

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

MapStoreFS : developing a file system on MapStore

Permalink

<https://escholarship.org/uc/item/2sx9s9n7>

Author

Hong, Jim

Publication Date

2009

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

MapStoreFS: Developing a File System on MapStore

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Jim Hong

Committee in charge:

Professor Amin Vahdat, Chair
Professor Stefan Savage
Professor Geoffrey M. Voelker

2009

Copyright
Jim Hong, 2009
All rights reserved.

The Thesis of Jim Hong is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

TABLE OF CONTENTS

	Signature Page	iii
	Table of Contents	iv
	List of Figures	vii
	List of Tables	viii
	Acknowledgments	ix
	Abstract	x
Chapter 1	Introduction	1
Chapter 2	Background	3
	2.1. File Systems	3
	2.1.1. General Components	3
	2.1.2. Unix	4
	2.1.3. ZFS	5
	2.2. Mace	7
	2.3. MAPSTORE	8
	2.3.1. Overview	8
	2.3.2. Append-Only Maps	8
	2.3.3. Atomic Operations	9
	2.3.4. Permissions	9
	2.3.5. API	9
	2.4. File System in Userspace	11
	2.4.1. Overview	11
	2.4.2. Path Traversal	12
	2.4.3. Write and Read Behavior	13
	2.4.4. Data Cache	13
Chapter 3	File System Design	15
	3.1. Files	15
	3.2. Inodes	17
	3.3. Directories	18
	3.4. Permissions	18
Chapter 4	Implementation	20
	4.1. Service Modules	20
	4.2. Map Naming Scheme	22
	4.3. Path Traversal	22

4.4. Metadata Cache	23
4.5. Asynchronous Writes	24
4.6. Write Buffer	25
4.7. Journal Operations	25
4.8. Operations	26
4.8.1. getattr(path)	26
4.8.2. access(path, mode)	27
4.8.3. readlink(path)	27
4.8.4. readdir(path)	27
4.8.5. mkdir(path, mode)	28
4.8.6. rmdir(path)	29
4.8.7. mknod(path, mode)	31
4.8.8. symlink(from, to)	31
4.8.9. unlink(path)	32
4.8.10. rename(from, to)	33
4.8.11. chmod(path, mode)	34
4.8.12. open(path, read, write)	35
4.8.13. write(path, mapAddr, buf, offset)	36
4.8.14. truncate(path, mapAddr, size)	36
4.8.15. read(path, mapAddr, offset, size)	36
4.9. Applying Logs to the Checkpoint	37
Chapter 5 Initial Design	40
Chapter 6 Performance Evaluations	42
6.1. Tuning Variables	42
6.1.1. Outstanding Writes	43
6.1.2. Applying Logs in Batches	44
6.1.3. Log Application Timer	45
6.1.4. Journal Timer	47
6.2. Micro-Benchmarks	47
6.2.1. Metadata Performance	49
6.2.2. Uncached Reads	50
6.2.3. Cached Reads	51
6.2.4. Writes	51
6.3. Macro-Benchmarks	53
6.3.1. PostMark	53
6.3.2. Compiling OpenSSH	55
6.3.3. Hypothetical OpenSSH Compilations	56
Chapter 7 Related Works	58
7.1. Log-Structured File System	58
7.2. Frangipani	59
7.3. Dynamo	60

	7.4. Sinfonia	60
	7.5. Cooperative File System	61
Chapter 8	Conclusion	63
	Bibliography	65

LIST OF FIGURES

Figure 2.1	Example of a Directory Structure	13
Figure 3.1	Example of a MAPSTOREFS Directory Structure	16
Figure 4.1	Diagram of MAPSTOREFS Architecture	21
Figure 4.2	Example for Path Traversal	23
Figure 4.3	Detailed Example of File Maps	37
Figure 6.1	Evaluation Setup	43
Figure 6.2	Outstanding Writes	44
Figure 6.3	Applying Logs in Batches	45
Figure 6.4	Write Log Timer	46
Figure 6.5	Journal Timer Performance	48
Figure 6.6	Uncached Read Performance	51
Figure 6.7	Cached Read Performance	52
Figure 6.8	Write Performance	53

LIST OF TABLES

Table 4.1	Overview of mkdir	28
Table 4.2	Overview of rmdir	30
Table 6.1	Metadata Performance	49
Table 6.2	PostMark Evaluation	54
Table 6.3	OpenSSH Compilation	55
Table 6.4	Hypothetical OpenSSH Compilation for MAPSTOREFS . . .	57

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Amin Vahdat, for his support, guidance, and encouragement throughout my graduate years. It has been a great honor and privilege to work with him.

I would like thank James Anderson, Alex Rasmussen, and Charles Killian for all their support in the development of MAPSTOREFS. James' work on MAPSTORE is the foundation of MAPSTOREFS. His support and patience provided me with the necessary tools to progress through my academic career. I thank Alex for his help in implementing MAPSTOREFS. He has been there since the beginning, and without him, I cannot imagine what I would do. I thank Charles for introducing me to the world of Mace. It was a lifesaver.

