI performance under unideal network conditions.

The deadline scheduler is the default for recent Ubuntu distribution. It sorts and serves block I/O requests in the order of logical block address (LBA). Two other first-in first-out (FIFO) queues are maintained, one for read requests and one for write requests, based on their arrival time. Normally requests are served, or submitted to the block device, in the order of LBA. However, if a request at one of the FIFO queues has waited longer than a certain amount of time (the "deadline"), it switches to serve some requests from that queue before switching back. Thus, the deadline scheduler tries to avoid missing deadlines too much.

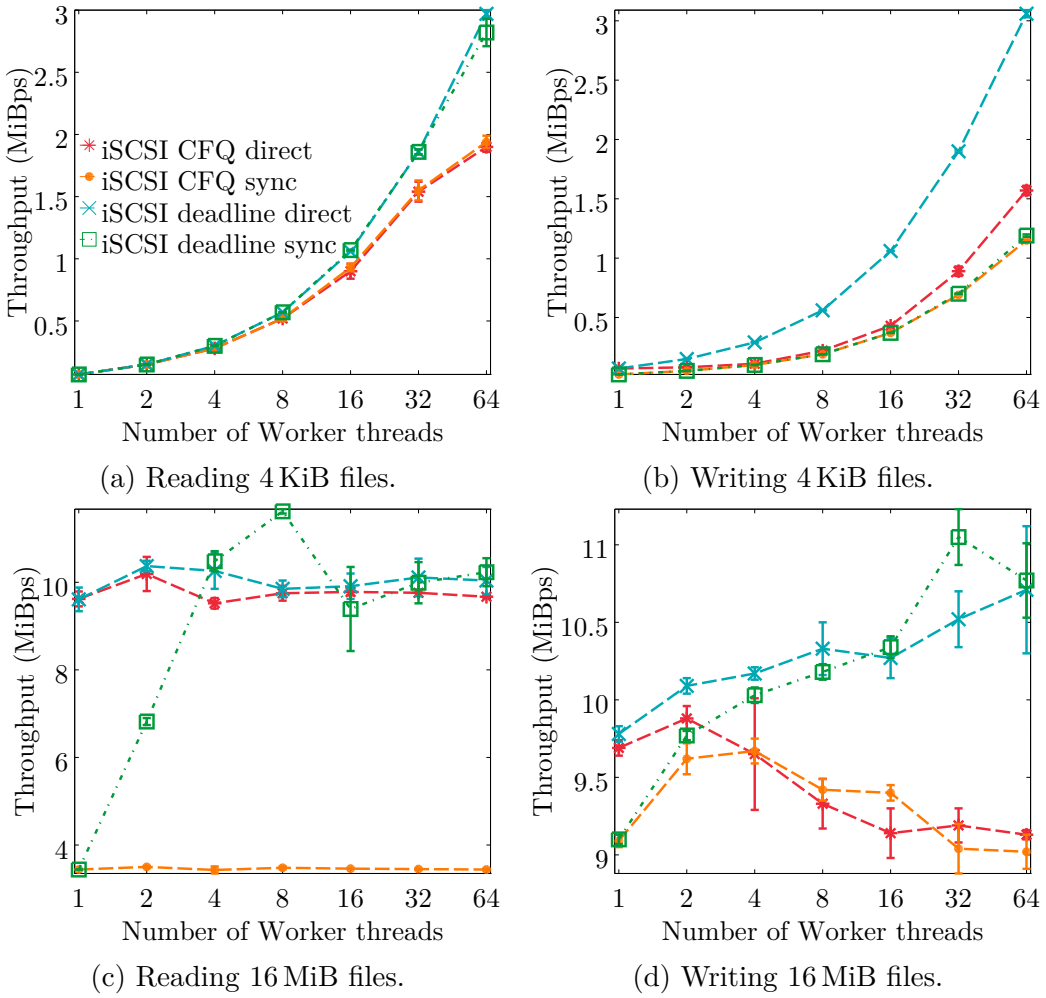
The Completely Fair Queuing (CFQ) is the default for Fedora distribution. It aims to fairly divide I/O bandwidth among processes. In Linux, writes to a block device are carried out by the `pdflush` kernel thread of the corresponding device. Thus, CFQ in practice divides only read bandwidth among processes.

The difference, in terms of throughput, between CFQ and deadline is not so obvious in Figure 4.8a because the file size is only 4 KiB, which requires one round of command exchange to read anyway. However, the difference become significant in other figures.

Threads, such as the ones created by our benchmark using `pthread` library, are light weight processes (LWPs) and each receives its own share of read bandwidth from CFQ. However, under the effect of network latency, a large portion of each time slice was wasted on waiting for the response from iSCSI target, leading to the aforementioned low throughput. We see in Figure 4.8c that deadline scheduler does not have such problem.

We also note that deadline scheduler is able to submit multiple I/O requests, possibly from different processes, in the same batch to the iSCSI target. deadline scheduler thus achieves higher throughput than CFQ in writing, as shown in Figure 4.8b and 4.8d.

We conclude from the aforementioned observations that block I/O scheduler plays an important role in iSCSI based storage systems, particularly for synchronised file operations.



(a) Reading 4 KiB files.

(b) Writing 4 KiB files.

(c) Reading 16 MiB files.

(d) Writing 16 MiB files.

Figure 4.8: Performance comparison of different block I/O schedulers. 50 ms network latency. Note the different scales.

4.7 Amplification

Another interesting phenomenon we observed is the huge volume of transmission incurred by iSCSI when the file is small, sometimes saturating the 100 Mbps link while delivering low throughput. This is the read and write amplification caused by ext4 metadata and journaling [11]. It is rarely reported in literature in the past as file systems, such as ext4, are generally used on direct attached storage (DAS) or SAN over high bandwidth, low latency connections. Write amplification has become an issue only after the emergence of flash memory based solid state storage devices whose lifespan is limited by a comparably small number of erase cycles [41].

The amplification ratio, computed by dividing total transmission in both directions by file size for single file access and by dividing the sum of line rates in both directions by throughput for batch operations, of different storage systems

is illustrated in Figure 4.9.

iSCSI incurs high overhead for single file operations, as is evident in Figure 4.9a and 4.9b. Examination of captured packets showed that NFS overhead consists of protocol messages and is approximately several KiBs in size. Swift requires only a few HTTP requests and thus incur the lowest overhead.

For iSCSI reads, we find that in addition to the file itself, there are always additional $40 + 92$ KiBs read. After reading the file, $8 + 4 + 4 = 16$ KiBs were written back. These are likely the ext4 metadata. For iSCSI writes, we find that there are additionally $4 * 37 = 148$ KiBs read and $28 + 7 * 4 = 56$ KiBs written. These are metadata and journal entries.

We thus have the following estimations for iSCSI amplification ratio:

$$A_{iSCSI}^{read}(s) = \frac{148 + s}{s} \text{ for read}$$

and

$$A_{iSCSI}^{write}(s) = \frac{204 + s}{s} \text{ for write,}$$

where s is the size of the file in KiBs. Note that these are the values observed in our testbed. It is anticipated that different system configurations, file system, and file system options affect the values.

From Figure 4.9c through 4.9f we see that iSCSI overhead become much lower under batch operations. This is because that metadata read from the disk are shared by many files. We also note from Figure 4.9d and 4.9f that iSCSI direct write overhead is very low. We found from the traffic capture that iSCSI direct writes metadata and journal in large batches, around 2 MiB per 10000 files. On the contrary, iSCSI sync writes metadata and journal for each file written[7], around 23 KiBs per file.

To sum up, we find that Swift incurs the lowest overhead and NFS incurs moderate overhead. On the contrary, iSCSI suffers from the amplification caused by ext4 file system for small files and for write operation.

[7]See Section 4.2 for more details.