I further would like to thank Professor Geoffrey Voelker and Professor Stefan Savage, my thesis committee members, for their precious time and guidance.

Last but not least, I would like to thank Taurin Tan-atichat, Kaisen Lin, and Sebastian Becerra for their undying moral support. A morale boost is always appreciated.

ABSTRACT OF THE THESIS

MapStoreFS: Developing a File System on MapStore

by

Jim Hong

Master of Science in Computer Science

University of California, San Diego, 2009

Professor Amin Vahdat, Chair

Designing and implementing a highly reliable and available data storage layer service across multiple machines is no easy feat. In response to this impediment, MAPSTORE was developed to provide a storage layer designed to run on a network of data centers with the ability to withstand both transient and permanent failure of entire data centers. Developers can implement applications on top of MAPSTORE by accessing its data through uniquely-named maps, which map an arbitrary number of keys to values. MAPSTORE provides an API for constructing and manipulating its maps while providing strong data consistency guarantees.

The focus of this thesis is validating the expressiveness of MAPSTORE's map interface by implementing MAPSTOREFS, a fault-tolerant, high-performance, distributed file system on top of MAPSTORE. Previous work has shown that a mail server and an object store system are expressible by MAPSTORE's map interface, but only demonstrates a limited scope of potential applications. This thesis expands MAPSTORE's application space and complexity by detailing the development of MAPSTOREFS.

Chapter 1

Introduction

Data storage has become an ever-growing integral part of many businesses' infrastructure. Banks rely on having customer information readily available as well as consistently updated to appease customers. E-commerce businesses depend on providing reliable customer transactions for their merchandise while managing their inventory. The advent rise in streaming media, from music to high-definition video, has further forced the necessity of massive data storage solutions. The burden of storing user data has shifted from clients to web services because these services can supply availability, consistency, and reliability guarantees that cannot easily be sustained by a single client. As a result, many cloud computing services have emerged to tackle these issues.

Many companies invest in a number of storage solutions. Companies with the financial resources employ their machines across multiple data centers. Developing their own data management services, data can be replicated between machines within and across data centers to provide reliability and availability. The cost of this solution requires immense financial resources to deploy machines, to power, and to develop the data management system tailored to needs of the business. For a small e-commerce company, to scale to such a storage solution is not feasible. As a result, some businesses rely on outsourcing this development to other businesses, such as Amazon's S3 [2].

In light of growing demand for highly available and strongly consistent storage, MAPSTORE was developed to address these concerns. MAPSTORE is a storage

layer designed to run on a network of data centers. It is devised to withstand both transient and permanent failure of entire data centers and to provide strongly consistent access to its data. The basic unit of storage of MAPSTORE is a uniquely-named *map* that maps an arbitrary number of keys to values. By conforming to the MAPSTORE API, users can create, remove, and manipulate their data.

Maps provide a programming abstraction suitable for a wide range of applications. As a building block, maps can be constructed into a collection of records. They function similarly to the database abstraction of tables, a data storage abstraction that has supported countless services. MAPSTORE's map interface exposes a low-level data unit that enforces an essentially type-less key-value pair. Every key and value is simply a string of characters, which a programmer can abstract to higher-level data types. MAPSTORE's map interface is intended to be unrestrictive and simple to allow programmers free reign over their services.

In order for MAPSTORE to be a feasible data storage to users, its API must be expressible for a breadth of applications. Previous work has shown that a mail server and an object storage system can be built on top of MAPSTORE, but exposes only a limited scope of applications that MAPSTORE's API can convey. This thesis expands MAPSTORE's potential application space and complexity by introducing MAPSTOREFS, a fault-tolerant, high-performance, distributed file system.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of general file system design, MAPSTORE, and various resources used in the development of MAPSTOREFS. Chapter 3 discusses the high-level design of MAPSTOREFS. Chapter 4 covers the implementation details of several aspects of MAPSTOREFS. Chapter 5 discusses the initial design of MAPSTOREFS and reasons for its departure. Chapter 6 evaluates the performance of MAPSTOREFS. Chapter 7 examines related works. Finally, Chapter 8 concludes this thesis.

Chapter 2

Background

This chapter provides an overview of the design influences contributed by general file system designs, the Mace programming language, MAPSTORE, and the FUSE kernel module. Each in turn illustrates a stepping stone to the eventual design of MAPSTOREFS.

2.1 File Systems

To understand what is needed in a file system, a brief explanation of file system components as well as real implementations will be discussed in the following sections.

2.1.1 General Components

A file system is used for persistent storage of data in an organized manner. It maintains a unit of data called a *file*, which it allows processes to read and write to. When viewed by a user, a file's data is merely a sequence of bytes. Yet, the file system maintains a different representation of the data based on its design and the storage medium (e.g., a disk) where its data resides. As a result, the user does not need to be aware of how data is laid out on a storage medium. In addition, the file system controls how files are accessed and protected so that only authorized users may read or write to a certain set of files.

Associated with each file is its metadata, which supplies information about the file itself. For example, some typically attributes found in the metadata of file systems are access rights, ownership, size, and the last modification time of a file. Such attributes can be very useful to applications. For instance, recompilation of source code can be done if the last modification has changed since the last compilation.

To aid in the organization of files, file systems provide a structure called a *directory*. Directories store a collection of files as well as other directories. The ability to contain a directory within another directory allows a file system to create a hierarchy of directories, which permits files with the same name to coexist as long as they reside in different directories. Not all file systems support a hierarchy though. For example, some file systems may have a single directory where all files reside, which is considered a flat hierarchy.

As a consequence of a directory hierarchy, the retrieval of a file by name does not sufficiently provide enough information for a lookup since files may have the same name when stored in different directories. Thus, to retrieve a file, a path name must be submitted to the file system to know exactly the set of directories that it must traverse in order to locate the file. There are two types of path names that a file system support: an absolute and relative path name. An absolute path name lists a complete set of directories the file system must traverse, starting from the root directory to the file. As for a relative path name, the file system searches a path relative to a current working directory.

In its essence, a file system stores data in the form of files and directories while providing an interface to create, delete, write, and read files.

2.1.2 Unix

The UNIX file system is one of the most influential systems ever created. Therefore, a close examination of its design is discussed. Specifically, the Fast File System (FFS) [9] will be dissected.

FFS was designed for a disk medium. A special region of the disk is reserved

for the inode table, which maps an inode number to its corresponding inode. An inode is essentially a file's metadata whereas the inode number is its unique identifier. The inode contains such information as the type, size, and permissions of the file. In addition, an inode contains a fixed-size array of disk addresses to a file's data blocks.

Directories in the UNIX are files themselves. The directory file contains a table that maps the names of directory entries to their corresponding inode numbers. In order to locate an object within a directory, the corresponding inode number of its directory entry is used to index the inode table region of the disk.

A known region of the disk is reserved for the superblock of the file system. This block contains critical information about the file system, such as the file system type, the file system block size, and the inode number of the root directory. Since the superblock contains vital information about the file system, corruption of its data must be prevented. As a result, FFS replicates the superblock in multiple regions of the disk.

One of the key designs of FFS is dividing the disk into cylinder groups. Each group contains a copy of the superblock, as mention before to increase the reliability of the system. Furthermore, a cylinder group includes inodes, a bitmap of free blocks within the group, and data blocks. To further prevent the corruption of the superblock, each cylinder group stores the superblock at various offsets from the beginning of the group. The reason for this redundancy pattern is to prevent the failure in the alternate design: all superblocks are placed in the same location from the beginning of a cylinder group. In this situation, all superblocks are located on the same platter of a disk. If this single platter is damaged, essentially all replicated superblocks will be corrupted. Therefore, by replicating the superblocks at various offsets, a replica will likely persists in case of a failure.

2.1.3 ZFS

Sun Microsystems' Zettabyte File System (ZFS) [10] has gained popularity in the recent years. One of its distinguishing points is that it is a 128-bit file system, a capacity that will unlikely be a storage bottleneck for quite some time. It provides

strong consistency guarantees for its data by utilizing various techniques.

For instance, ZFS uses a pooled storage model where multiple storage devices are aggregated together as a single unit. In comparison with the volume abstraction used by many file systems, such as FFS, the multiple devices in the pooled storage act as a single volume. However, ZFS does not allocate a fixed-size file system on top of its pooled storage, but only occupies the space that it needs. ZFS never has to preallocate its system size in a form of partitions or volumes. As a result, multiple ZFS file systems can be built from a common pooled storage without prior planning.

The pooled storage model provides many benefits. One of the main benefits of this model is that the file system can easily scale in storage. If more storage is required, a storage device can simply be added to the pooled storage without reconfiguring the file system. Another advantage is device utilization. In the volume abstraction, a file system allocates a fixed-size partition for its use. Unused space allocated by this system cannot be used by another file system; thus, space is idly wasted. Since ZFS consumes the amount of space it needs from the pooled storage, other ZFS systems are free to utilize all storage devices' unused space. Lastly, a pooled storage enables a filesystem to use the full aggregate bandwidth of all storage devices since none of the devices are fully obligated to a single file system.

ZFS provides strong data consistency by employing transactional writes. In order to achieve such transactional guarantees, it utilizes a copy-on-write scheme. When writing to an existing data block, instead of immediately overwriting the block, a new data block is created and written to. Once the write has completed, the existing data block's data pointer is adjusted to reference the new data block. This step commits the write's changes to the filesystem, as it would in a typical transaction.

In addition to transactional writes, ZFS ensures further consistency by storing checksums with data pointers. Upon a write completion, a checksum of the data block is stored alongside the data pointer. On each read of a data block, the checksum is calculated and compared to the stored checksum to ensure that the data has not been corrupted. If its checksum does not match, then ZFS will attempt to fix the data

block. When storage devices are in a mirrored configuration, ZFS would merely use the mirrored copy to fix the corrupted data block.