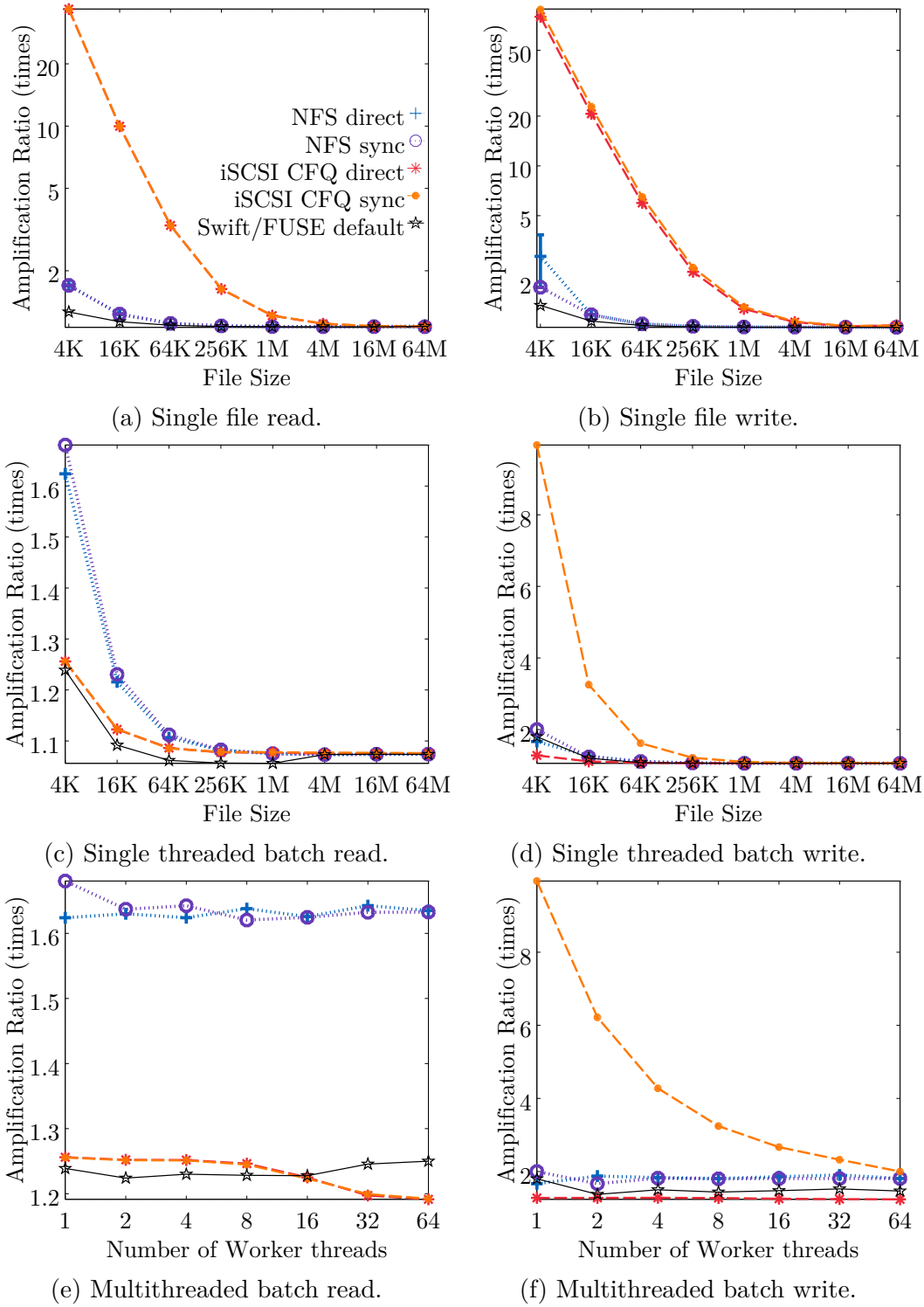


Figure 4.9: Overhead of file access. Ideal network. Amplification ratio is computed by dividing total transmission in both directions by file size for single file access and by dividing the sum of line rates in both directions by throughput for others. Note that two iSCSI lines coincide in all read plots and the different scales.

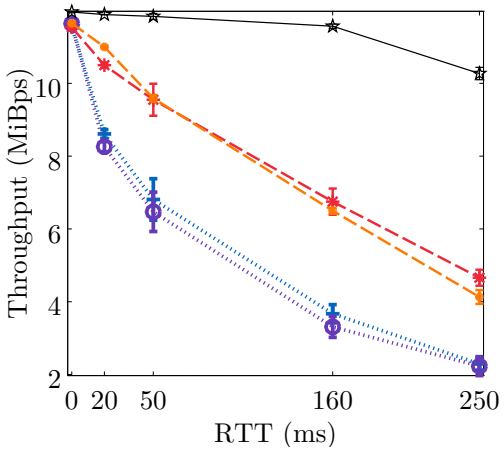
4.8 Increased bandwidth

In this section we briefly examine the effect of increasing bandwidth to 1 Gbps. Such links will become available in the near future through Fibre to the Home (FTTH) or LTE-Advanced [71] subscriptions, or through IEEE 802.11ac [33] networks at home or in office. Furthermore, the larger bandwidth delay product (BDP) resulted may amplify the effects of network complexities on the performance [28, 77].

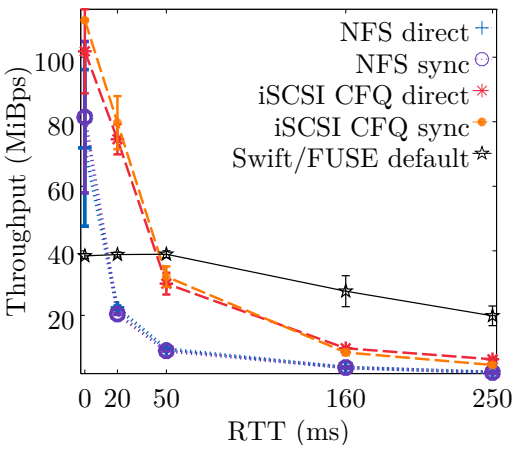
Comparing Figure 4.10b to Figure 4.10a, we see that the curves become steeper when the bandwidth increased to 1 Gbps. The only exception is Swift which remained relatively constant at a medium level. This showed that iSCSI is more capable of "filling the pipe", or allows more outstanding requests than NFS. As mentioned in Section 4.6.2, CloudFuse is capable of using multiple TCP connections each having its own congestion window. Swift/CloudFuse thus suffered the least from increased BDP.

Larger BDP also amplified the difference between different block I/O schedulers concerning iSCSI based storage (see Section 4.6.3). Figure 4.10d and 4.10c suggests that, since deadline scheduler allows more outstanding requests, it is more resistant to network latency and increased BDP. Similar phenomenon is also observed in Figure 4.10e (compared to Figure 4.8c) and Figure 4.10f (compared to Figure 4.8d).

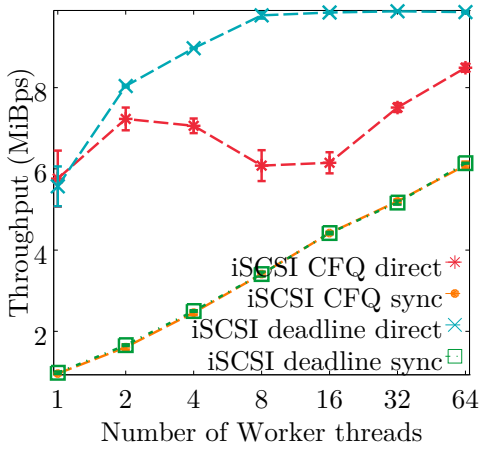
Overall, we find that iSCSI with deadline block I/O scheduler and the concurrent TCP connections of Swift/CloudFuse are highly resistant to the increased BDP.



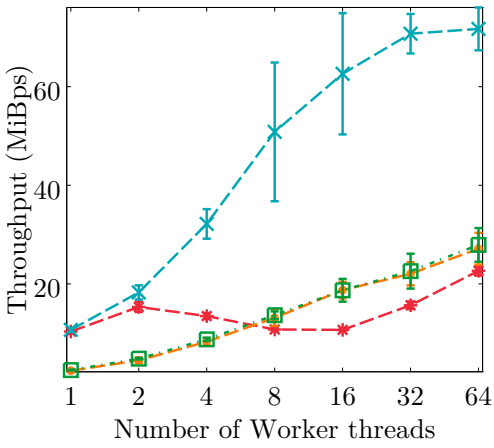
(a) 100 Mbps. Reading 1 MiB files using 32 threads.



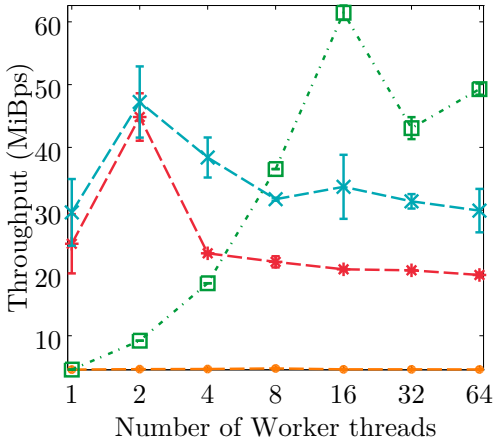
(b) 1 Gbps. Reading 1 MiB files using 32 threads.



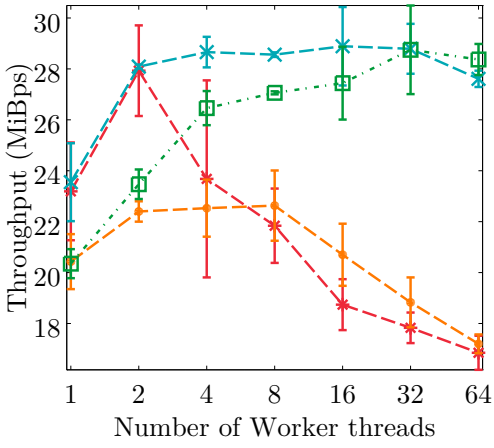
(c) 100 Mbps. iSCSI writing 4 KiB files.



(d) 1 Gbps. iSCSI writing 4 KiB files.



(e) Reading 16 MiB files, iSCSI under 1 Gbps bandwidth and 50 ms latency.



(f) Writing 16 MiB files, iSCSI under 1 Gbps bandwidth and 50 ms latency.

Figure 4.10: Effect of increased bandwidth. Note the different scales.

4.9 Summary

In this chapter, we introduced the design of our microbenchmark in Section 4.1 and discussed two important parameters to POSIX file API – form of access (Section 4.2) and access unit size (Section 4.3). Single file access experiments are discussed in Section 4.4. We found that the larger the file and the access unit, the higher the throughput. Moreover, network latency has a great effect on throughput and network loss impacts the access of large files severely.

Discussion on batch file access are divided into two parts – single threaded application (Section 4.5) and multithreaded application (Section 4.6). Generally, iSCSI excels under ideal network condition. All of NFS, iSCSI, and Swift, are severely impacted by network latency. Network loss and jitter further degrades performance. In particular, performance of iSCSI in synchronised operation stopped increasing for files larger than 4 MiB.

For single threaded application, iSCSI delivers the best performance for small file access and all three systems perform on the same level for large files. For multithreaded application and small files, Swift is the best for read and iSCSI is the best for write due to the aggregation of ext4 file system. For large files, Swift delivers outstanding performance due to its capability of utilising concurrent TCP connections.

The impact of Linux block I/O schedulers is discussed in Section 4.6.3. We found that block I/O scheduler plays an important role in iSCSI based storage systems, particularly for synchronised file operations.

Transmission overhead, or amplification, is investigated in Section 4.7). It is found that Swift incurs the lowest overhead and NFS incurs moderate overhead. On the contrary, iSCSI suffers from the amplification caused by ext4 file system for small files and for write operation.

In Section 4.8 we investigated the impact of increasing network bandwidth to 1 Gbps. We found that iSCSI with deadline block I/O scheduler and the concurrent TCP connections of Swift/CloudFuse are highly resistant to the increased BDP.

Chapter 5

Metadata Operations

In this chapter, we compare the performance of three storage systems on metadata operations. The benchmark simulates a user giving commands via operating system shell. It creates n directories with `mkdir` in the working directory, then creates n empty files with `touch`, then lists the contents of current directory with `ls`, and then deletes the files with `unlink` and the directories with `rmdir`. The working directory is initially empty. Command `sync` is issued and after that local kernel cache is cleaned (see Figure 4.1) between different operation runs, such as after n `rmdir`'s and before n `unlink`'s. Completion time and traffic volume are measured from the beginning of the first operation of the same kind until the completion of the corresponding `sync`.