2.2 Mace

Since MAPSTORE and MAPSTOREFS were developed using Mace, a brief discussion of Mace's features and how it functions will provide a better understanding of both services' implementations. Mace [6] is a C++ language extension and source-to-source compiler designed for the development of distributed systems. The goal of Mace is to be an expressive high-level language for distributive systems while maintaining the performance of low-level implementations.

Systems developed in Mace are a layering of event-driven services that abstract a state-transition system and interact with one another through message transmissions. A service implements a defined interface that lists its functionality and the requirements that higher-level services must support. Events in Mace are abstracted as transition methods associated with a service. More specifically, these transitions define the functionality and the requirements of a service.

One of the main benefits in developing in Mace is its support tools to evaluate a system's correctness and performance. Mace services are capable of automatically generating logs of events at each node, capturing the order and timing of events and state information. Using the generated logs, MDB, a replay debugger, allows a developer to further analyze the system by stepping through a service's execution. In addition to a replay debugger, Mace also supplies a model checker, MaceMC [7]. MaceMC attempts to exhaustively search the execution space of a service to find subtle implementation bugs, such as safety and liveness violations. The model checker also produces logs in which can be fed into MDB for further analysis.

Due to Mace's expressive nature towards distributed systems development and its immense analysis tools, MAPSTOREFS was implemented in Mace.

2.3 MAPSTORE

MAPSTOREFS inherits multiple of its attributes from its underlying storage layer service, MAPSTORE. Therefore, to understand how MAPSTOREFS functions, an understanding of MAPSTORE is necessary. The following sections will discuss a general overview of MAPSTORE and its operations.

2.3.1 Overview

MAPSTORE is a highly available, strongly consistent storage layer designed to run on a network of data centers. It is designed to survive both transient and permanent failure of entire data centers and to provide strongly consistent access to its data. The basic unit of storage in MAPSTORE is a uniquely-named *map* that maps an arbitrary number of keys to values. A value has a bounded size, which MAPSTORE currently limits to 64 kilobytes. By default, all key-value pairs are sorted by key. In order to maintain high availability of its data, MAPSTORE replicates its data across multiple data centers as well as within them. Data centers achieve consensus on the order of writes to a map by using the Paxos [8] protocol. Because MAPSTORE provides high availability and strong consistency of its data, MAPSTOREFS inherits these attributes for its data as well.

2.3.2 Append-Only Maps

MAPSTORE also supports an append-only map that is heavily used by MAPSTOREFS. Fundamentally, an append-only map acts as a queue. The only operation permitted for adding new values to the map is `Append`, which specifies the map address and the value to be written. Upon receiving the `Append` request, MAPSTORE will form consensus between its servers to determine a key to associate with the request's value. The chosen key will appropriately order the request as the last key-value pair within the map. Once determined, the chosen key and request's value will be inserted into the map. As an effect, the consensus on keys creates an ordering of writes to the map.

2.3.3 Atomic Operations

MAPSTORE features two operations, `CompareAndSet` and `MultiOp`, that atomically update and/or remove key-value pairs within a single map. These operations provide a set of key-value pairs that must match the current state of a map's key-value pairs. Only if these comparisons match will all specified writes and removals be committed to the map. More details on these operations can be found in Section 2.3.5.

2.3.4 Permissions

MAPSTORE enforces access control for individual maps by using a combination of access control lists and capabilities. When creating a map, a user specifies three sets of credentials that indicates who is allowed to read and write to the map as well as who has ownership privileges. When a user wishes to gain access to a map, the user must provide his or her credentials to MAPSTORE. In turn, MAPSTORE will compare the user's credential to the map's credential set and return a capability *MapDescriptor* to the client. The *MapDescriptor* lists the map address, indicates if MAPSTORE has granted read and/or write access, and contains an expiration time and a signature from the MAPSTORE server that created the *MapDescriptor*. Every time a client requests an operation on the map to MAPSTORE, it will send its *MapDescriptor* along to prove its capabilities. The *MapDescriptor* contains enough information that any MAPSTORE can verify its authenticity upon each request.

2.3.5 API

MAPSTORE provides an associated client service to communicate to its servers. Only the relevant subset, in respect to MAPSTOREFS, of the client service's API will be discussed. All operations contain an *id* parameter that MAPSTORE returns in every operation response in order to match requests. For brevity, the *id* parameter along with parameters not used by MAPSTOREFS will not be listed in the operation details.