We experimented with 100 Mbps bandwidth and 1 Gbps bandwidth and found no noticeable difference. This is because that metadata operations are not data intensive. Therefore, we report only the results from experiments with 100 Mbps link. We found that the performance of `mkdir`,`touch`,`unlink`, and `rmdir` all show the same trend. On the other hand, `ls` performance is similar to reading files with a small access unit size (Section 4.3).

5.1 Completion time

In this section, we compare the performance based on the time required to complete a given number of metadata operations. The completion time of `mkdir` operations and the effect of network latency and network loss are illustrated in Figure 5.1.

We see from Figure 5.1a that iSCSI and NFS are comparable under ideal network condition, whereas Swift is an order of magnitude slower. The reason turned out to be internal coordination of different Swift components (see Section 2.3). Each request requires approximately 30 ms to complete. When latency and other complexities are added to the network, as illustrated in Figure 5.1b, round trip time (RTT) began to dominate the completion time.

Another interesting finding from the same figure is that the completion time of NFS soared after the introduction of network complexities. We found, by examining the traffic capture, that NFS requires two round trips for each metadata operation – a CREATE operation followed by an ACCESS operation. It is noted that the ACCESS RPC is used extensively in NFSv4 [58].

Figure 5.1c illustrates the effect of network latency on completion time. Completion time of NFS and Swift are proportional to RTT, whereas iSCSI is only slightly affected due to the aggregation of ext4 file system [58]. In our

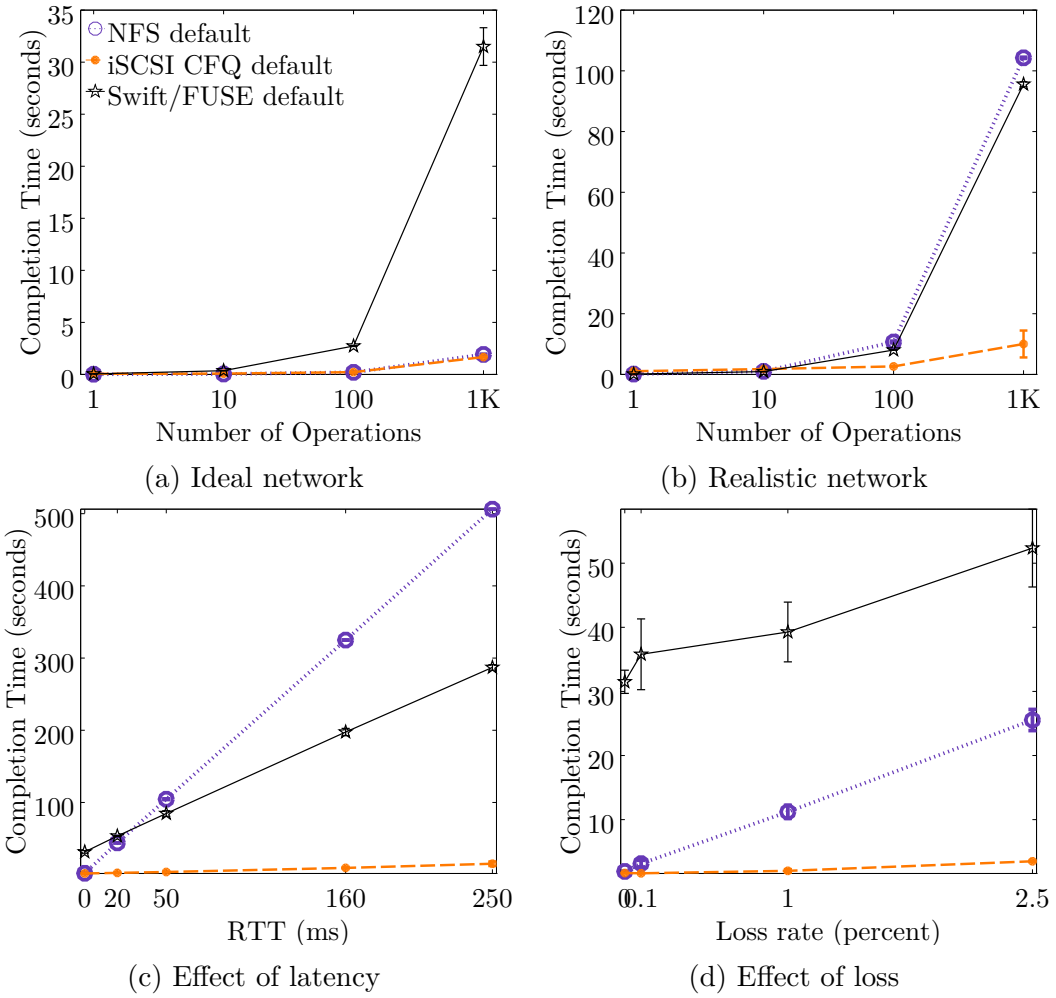


Figure 5.1: Completion time of `mkdir` operations. 1000 operations were carried out in (c) and (d). Loss is 0% for latency plots and latency is 0 ms for loss plots. Note the different scales.

emulated Internet, the mean RTT is 50 ms, so each operation on Swift requires approximately

$$T_{Swift} = T_{internal} + RTT = 30 + 50 = 80\,\text{ms}$$

to complete and each operation on NFS requires approximately

$$T_{NFS} = 2 * RTT = 100\,\text{ms}$$

to complete.

The impact of network loss on completion time is illustrated in Figure 5.1d. Packet capture revealed that approximately 3% of the packets of NFS and Swift are retransmissions under 2.5% packet loss. This results in approximately 120 retransmissions each causing 0.2 s extra latency and sums up to 24 seconds in the completion time. On the other hand, approximately 7% of the packets

of iSCSI are related to TCP retransmission. However, iSCSI completion time only increased slightly because that, due to ext4 aggregation, the transmission is longer and thus benefits from TCP fast retransmit.

5.2 Network traffic

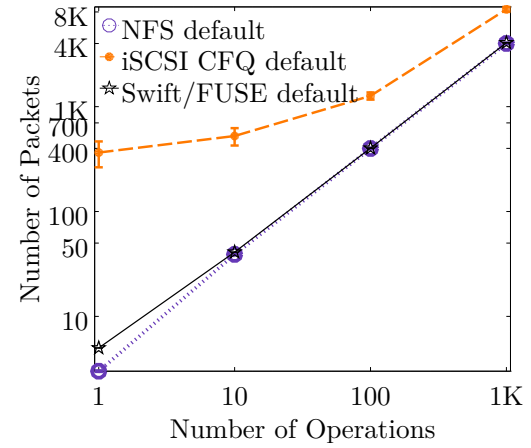
In this section, we examine the network traffic incurred by a given set of metadata operations. The number of packets, mean packet size, and the effect of network latency and network loss are illustrated in Figure 5.2.

From Figure 5.2a and 5.2b we see that, since individual operations on NFS and Swift are independent, the number of packets increases linearly as the number of operations increases while the packet size remained constant. On the other hand, due to the caching and aggregation of ext4 file system on iSCSI disk [58], traffic overhead is substantial when the number of operations is small. This is similar to the overhead incurred by batch operations of small files (Section 4.7). Mean packet size of iSCSI increased slightly in Figure 5.2b as the size of batch transmission increased.

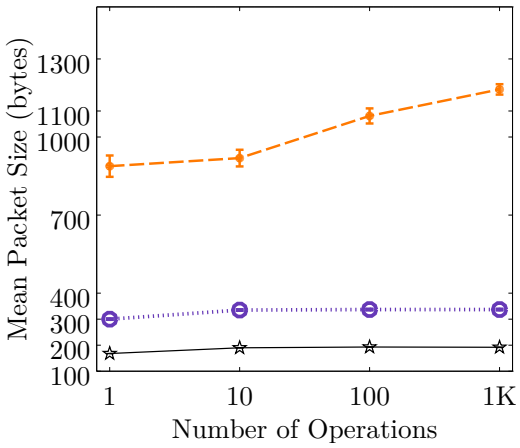
From Figure 5.2c and 5.2d we see that network latency affects only minimally. We note that the increased number of packets in Figure 5.2c is due to other background activities on the testbed during the prolonged observation period.

The effect of network loss on traffic is illustrated in Figure 5.2e and 5.2f. From Figure 5.2e we see that, since the traffic volume of iSCSI is several times higher than others, the resulting retransmission and thus the increase in number of packets is also higher under the same loss rate. Meanwhile, iSCSI benefits from TCP fast retransmit, packets involved are mostly duplicate ACKs which are small, thus lowered the mean packet size, as illustrated in Figure 5.2f.

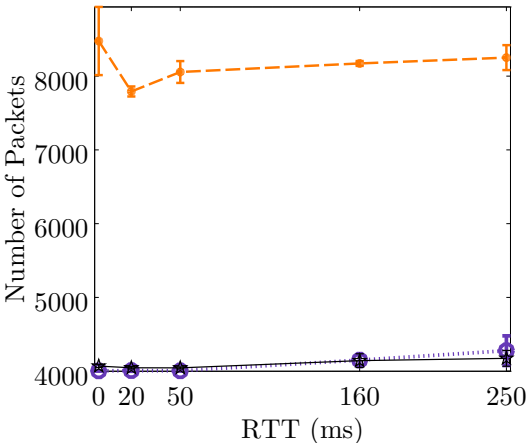
On the other hand, NFS and Swift operations fit into one single TCP packet. Their retransmissions involve mostly self contained RPCs or HTTP requests. Therefore, although the number of packets increases due to retransmission, the mean packet size remains the same.



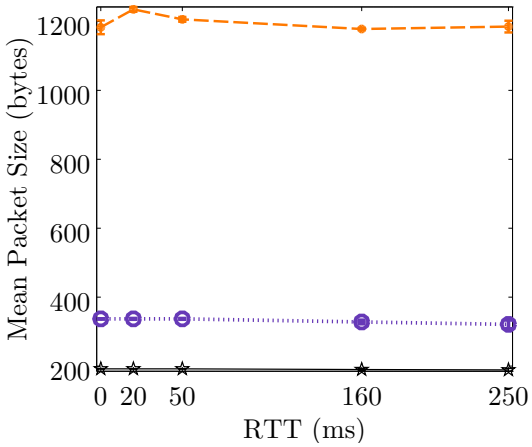
(a) Ideal network; number of packets.



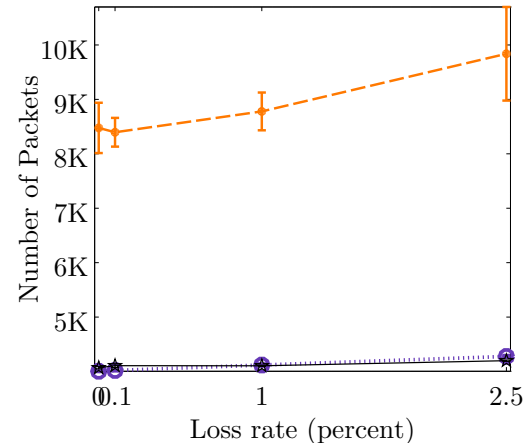
(b) Ideal network; mean packet size.



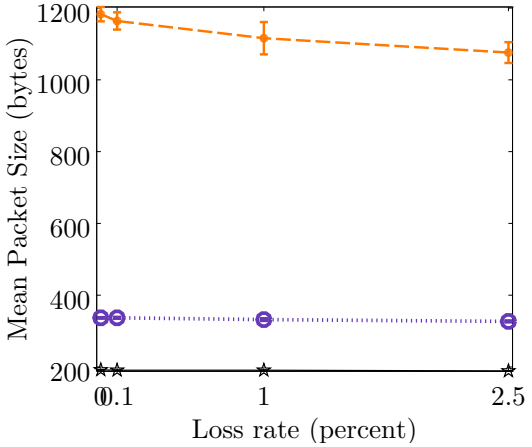
(c) Effect of latency; number of packets.



(d) Effect of latency; mean packet size.



(e) Effect of loss; number of packets.



(f) Effect of loss; mean packet size.

Figure 5.2: Network traffic of metadata operations. 1000 operations in total. Loss is 0% for latency plots and latency is 0 ms for loss plots. Note the different scales.

Chapter 6

Macrobenchmarking

In microbenchmarking experiments discussed in previous chapters, we simulated an application accessing the storage service in a batch pattern. Our macrobenchmarking experiments, in contrast, simulate the behaviours of some typical applications which access the storage service interactively.

We used PostMark [36] and FileBench [67] for macrobenchmarking experiments to evaluate the performance of the storage systems under workloads similar to real applications. Both PostMark and FileBench are configurable to simulate a wide range of application workloads and are frequently used in research works. They are configured to simulate the behaviours of some typical applications which an individual user might use.

The metric in macrobenchmarking experiments is completion time for PostMark and IOPS for FileBench. TCP offload engines remain disabled in this chapter.

6.1 PostMark

PostMark is a single threaded workload generator, originally developed by Jeffrey Katcher of NetApp, Inc. in 1997[1]. It measures the performance of file system using a configurable workload which consists of many short lived, small files.

PostMark first prepares a pool of files, then performs a mixture of file create, file read, file append, file write, and file delete operations on the pool. This access pattern is similar to that of a typical mail server or a heavy user of e-mail. After finishing the assigned amount of transactions, it deletes the remaining pool.

Our workload configuration is as follows: 500 files, ranging from 5 KiB to 512 KiB in size; a total of 25000 transactions; and using unbuffered I/O. All other parameters are left as default. The results from tests under different network conditions are summarised in Table 6.1.

[1]We used version 1.53 from Launchpad in this thesis. It is available at `https://launchpad.net/ubuntu/+source/postmark`

Network	**NFS**		**iSCSI CFQ**		**iSCSI deadline**		**Swift**	
	IOPS	**Time**	**IOPS**	**Time**	**IOPS**	**Time**	**IOPS**	**Time**
100 Mbps ideal	15	1642	1176	23	1242	20	6	3781
added 50 ms latency	2	11681	1250	22	1190	22	2	15506
added 0.1% loss	1	14248	1230	22	1230	21	1	19005
Realistic network	1	15643	1210	22	1210	21	2	11841
1 Gbps ideal	24	1062	1293	20	1293	19	8	2756
added 50 ms latency	2	10856	1026	26	1138	22	1	16951
added 0.1% loss	2	12117	749	52	1163	22	1	24624
Realistic network	1	16352	630	188	1220	22	0	26866

Table 6.1: PostMark results. Completion time is in seconds.

It is clear from the results that iSCSI benefits greatly from aggregation of ext4 file system. On the contrary, NFS and Swift suffer severely from network latency. By examining the log files and network traffic, we find that NFS and Swift require several message exchanges to complete each operation. It is thus highly sensitive to network latency [76].

The difference in resistance against increased BDP between different block I/O schedulers is again highlighted. When network bandwidth is increased to 1 Gbps, iSCSI performance halved in the case of lossy and realistic network when CFQ scheduler is used, whereas the performance remained almost unaffected with deadline scheduler.

What caused the variation in Swift completion time under 100 Mbps bandwidth is unclear, but we believe that it is due to Swift internal cooperation, TCP internals, and the interaction between CloudFuse and Linux FUSE.

From these results we conclude that special attention must be paid when storing data such as mailbox or web browser cache over network storage.

6.2 FileBench

FileBench is a flexible workload generator which can theoretically be programmed to simulate workloads of any application. FileBench was originally developed by SUN Microsystems in 2004 and the version used in this section is 1.4.9.1. In this section, we investigate the performance of three storage systems under two different workloads[2].

In the work flow of FileBench, it first prepares a set of files with given characteristic, then performs the test. Performance measurements are reported after a specified length of time.

For the sake of timeliness, we disable network emulator during the preparation phase and re-enable it before the testing phase[3]. All measurements were made after running the test for 120 seconds.

6.2.1 `varmail` workload

This workload simulates the access pattern of a simple mail server which stores each mail in a separate file. It consists of 1000 files with average size 16 KiB, 80% of which are prepared beforehand into a flat directory. File operations are a mix of create-append-sync, read-append-sync, read, and delete operations from 16 worker threads. Average append size is 16 KiB and access unit size is 1 MiB.

From the results summarised in Table 6.2 we see that the general trend is similar to our other experiments. Due to the numerous, small, synchronised operations, the performance is impacted more by network latency than by network loss, as illustrated in Figure 6.1. Increasing bandwidth to 1 Gbps increases BDP and amplifies the effects.

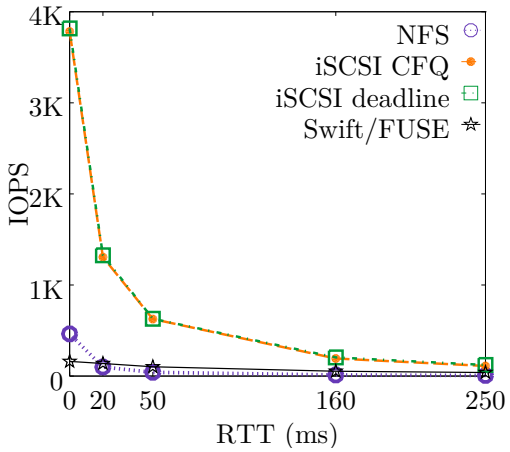
However, compared to PostMark results, the difference between NFS and iSCSI narrowed significantly and Swift throughput remained relatively constant. This is likely due to the increased data activity.

[2] *personality* in FileBench terminology.

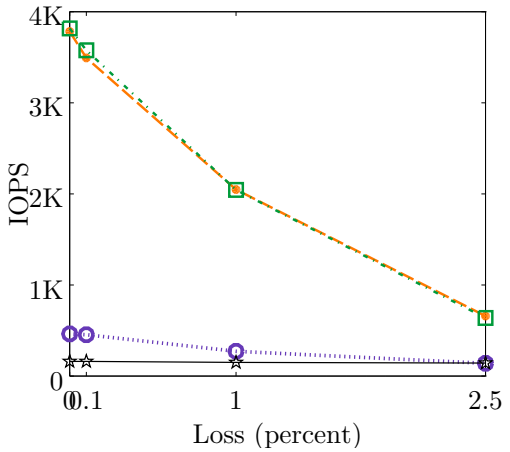
[3] A tiny modification to FileBench was made. Invocation of relevant commands are inserted into `fileset_createsets()` in `fileset.c`.

Network	NFS	iSCSI CFQ	iSCSI deadline	Swift
100 Mbps ideal	463	3783	3817	161
added 50 ms latency	41	626	629	102
added 0.1% loss	40	318	362	104
Realistic network	37	291	348	99
1 Gbps ideal	461	14813	15175	158
added 50 ms latency	41	633	647	104
added 0.1% loss	40	405	457	104
Realistic network	38	196	205	102

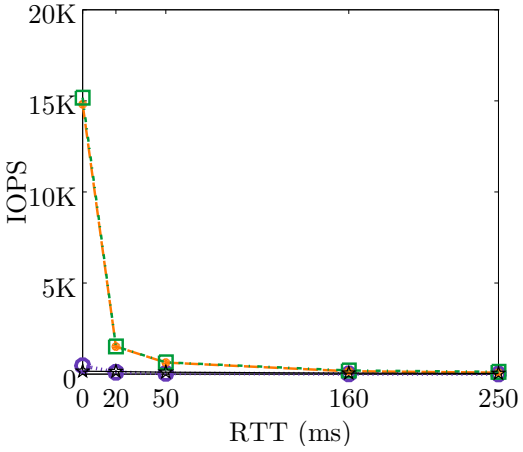
Table 6.2: FileBench `varmail` results. Run time: 120 s. Unit: IOPS.