- **Alloc**(*addr, readers, writers, owners*): Creates a map with the name *addr*. The last three arguments provide the credential sets of the map for readers, writers, and owners.
- **Open**(*addr, read, write*): Requests for a *MapDescriptor* from the map named by *addr*. The *read* and *write* indicate the access rights that the client is requesting. It must be noted that the MAPSTORE client service caches the *MapDescriptor* and periodically requests a new *MapDescriptor* on the client's behalf when it nears expiration. In essence, once a map is opened, any open request afterwards returns the cached *MapDescriptor* and thus avoids contacting MAPSTORE.
- **Free**(*addr*): Deallocates a map named by *addr*.
- **SetPermissions**(*addr, readers, writers, owners*): Sets the permissions of the map named by *addr* using the credential sets *readers*, *writers*, and *owners*.
- **Read**(*addr, key*): Returns the corresponding value of *key* from the map named by *addr*.
- **ReadList**(*addr, keys*): Returns the set of values specified by the *keys* list from the map named by *addr*.
- **ReadRangeMap**(*addr, lowerbound, upperbound, limit*): Returns the set of values specified by a key range from the map named by *addr*. Map values are sorted by key so a key range can be requested by using a *lowerbound* key and an *upperbound* key. If both keys are empty, all key-value pairs from the map will be returned. The integer value associated with the *limit* parameter restricts the maximum number of key-value pairs that will be returned. Since key-value pairs are sorted, the first number of pairs to *limit* will be returned.
- **Write**(*addr, key, value*): Writes the *key-value* pair to the map named by *addr*.
- **Append**(*addr, value*): Creates a new key designated as the last key and writes *value* to the map named by *addr*.

- **Remove**(*addr, key*): Removes the key-value pair designated by *key* from the map named by *addr*.
- **CompareAndSet**(*addr, key, compare, value*): Writes *value* to *key* contained in the map named by *addr* on the condition that the key's previous value equals to *compare*. If the condition fails, the write is not committed.
- **MultiOp**(*addr, comparisons, writes, removes*): Writes and removes key-value pairs from the map named by *addr* on a conditional basis. *comparisons* is a map that lists key-value pairs that the map must currently contain in order for the writes and removes to be committed to the map. If any key-value pair in *comparison* does not match the current state of the map, no writes or removes may proceed. *writes* is a map of key-value pairs that will be written to the map. *removes* is a list of keys that will be removed from the map.

2.4 File System in Userspace

MAPSTOREFS is built using the File System in Userspace (FUSE) [5] kernel module, version 2.7.4. FUSE enforces multiple requirements in order for a service to utilize it and as a result, influenced many design decisions of MAPSTOREFS.

The following sections describe FUSE and its peculiarities.

2.4.1 Overview

FUSE is a kernel module that simplifies the creation of a file system by a non-privileged user. A user defines his or her file system by implementing various functions required by the FUSE API. Once a user-defined file system is mounted, all operation calls received by the local file system will be redirected to the FUSE module. From there, the FUSE module passes operation calls along to the user's defined FUSE API functions where they are handled accordingly to the user's specification.

Creating a file system is not a trivial task. Integrating a new file system and managing system calls to a preexisting kernel would require quite some effort along

with an increase risk of introducing bugs. FUSE employs many benefits in reducing the entry cost of implementing and deploying such file systems. As stated before, the user does not need to modify the kernel in order to create his or her file system. By abstracting file system operations to various function interfaces, FUSE tremendously eases the effort in the implementation process. Furthermore, once a user's file system is defined, it can be easily ported to any operating system as long as the FUSE module has been implemented for it. FUSE has already been implemented for various operating systems, such as Linux and Mac OS X.

Although there is a penalty for the communication indirection that FUSE requires, it is not the local system calls that are the bottleneck to the system, but the network. The intended deployment of MAPSTORE requires machines across multiple data centers. Consensus between data centers consequently involves multiple round-trips, which is a inevitable delay that MAPSTOREFS must endure. Compared to a local file system data access with its hard drive, MAPSTOREFS's data access is vastly slower. The delay introduced by the context switching between FUSE and kernel is miniscule relative to the MAPSTORE communication time. Reducing this context switching delay would require developing within the kernel, which undoubtedly complicates the development process for small performance gains. Therefore, we opted for FUSE in MAPSTOREFS's development to reduce the time and effort to demonstrate the expressiveness of MAPSTORE's map interface to support a file system.

2.4.2 Path Traversal

Path names are the logical identifiers for objects in a file system. They list the directories that must be traversed to retrieve requested objects. When a user submits a file system operation request, the FUSE kernel module administers the path traversal by calling `getattr`, which retrieves the inode data from an object, on all directories starting from the root leading to the object and on the object itself. By systemically acquiring inodes one by one, FUSE can determine the existence of directories along the path to check its validity. In addition, FUSE expands symbolic links found by calling

`readlink`, which it then recursively calls `getattr` on to continue the traversal.

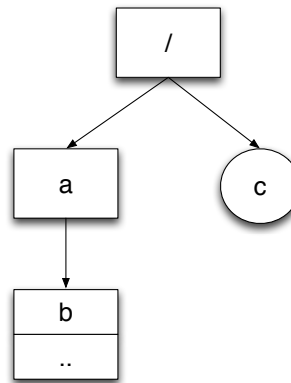


Figure 2.1: Example of a Directory Structure

For example, consider the file system layout in Figure 2.1 where rectangles, circles, and double nested rectangles represent directories, files, and symbolic links, respectively. Given a request path of `/a/b/c`, FUSE requests a `getattr` on the root directory `/`, then on directory `/a`, and followed by the symbolic link `/a/b`. Since the symbolic link references a relative path of `..`, the path resolves to `/c` by calling `readlink`. To complete the traversal, `getattr` is called on `/c`.