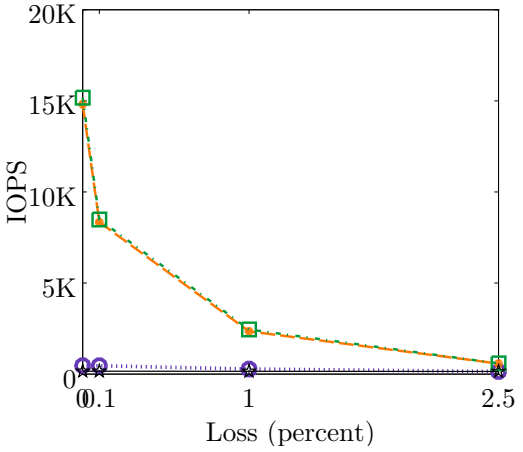
(a) Effect of latency. 100 Mbps network.



(b) Effect of loss. 100 Mbps network.



(c) Effect of latency. 1 Gbps network.



(d) Effect of loss. 1 Gbps network.

Figure 6.1: Effects of network latency and loss. FileBench `varmail` workload. Run time: 120 s. Unit: IOPS. Loss is 0% for latency plots and latency is 0 ms for loss plots. Note the different scales.

6.2.2 `fileserver` workload

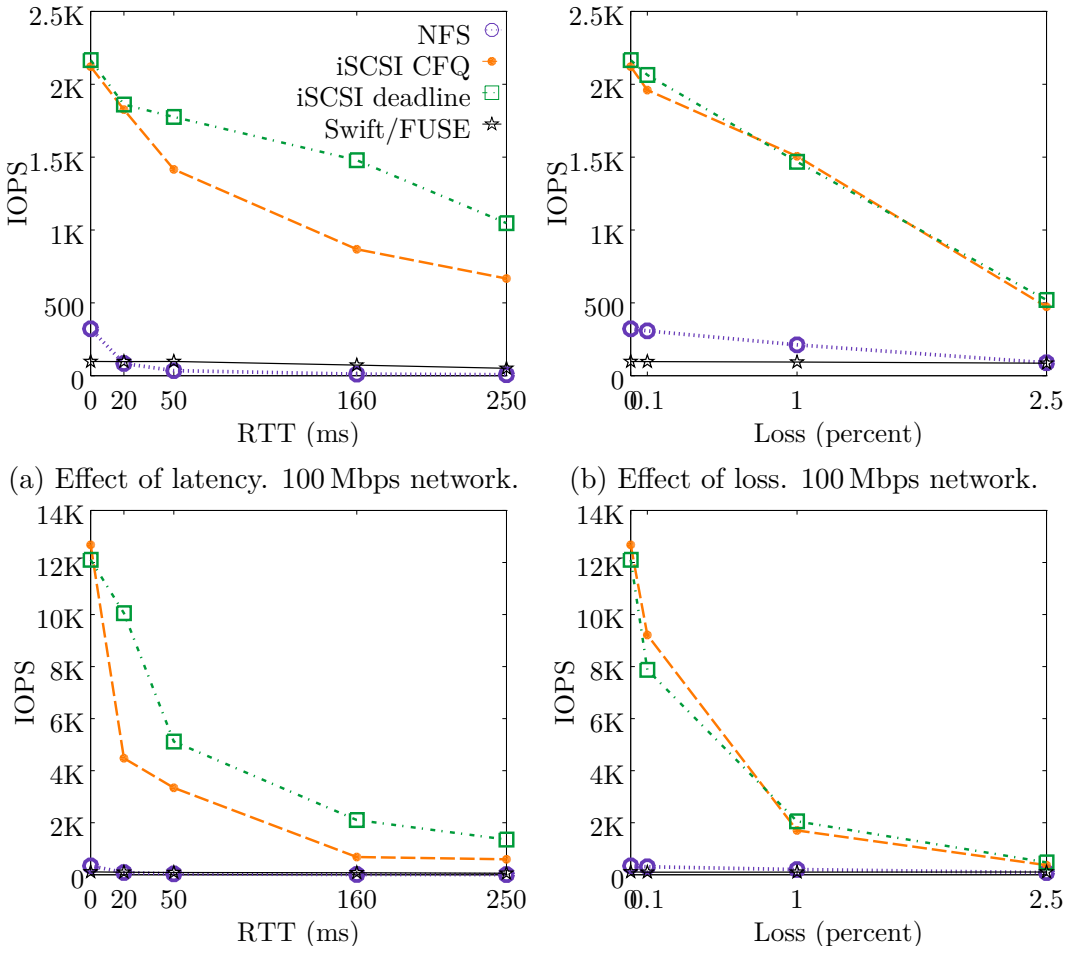
This workload simulates the I/O activity of a simple file server. It consists of 10000 files with average size 128 KiB, 80% of which are prepared before other operations into a directory tree of average width 20. File operations are a mix of file create, read, write, append, and delete, from 50 worker threads. Average append size is 16 KiB and access unit size is 1 MiB.

From the results summarised in Table 6.3 and illustrated in Figure 6.2 we see that its performance is less sensitive to network latency. Compared to `varmail` workload, `fileserver` workload is composed of larger asynchronous file access. The increased data intensity is also confirmed by the difference in performance between iSCSI with CFQ scheduler and iSCSI with deadline scheduler.

Multiple round trips required for NFS operations still limited its performance, despite the increased data intensity. On the other hand, Swift throughput is dominated by whole object transmissions and remains relatively constant. We conjecture that the increased IOPS when loss is introduced is likely due to TCP internals.

Network	**NFS**	**iSCSI CFQ**	**iSCSI deadline**	**Swift**
100 Mbps ideal	322	2121	2166	98
added 50 ms latency	35	1416	1776	98
added 0.1% loss	31	377	419	104
Realistic network	29	384	367	98
1 Gbps ideal	342	12678	12104	111
added 50 ms latency	35	3341	5120	87
added 0.1% loss	32	621	558	92
Realistic network	25	293	314	89

Table 6.3: FileBench `fileserver` results. Run time: 120 s. Unit: IOPS.



(a) Effect of latency. 100 Mbps network.

(b) Effect of loss. 100 Mbps network.

(c) Effect of latency. 1 Gbps network.

(d) Effect of loss. 1 Gbps network.

Figure 6.2: Effect of network latency and loss. FileBench `fileserver` workload. Loss is 0% for latency plots and latency is 0 ms for loss plots. Run time: 120 s. Unit: IOPS. Note the different scales.

Chapter 7

Discussion

Generally speaking, under an ideal network condition to which LANs are often similar, we find that iSCSI outperforms NFS and Swift. This result is foreseeable since iSCSI encapsulates SCSI commands whose main usage is to exchange data between host computer and peripheral devices over near ideal interconnections. Furthermore, iSCSI based storage systems work at block level and thus benefit from aggregation and caching by the file system, especially for small files and metadata. Note, however, that there is a potential tradeoff between consistency and performance pertaining to it – iSCSI may acknowledge completion immediately to achieve performance at the risk of failure before relevant blocks are written back to the remote storage.

On the contrary, NFS and Swift work at file level. Each whole file operation entails at least one request-response round trip, regardless of the size of the file accessed. Thus, they deliver similar and worse than iSCSI performance. They are therefore unsuitable to replace iSCSI in performance critical environments. However, taking into account the features of cloud storage, such as resilience and elasticity, a properly deployed installation of Swift is a viable replacement for NFS.

Network complexities, such as latency, packet loss, and jitter, are present in realistic network environments, such as the Internet. We see from Section 4.6.2 that Swift excels under unideal network conditions for large file transmissions. Swift is designed to allow parallel access to objects stored in the cloud storage over the Internet. Network complexities were taken into consideration from the very beginning and the use of multiple TCP connections are supported. The use of concurrent TCP connections substantially improved the throughput when faced with network complexities.

Our Swift implementation was continuously updated during the course of this thesis. As mentioned by Cooper et al. [13], different versions may exhibit different behaviours and deliver different performance. This is mitigated by repeating the experiments over the time. The variance was found to be small and we thus believe the bias contributed by continuous update to be minimal.

7.1 Recommendations

Our microbenchmarking results suggest that bundling small requests into larger ones and breaking very large requests into moderately sized ones are likely to improve performance. Similar approaches are also proposed by Drago et al. [15, 16]. Moreover, access unit should be as large as possible. For iSCSI this

also implies increasing maximum data segment length per iSCSI command[1] and largest access unit size per SCSI command. Furthermore, larger transmission unit would also improve energy efficiency in next generation access network [32].

Thanks to the capability of utilising concurrent TCP connections, Swift exhibited outstanding endurance against network complexities. NFS and iSCSI can arguably benefit from the use of multiple TCP connections as well. However, due to the increased complexity, such capability is generally available only in commercial implementations.

For iSCSI based storage systems, factors such as Linux block I/O schedulers [2] and local file system [64] play important roles. They deserve attention and shall be carefully configured according to the expected workload, the network condition, and the system architecture [6, 57]. Such effort may be relieved with the assistance from machine learning [78].

7.2 Future research directions

7.2.1 Feature

Hacker et al [27] studied the effect of concurrent TCP connections on performance in a lossy network. Using an appropriate number of connections improves performance as it speeds up the recovery from loss. However, an excessive amount of connections may cause congestion in the bottleneck link and thus have negative effect. Depending on the loss distribution, RTT, and MTU, the optimal number of concurrent TCP connections differs. They also mentioned that using multiple TCP connections may have an impact on fairness and that determining the right number of connections in advance is difficult. The suggested way is to collect statistics on the fly and adjust dynamically.

Another problem may arise is that whether this would bring too much load to the server side. This happens when there are many clients utilising multiple TCP connections being served by the same server.

Server side CPU utilisation and others may be of interest as well. Sehgal et al. [64] found that the file system used, format parameters, and mount options have a significant impact on the performance and energy efficiency of storage systems. Default options generally lead to suboptimal performance. Appropriate options must be determined according to expected workload and by testing.

Our results in Chapter 5 suggests that coalescing more NFS operations into one single COMPOUND operation could improve performance. However, as Pawlowski et al. [56] mentioned, combining unrelated operations into one COMPOUND operation may increase the complexity of error handling and recovery, as the process of the COMPOUND operation halts as soon as an error is encountered when processing constituent operations in it. Neverthe-

[1] Up to 16 MiB should be possible since `DataSegmentLength` is 3 bytes long.

less, the possibility to coalesce the seemly redundant ACCESS operation with corresponding CREATE operation can still be investigated.

In preliminary experiments we launched multiple processes deleting files from the same Swift container and found the performance to be the same as a single process. Swift log indicates that the requests from parallel clients are processed sequentially. We conjecture that the underlying SQLite database is the most likely bottleneck.

We observed 30 ms processing time for each operation caused by internal coordination of Swift in Section 5.1. It is anticipated that in a fully distributed and replicated Swift installation, such internal coordination would take longer. For Swift to be useful for metadata intensive workloads, its internal mechanism and the design of the application require further consideration.

It can be argued, though, that applications written for cloud services such as Swift may not require a POSIX file interface at all. In this thesis, we tried to investigate the possibility of replacing traditional storage with cloud based object storage with minimal modification to the application. It is, of course, possible to construct, to extend[2], or to adapt an existing application or benchmark to evaluate traditional storage systems and cloud based object storage services using their native interfaces.

7.2.2 Depth

We have seen in Figure 4.2 and 4.4b that write operation of iSCSI under direct I/O is much faster than under synchronised I/O. At the moment we conjecture that the difference is due to the absence of local buffering.

The block size of our iSCSI target is configured to 1024 bytes instead of 512 bytes of the underlying hard disk. Whether this block size emulation has an impact on performance or not is yet to be investigated.

As mentioned earlier in Section 3.2, Swift occasionally returns HTTP 404 for some requests. Although we conjecture that this is due to cooperation between Swift components, the exact nature is unclear and could be further investigated. Our modification to CloudFuse was more a preventive measure. Furthermore, the occasional failures of NFS and iSCSI mentioned in Section 4.5 are yet to be investigated as well.

In Section 4.6 we found that Swift outperforms NFS and iSCSI for large files under unideal network condition. More insights may be gained by disabling multithreading support of CloudFuse. In addition, NFS performance stopped increasing between 8 to 16 threads (sec 4.6). The bottleneck is likely a small sized queue in some layer.

We observed tremendous and counter-intuitive variation in Swift performance under 100 Mbps bandwidth in Section 6.1. Furthermore, in Table 6.3 we see that the performance of Swift increased slightly when packet loss is

[2]This would be easier for benchmarks which allow different back end interface modules to be plugged in easily [13].

introduced. The reason is unclear and we conjecture that it is due to Swift internal cooperation, TCP internals, and the interaction between CloudFuse and Linux FUSE.

Since we are studying network storage systems and all of them use TCP for transport, the protocol itself could also be examined further. For example, the distribution of packet size, variation of TCP congestion window size, and whether Nagle's algorithm [48] has an effect on the performance, to name a few.

7.2.3 Scope

This thesis is limited to NFS of Linux kernel, Intel iSCSI, and OpenStack Swift with CloudFuse only. To further generalise the comparison to NAS, SAN, and cloud storage, more samples and implementations from each family of storage systems have to be examined.

Other metrics, such as performance per euro (€), availability, reliability, and elasticity, may be more of interest in evaluating the viability of replacing NAS or SAN with cloud storage. This could be further augmented by increasing the complexity of the scenario, such as to inject crashes into storage systems, inject outages and load spikes into the network.

Preliminary experiments with wireless interface showed asymmetry between download and upload link capacity. Moreover, we found that by replacing the home router with another one this phenomenon vanishes. This could be further investigated as well.

As Feeney et al. [20] indicated, energy consumption of wireless interface in ad hoc scenarios cannot be determined by transmission throughput. The applicability of our results in wireless environments is thus limited to those with a base station.

7.2.4 Methodology

As discussed in Section 4.5, our benchmark has to be run long enough to obtain steady state performance. Dukkipati et al. [17] suggested that increasing TCP initial window size overcomes slow start. This could relieve or even remove the need to run the benchmark long enough.

It has been noted that much effort could be saved with automation. Such as automating the process of sweeping through parameter space [66] or adjusting the workload according to the environment and the capacity of the platform under test [12].

It may be beneficial to the storage community by constructing mathematical models for storage systems and benchmarks [23] or formulating the system and corresponding measurements as inverse problems. That way, the performance and the configuration can be studied mathematically.

Chapter 8

Conclusions

Mobile platforms with access to high speed wireless network have become ubiquitous. Advancements in network technology and consumer electronics have brought traditional storage systems into offices and homes. Services based on cloud technologies, including object based storage, have gained popularity among both private users and enterprises. However, battery capacity has seen only limited improvement. Energy efficiency has thus become the salient concern in mobile computing.

Studies indicate that energy consumption of a mobile platform depends greatly on the transmission throughput of wireless interface. In this thesis, we evaluated the performance of three popular storage systems, namely NFS, iSCSI, and OpenStack Swift, which can potentially be used by mobile platforms over wireless network.

We built a testbed and an in house, ad hoc microbenchmark to study the impact of network complexities and access behaviours of an application. In addition, we employed two widely used macrobenchmarks, PostMark and FileBench, to simulate the interactive workloads of typical applications. Application throughput is the primary metric in this thesis.

Under our assumption of exclusive access from single user, we found that iSCSI excels in networks whose condition is as good as LAN, whereas NFS and Swift are more suitable for complex networks such as wireless network and WAN. Furthermore, we found Swift to be a viable replacement for NFS in all investigated scenarios.

Note, however, that system configuration on the client side impacts storage performance significantly and deserves adequate attention. Based on our observations, we made recommendations to storage system implementors and operators as well as application developers. Moreover, we pointed out numerous possible directions for future research.

As the final remark, we found that there is no “silver bullet”, or “one fits all” storage solution.

Appendix A

Cloud storage examples

Service/Software	License	Website
115 Cloud drive	Proprietary	http://www.115.com/
Amazon S3	Proprietary	http://aws.amazon.com/s3/
Apple iCloud	Proprietary	https://www.icloud.com/
ASUS WebStorage	Proprietary	http://asuswebstorage.com/
Baidu Cloud Storage	Proprietary	http://pan.baidu.com/
Dropbox	Proprietary	http://dropbox.com/
Google Drive	Proprietary	http://drive.google.com/
KuaiPan	Proprietary	http://kuaipan.cn/
OneDrive	Proprietary	http://onedrive.live.com/
Saunalahti Pilvilinna	Proprietary	http://saunalahti.fi/pilvilinna/
OpenStack Swift	Apache License	https://wiki.openstack.org/Swift
ownCloud	GNU AGPLv3	https://owncloud.org/
pCloud	Proprietary	https://www.pcloud.com/
Seafile	GNU GPLv3	http://seafile.com/

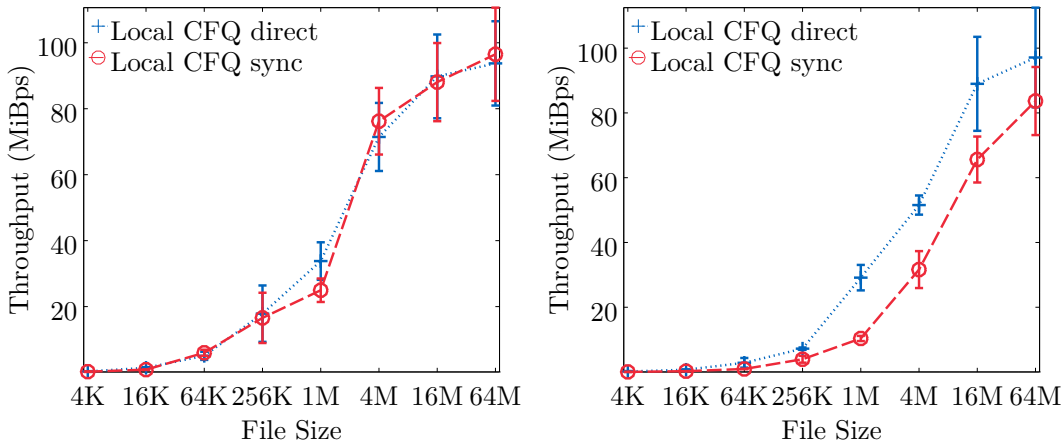
Table A.1: Cloud storage examples. For most of them the server side is proprietary and cannot be set up in a laboratory environment.

Appendix B

Baseline performance

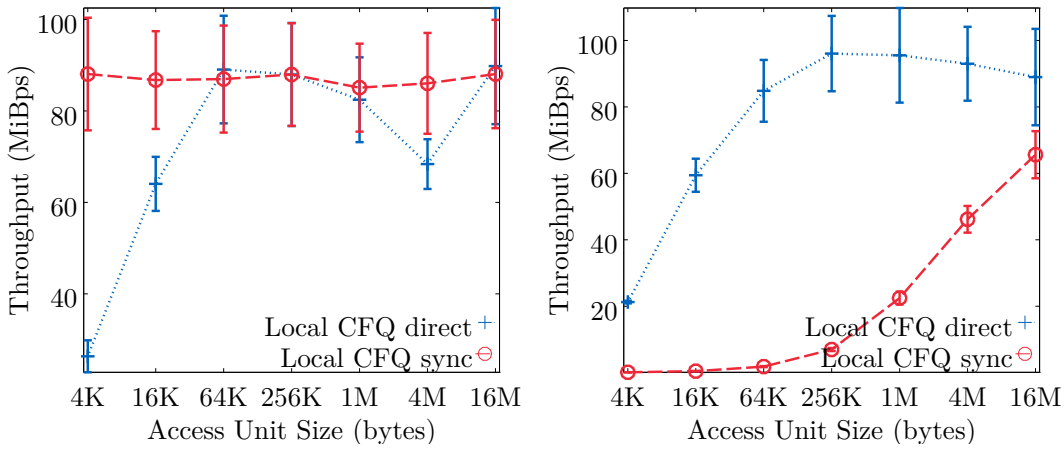
The baseline throughput of the hard disks, as mentioned in Section 3.3, measured by conducting sequential read and write using `dd` system tool with `direct` flag set[1] is 110 MiB/s for read and 100 MiB/s for write.

Baseline performance of macrobenchmarks and macrobenchmarks when run on the server machine is illustrated in the following figures and table.