2.4.3 Write and Read Behavior

One of the limitations of FUSE API is its restrictive write request size of four kilobytes. Along with the synchronous behavior of write requests, consecutive writes must block till a previous request completes. Similar to writes, FUSE limits a read request size to 128 kilobytes. Fortunately, FUSE supports a read ahead scheme that parallelizes read request to avoid unnecessarily blocking. These restraints drastically dictated the design of MAPSTOREFS.

2.4.4 Data Cache

FUSE provides an option to cache file data based on modification times. The data remains in cache as long as the modification time has not changed since the last

open. Otherwise, the data will be flushed. FUSE only caches data during read operations. Consequently, a write operation has no effect on the data cache so a subsequent read after a write misses the data cache. MAPSTOREFS enables this option nonetheless.

Chapter 3

File System Design

Before delving into the details of file system operations, our techniques to abstract the common components of a file system must be understood. The following sections examine MAPSTOREFS's file, inode, and directory abstractions. As a guide, Figure 3.1 represents a simple directory structure in MAPSTOREFS to illustrate the various file system components.

3.1 Files

Files in MAPSTOREFS are abstracted as two MAPSTORE maps, as presented in Figure 3.1 as the two upper right maps. A file could not simply be defined as a single key-value pair in a map because values in MAPSTORE have a maximum size, currently 64 kilobytes.

The first of the two maps is considered the checkpoint map. Similar to how UNIX divides a storage device into fixed size blocks, a single block in MAPSTOREFS is defined as a key-value pair. All values entries in the checkpoint map are of fixed size to represent the block. As for the keys, they represent the block offset of the file. By itself, the checkpoint map represents the current snapshot of the file, which may not be the most recent representation of the file.

The additional second map is what completes the file abstraction as a whole. This map is called the log map because it maintains a log of all write and truncate oper-

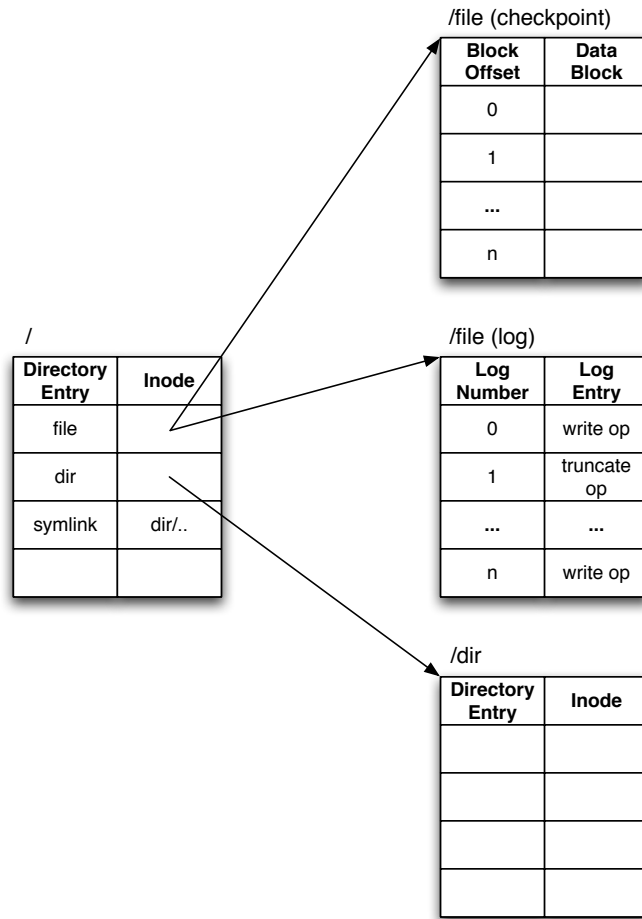


Figure 3.1: Example of a MAPSTOREFS Directory Structure

ations in the order they were submitted to MAPSTORE for its corresponding checkpoint map. The only MAPSTORE operation permitted on the log map is the MAPSTORE Append. Since Append maintains an order by producing nondecreasing key values, file operations logs are sorted correctly by keys. To retrieve the most recent view of the file, the file system retrieves all logs from the log map and applies them to its corresponding checkpoint and inode.

The effects of an append-only map for committing writes are twofold. Firstly, the write operation only requires one MAPSTORE operation, Append, making writes very fast at the expense of reads. Since a read request must apply all logs to present the current state of a file, the read processing time grows with the number of logs. To

reduce the number of logs present in the log map, logs must be permanently applied to its corresponding checkpoint map and removed from the log map. As long as the logs are purged in a timely manner, reads should only be marginally penalized. More details on this log application process will be discussed in Section 4.9. Secondly, an append-only map serializes requests from multiple writers, simplifying write contention logic in the system. As a result, MAPSTOREFS's write operations provide an efficient and consistent update to file data.

One may question why a file was abstracted to two maps instead of one. The original design of the MAPSTOREFS used a single map to represent a file, but due to the complexity in maintaining a file's metadata consistent while providing acceptable write performance, the current implementation was chosen. Further details on how the single map file abstraction was designed and its limitation can be found in Chapter 5.

3.2 Inodes

Unlike UNIX, MAPSTOREFS stores inodes in directory maps rather than in a single designated region. More specifically, the inodes of files, directories, and symbolic links reside in its parent directory map. As an exception, the root directory has no parent directory so its inode resides in its own map. Similar to UNIX's POSIX *stat* structure, a MAPSTOREFS inode contains the following attributes:

- **Mode:** Designates if the inode corresponds to a file, a directory, or a symbolic link.
- **Permissions:** The basic UNIX permissions designating the read, write, and executable rights for a user and others. There are no rights specified for groups since MAPSTORE permissions do not have an equivalent permission class. Further details about permissions is considered in Section 3.4.
- **Map Address:** Similar to UNIX's data pointer, this attribute contains the MAPSTORE map address of the corresponding file or directory. Symbolic links do

not have a corresponding MAPSTORE map because they are only embodied as an inode.

- Owner: The credential of the owner of the file, directory, or symbolic link.
- Modification Time: The timestamp of the last modification.
- Size: Size of the corresponding file. Directories and symbolic links have no size.

3.3 Directories

A directory in MAPSTOREFS is merely a MAPSTORE map, as portrayed in Figure 3.1 as the left and lower right maps. The contents of a directory map are the inodes of files, directories, and symbolic links contained within the directory. No actual data content is stored in directories since file content is stored in separate maps, as explained in Section 3.1.

In addition, a special key-value pair is reserved in the directory for a journal. A journal details a file system operation that requires atomicity across multiple maps and affects metadata in the residing directory. Further explanation of journals will be found in Section 4.7.

3.4 Permissions

As previously mentioned in Section 2.3.4, each MAPSTORE map associates sets of credentials that provides access control. Translating MAPSTORE permissions to basic UNIX permission for the file system is a trivial task for user and other access. If a user credential is listed in the read credential set, then the user has read access. In terms of other access, if the read set is empty, MAPSTORE allows all users read access. The same scheme applies to the write credential set as well. Furthermore, in order to deny access in either read or write set, MAPSTORE provides a special credential that prevents access if contained in a credential set. Unfortunately, the credential system of MAPSTORE does not equate well into the group access notion of UNIX. In order to support

group access in MAPSTOREFS, each user within the group must be supplied with the same credentials. To MAPSTORE, all users within the group would be identified as the same entity. Therefore, removing individual users from a group would be impossible since members are unidentifiable. MAPSTOREFS itself would have to implement its own access control above MAPSTORE. Unfortunately, we did not deem it necessary at the time to support group access control.

Chapter 4

Implementation

This chapter presents how MAPSTOREFS implements its file system responsibilities along with optimization details.

4.1 Service Modules

MAPSTOREFS divides the responsibility of the system into various service modules: a FUSE application, a *file system wrapper* Mace service, and many small *file system operation* Mace services. Figure 4.1 presents the hierarchy of the services and their communication points (represented by bidirectional arrows).

The FUSE module is a C++ application that handles the mounting of the file system and is the communication point with the FUSE kernel module. When a user requests an operation to the file system, the request dispatches to the FUSE application where it passes along to the next service module, the file system wrapper service.

The file system wrapper service is a Mace module that preps the file system request arguments as well as handles the metadata cache. Because FUSE operation requests designate objects by path names, path names need to be translated into their corresponding map address equivalent by the wrapper service. More details on this conversion process can be found in Section 4.3. Once the map addresses are known, the wrapper service hands off the operation request to one of the several file system operation services, which process the operation in MAPSTORE context. There are only

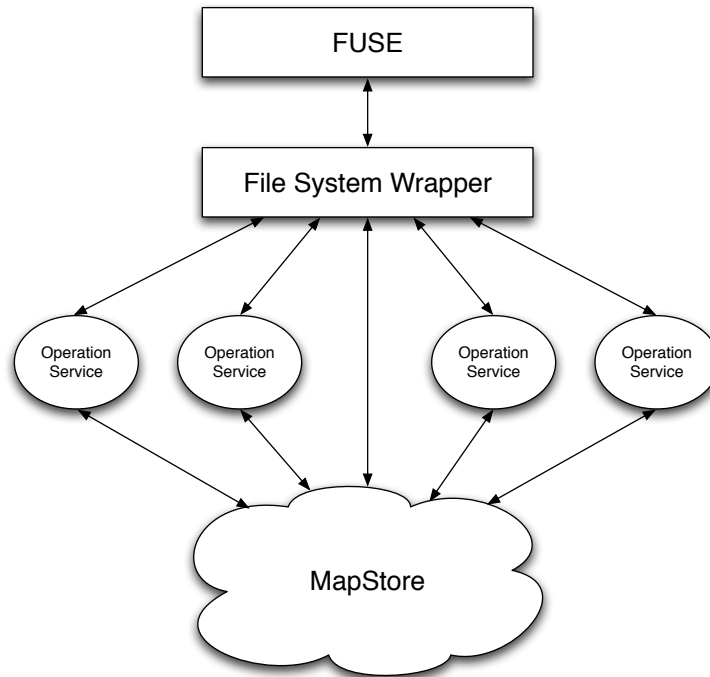


Figure 4.1: Diagram of MAPSTOREFS Architecture

three operations that the wrapper service manages on its own, which are `getattr`, `access`, and `readlink`. These operations only retrieve metadata information from inodes and are easily absorbed by the metadata cache. Furthermore, these operations simply formulate into one or two MAPSTORE operations that does not justify a separate service module; therefore, the wrapper service was the most logically module for the responsibility. The last function of the wrapper service is overseeing journal replays, which will be discussed in Section 4.7.