(a) Read (b) Write

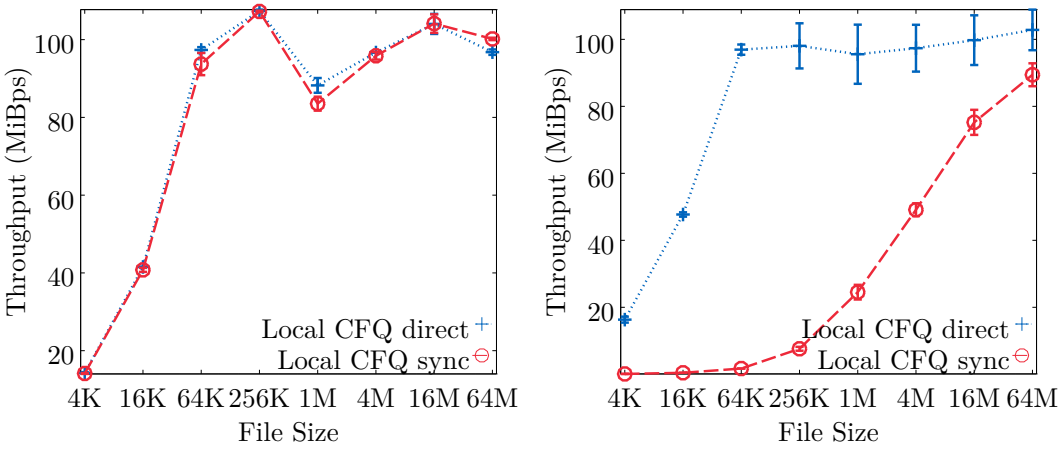
Figure B.1: Baseline – Microbenchmark – Single file access.



(a) Read (b) Write

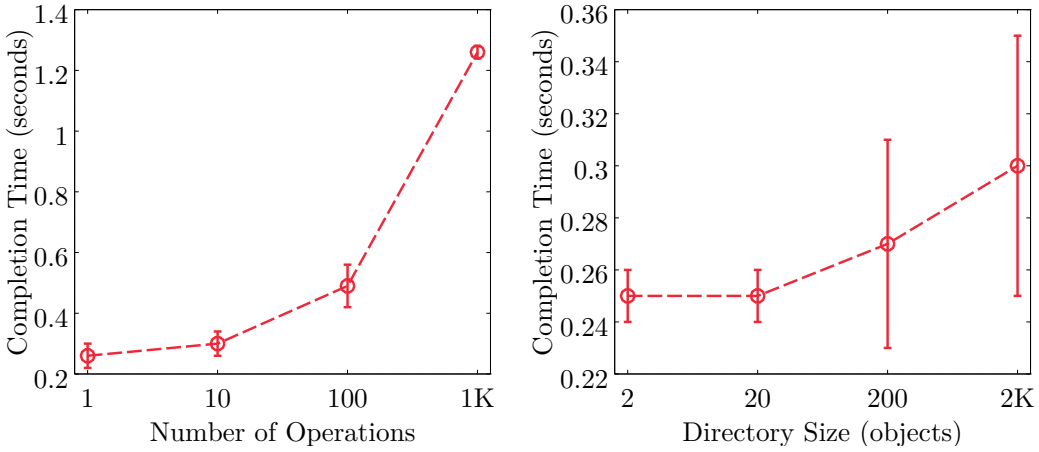
Figure B.2: Baseline – Microbenchmark – Effect of access unit size.

[1]This causes `O_DIRECT` to be passed to `open()` internally.



(a) Read (b) Write

Figure B.3: Baseline – Microbenchmark – Single threaded operation.

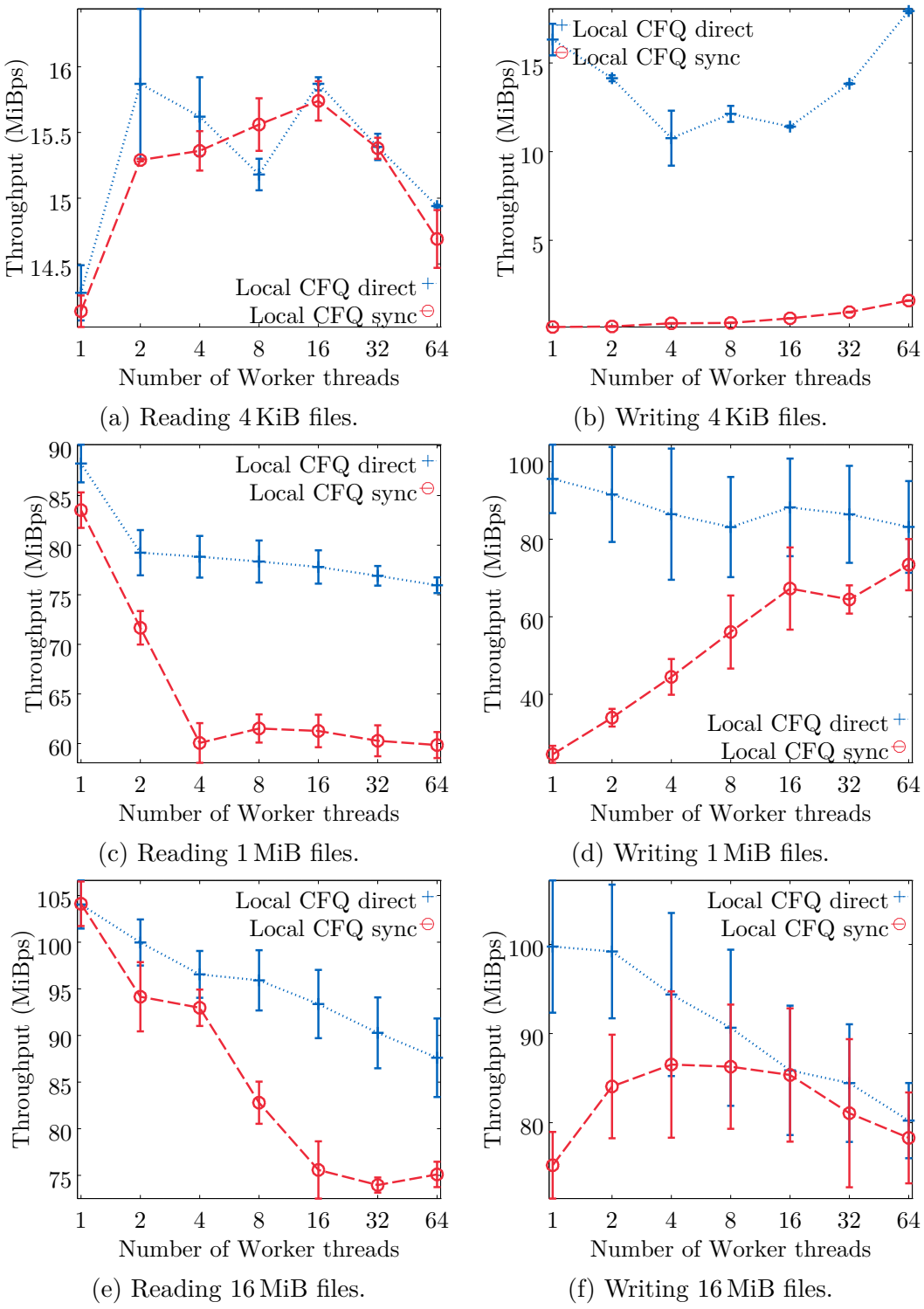


(a) `mkdir` (b) `ls`

Figure B.4: Baseline – Microbenchmark – Metadata operations. Note the different scales.

Benchmark	**Configuration**	**IOPS**	**Time**
PostMark	Section 6.1	1176.5	21.25
FileBench	`fileserver`	6703.62	120 (fixed)
FileBench	`varmail`	816.165	120 (fixed)

Table B.1: Baseline – Macrobenchmarks.



(a) Reading 4 KiB files.

(b) Writing 4 KiB files.

(c) Reading 1 MiB files.

(d) Writing 1 MiB files.

(e) Reading 16 MiB files.

(f) Writing 16 MiB files.

Figure B.5: Baseline – Microbenchmark – Multithreaded operation. Note the different scales.

Bibliography

[1] Mark Allman, Vern Paxson, and Ethan Blanton. TCP congestion control. RFC 5681, September 2009. URL `http://tools.ietf.org/rfc/rfc5681.txt`. 18

[2] Jens Axboe. Linux block IO - present and future. In *Linux Symposium*, pages 51–61, 2004. 13, 32, 50

[3] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 165–176, April 2003. doi: 10.1109/MASS.2003.1194853. 2

[4] Michael Barton. CloudFuse. URL `http://redbo.github.io/cloudfuse/`. 8, 13

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. ISSN 0734-2071. doi: 10.1145/2080.357392. 5

[6] David Boutcher and Abhishek Chandra. Does virtualization make disk scheduling passé? *SIGOPS Oper. Syst. Rev.*, 44(1):20–24, March 2010. ISSN 0163-5980. doi: 10.1145/1740390.1740396. 50

[7] Luis-Felipe Cabrera and Darrell Don Earl Long. Swift: a storage architecture for large objects. In *Mass Storage Systems, 1991. Digest of Papers., Eleventh IEEE Symposium on*, pages 123–128, Oct 1991. doi: 10.1109/MASS.1991.160223. 10

[8] Luis-Felipe Cabrera and Darrell Don Earl Long. Swift: Using distributed disk striping to provide high I/O data rates. Technical report, Santa Cruz, CA, USA, 1991. 10

[9] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855840.1855861`. 17

[10] Mallikarjun Chadalapaka, Julian Satran, Kalman Meth, and David L. Black. Internet Small Computer System Interface (iSCSI) protocol. RFC 7143, April 2014. URL `http://tools.ietf.org/rfc/rfc7143.txt`. 2, 5, 6

[11] Chen. Analysis of write amplification for synchronized write in Linux ext3/ext4 filesystems, December 2013. URL `http://ilinuxkernel.com/?p=1477`. 33

[12] Peter Ming-Chien Chen and David Andrew Patterson. A new approach to I/O performance evaluation: Self-scaling I/O benchmarks, predicted I/O performance. *SIGMETRICS Perform. Eval. Rev.*, 21(1):1–12, June 1993. ISSN 0163-5999. doi: 10.1145/166962.166966. 10, 52

[13] Brian Frank Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. 49, 51

[14] Andy Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, May 2004. ISSN 1542-7730. doi: 10.1145/1005062.1005069. 9, 12

[15] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 481–494, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1705-4. doi: 10.1145/2398776.2398827. 2, 9, 22, 24, 49

[16] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 205–212, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1953-9. doi: 10.1145/2504730.2504762. 9, 49

[17] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing TCP's initial congestion window. *SIGCOMM Comput. Commun. Rev.*, 40(3):26–33, June 2010. ISSN 0146-4833. doi: 10.1145/1823844.1823848. 52

[18] Michael Eisler, Alex Chiu, and Lin Ling. RPCSEC_GSS protocol specification. RFC 2203, September 1997. URL `http://tools.ietf.org/rfc/rfc2203.txt`. 6

[19] Daniel Ellard and Margo Seltzer. NFS tricks and benchmarking traps. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1247340.1247356`. 5

[20] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1548–1557 vol.3, 2001. doi: 10.1109/INFCOM.2001.916651. 52

[21] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887. 7

[22] Roy Thomas Fielding, Jim Gettys, Jeffery C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. URL `http://tools.ietf.org/rfc/rfc2616.txt`. 7

[23] Christoph M. Gauger, Martin Köhn, Sebastian Gunreben, Detlef Sass, and Samuel Gil Perez. Modeling and performance evaluation of iSCSI storage area networks over TCP/IP-based MAN and WAN networks. In *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, pages 850–858 Vol. 2, Oct 2005. doi: 10.1109/ICBN.2005.1589695. 9, 52

[24] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the Internet. *Commun. ACM*, 55(1):57–65, January 2012. ISSN 0001-0782. doi: 10.1145/2063176.2063196. 12, 14

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. 9, 22, 24

[26] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400105. 12, 18

[27] Thomas J. Hacker, Brian D. Athey, and Brian Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10 pp–, April 2002. doi: 10.1109/IPDPS.2002.1015527. 50

[28] Robert Henschel, Stephen Simms, David Hancock, Scott Michael, Tom Johnson, Nathan Heald, Thomas William, Donald Berry, Matt Allen, Richard Knepper, Matthew Davy, Matthew Link, and Craig A. Stewart. Demonstrating Lustre over a 100Gbps wide area network of 3,500km. In

Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 6:1–6:8, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389005`. 10, 36

[29] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference on*, pages 18–27, April 2005. doi: 10.1109/MSST.2005.14. 9

[30] Dean Hildebrand, Lee Ward, and Peter Honeyman. Large files, small writes, and pNFS. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 116–124, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi: 10.1145/1183401.1183419. 9

[31] John Hayes Howard, Michael L. Kazar, Sherri G. Menees, David Arthur Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988. ISSN 0734-2071. doi: 10.1145/35037.35059. 9

[32] Junxian Huang, Feng Qian, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307658. 18, 22, 24, 50

[33] Institute of Electrical and Electronics Engineers (IEEE). IEEE 802.11™Wireless Local Area Networks. URL `http://www.ieee802.org/11/`. 1, 13, 36

[34] Van Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988. ISSN 0146-4833. doi: 10.1145/52325.52356. 18

[35] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling bufferbloat in 3G/4G networks. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 329–342, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1705-4. doi: 10.1145/2398776.2398810. 12, 14

[36] Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, Inc. (now NetApp, Inc.), 1997. 3, 43

[37] Thomas M. Kroeger and Darrell Don Earl Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the*

General Track: 2002 USENIX Annual Technical Conference, pages 105–118, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-09-X. URL `http://dl.acm.org/citation.cfm?id=647055.715905`. 9

[38] Ilja Livenson and Erwin Laure. Towards transparent integration of heterogeneous cloud storage platforms. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 27–34, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0704-8. doi: 10.1145/1996014.1996020. 10

[39] Robert Love. Kernel Korner – I/O Schedulers. *Linux Journal*, 118, Feb 2004. ISSN 1075-3583. URL `http://www.linuxjournal.com/article/6931`. 32

[40] Yingping Lu and David Hung-Chang Du. Performance study of iSCSI-based storage subsystems. *Communications Magazine, IEEE*, 41(8):76–82, Aug 2003. ISSN 0163-6804. doi: 10.1109/MCOM.2003.1222721. 18

[41] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-99-7. URL `https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou`. 33

[42] Peter M. Mell and Timothy Grance. SP 800-145. The NIST definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011. 2

[43] Michael Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8):84–90, Aug 2003. ISSN 0163-6804. doi: 10.1109/MCOM.2003.1222722. 2

[44] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043563. 12

[45] Kalman Z. Meth and Julian Satran. Design of the iSCSI protocol. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 116–122, April 2003. doi: 10.1109/MASS.2003.1194848. 2, 6

[46] James H. Morris, Mahadev Satyanarayanan, Michael Haden Conner, John Hayes Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. *Commun. ACM*, 29(3):184–201, March 1986. ISSN 0001-0782. doi: 10.1145/5666.5671. 9

[47] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001. ISSN 0163-5980. doi: 10.1145/502059.502052. 9

[48] John Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, October 1984. ISSN 0146-4833. doi: 10.1145/1024908.1024910. 52

[49] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter Ming-Chien Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL `http://dl.acm.org/citation.cfm?id=1298455.1298457`. 9

[50] Jukka K. Nurminen. Parallel connections and their effect on the battery consumption of a mobile phone. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5, Jan 2010. doi: 10.1109/CCNC.2010.5421769. 17

[51] Open Scalable File Systems, Inc. The Lustre file system. URL `http://lustre.opensfs.org/`. 10

[52] OpenStack Foundation. OpenStack, . URL `https://wiki.openstack.org/wiki/`. 3, 7

[53] OpenStack Foundation. OpenStack object storage – Swift, . URL `https://wiki.openstack.org/wiki/Swift`. 3, 5, 7

[54] Zhonghong Ou, Shichao Dong, Jiang Dong, Jukka K. Nurminen, Antti Ylä-Jääski, and Ren Wang. Characterize energy impact of concurrent network-intensive applications on mobile platforms. In *Proceedings of the Eighth ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '13, pages 23–28, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2366-6. doi: 10.1145/2505906.2505909. 1, 2, 17

[55] Zhonghong Ou, Jiang Dong, Shichao Dong, Jun Wu, Antti Ylä-Jääski, Hui Pan, Ren Wang, and Alexander W. Min. Utilize signal traces from others? A crowdsourcing perspective of energy saving in cellular data communication. *Mobile Computing, IEEE Transactions on*, PP(99):1–14, 2014. ISSN 1536-1233. doi: 10.1109/TMC.2014.2316517. 2, 17

[56] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, 2000. 2, 5, 50

[57] Steven L. Pratt and Dominique A. Heger. Workload dependent performance evaluation of the Linux 2.6 I/O schedulers. In *Linux Symposium*, pages 427–450, 2004. 32, 50

[58] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, and Prashant Shenoy. A performance comparison of NFS and iSCSI for IP-networked storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 101–114, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1096673.1096688`. 5, 9, 18, 39, 41

[59] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *USENIX Annual Technical Conference, General Track*, pages 41–54, 2000. 9, 22, 24

[60] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985. 1, 5

[61] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage over IP: When does hardware support help? In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 231–244, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1090694.1090723`. 9

[62] Mahadev Satyanarayanan, James Jay Kistler, Puneet Kumar, Maria E. Okasaki, Ellen Harriet Siegel, and David Cappers Steere. Coda: a highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, Apr 1990. ISSN 0018-9340. doi: 10.1109/12.54838. 9

[63] 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. URL `http://dl.acm.org/citation.cfm?id=1973333.1973349`. 10

[64] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855511.1855530`. 50

[65] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network File System (NFS) version 4 protocol. RFC 3530, April 2003. URL `http://tools.ietf.org/rfc/rfc3530.txt`. 2

[66] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: Workbench automation for server

benchmarking. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 241–254, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1404014.1404032`. 10, 52

[67] SUN Microsystems et al. FileBench. URL `http://sourceforge.net/projects/filebench/`. 3, 43

[68] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1991596.1991609`. 10

[69] The 3rd Generation Partnership Project (3GPP). High Speed Packet data Access, . URL `http://www.3gpp.org/hspa`. 1

[70] The 3rd Generation Partnership Project (3GPP). LTE – Long Term Evolution, . URL `http://www.3gpp.org/lte`. 1, 13

[71] The 3rd Generation Partnership Project (3GPP). LTE–Advanced, . URL `http://www.3gpp.org/lte-advanced`. 36

[72] Robert Thurlow. RPC: Remote procedure call protocol specification version 2. RFC 5531, May 2009. URL `http://tools.ietf.org/rfc/rfc5531.txt`. 5

[73] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles Philip Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008. ISSN 1553-3077. doi: 10.1145/1367829.1367831. 3, 10

[74] Cheng-Chia Wang and Yarsun Hsu. Wofs: A distributed network file system supporting fast data insertion and truncation. In *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, pages 43–50, May 2010. doi: 10.1109/SNAPI.2010.13. 9

[75] Sage A. Weil, Scott Alan Brandt, Ethan L. Miller, Darrell Don Earl Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL `http://dl.acm.org/citation.cfm?id=1298455.1298485`. 10

[76] Dimitrios Xinidis, Angelos Bilas, and Michail D. Flouris. Performance evaluation of commodity iSCSI-based storage systems. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference on*, pages 261–269, April 2005. doi: 10.1109/MSST.2005.23. 44

[77] Yongjian Zhang and Mike H. MacGregor. Tuning Open-iSCSI for operation over WAN links. In *Communication Networks and Services Research Conference (CNSR), 2011 Ninth Annual*, pages 85–92, May 2011. doi: 10.1109/CNSR.2011.21. 36

[78] Yu Zhang and Bharat Bhargava. Self-learning disk scheduling. *Knowledge and Data Engineering, IEEE Transactions on*, 21(1):50–65, Jan 2009. ISSN 1041-4347. doi: 10.1109/TKDE.2008.116. 50

[79] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ViewBox: Integrating local file systems with cloud storage services. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 119–132, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-08-9. URL `http://dl.acm.org/citation.cfm?id=2591305.2591317`. 10