The file system operation services manage the bulk of the file system's functionality. Each service handles a single operation and receives dispatches from the wrapper service. Processing and error handling concerning a single operation are isolated into their corresponding services to simplify the implementation. Once a request has been attended to, the operation service informs the wrapper service, which in turn will notify the FUSE application of the result.

4.2 Map Naming Scheme

As previously mentioned, MAPSTORE maps are uniquely named. When creating or mounting MAPSTOREFS, a user must specify a unique name for the file system. This name is the map address of the file system's root directory.

File and directory (excluding the root directory) map addresses are derived by the concatenation of the file system name, delimited by a special character (a colon in the current implementation), and a 64-bit timestamp of creation. To reduce the likelihood of name collisions between multiple users creating files at the same time, a random 7-bit integer is attached to the higher bits of the timestamp. To completely avoid name collisions, the file system could employ a map that maintains a listing of all currently used names, similar to a free block map. Due to the additional complexity and an increase of a round-trip delay to file system operations, we did not choose this solution.

4.3 Path Traversal

Path traversal in the context of MAPSTORE map requires multiple `Open` and `Read` requests. For instance, consider Figure 4.2 and traversing path `/a/b/c`. The traversal begins by opening the root directory and reading the inode of directory `a`. Since an inode contains the address of a file or a directory, `a`'s inode provides `a`'s map address to open. Once opened, another read request is submitted to MAPSTORE to retrieve `b`'s inode. The same procedure repeats to obtain `c`'s inode. This example illustrates the recursive nature of `Open` and `Read` requests in order to resolve a path in MAPSTOREFS.

As previously mentioned in Section 2.4.2, FUSE recursively calls `getattr` on all directories along a path during traversal. Each `getattr` operation will independently call `Open` and `Read` requests from the root directory to its given sub-path. Examining Figure 4.2 again for path `/a/b/c`, FUSE submits four `getattr` requests for the following paths: `/`, `/a`, `/a/b`, and `/a/b/c`. As a result, MAPSTOREFS would read the root directory four times, directory `/a` three times, directory `/a/b` two times, and directory `/a/b/c` once. Consequently, there are redundant MAPSTORE requests that should be

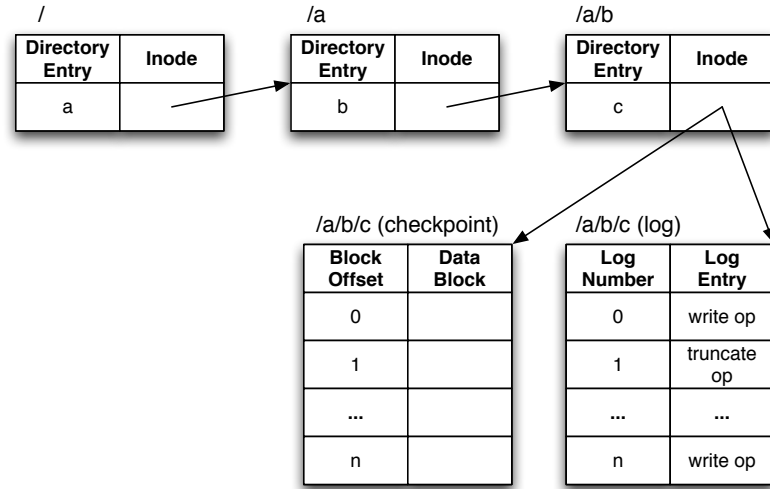


Figure 4.2: Example for Path Traversal

logically group into a single request. MAPSTOREFS avoids this wasted round-trips to MAPSTORE by introducing a metadata cache, which is presented in the next section.

4.4 Metadata Cache

The metadata cache compensates the expensive nature of inode data retrieval in MAPSTOREFS. Every time a file system operation retrieves or updates an inode, MAPSTOREFS caches it locally. Only when a request is unable to find the associated values in MAPSTORE or upon a deletion are inodes removed from the cache. In the scenario that an inode cannot be found in the cache, MAPSTOREFS reads all entries in the parent directory where the inode resides and caches all returned inodes. Because FUSE requests `getattr` operations for all directories from the root to the designated object during path traversal, parent directories are cached before requesting the child directory. Effectively, the redundant MAPSTORE calls in the FUSE's path traversal example described in Section 4.3 reduces to only one read operation for each `getattr` requests. Furthermore, all operations that involve read-only access to inode data are absorbed by the metadata cache.

4.5 Asynchronous Writes

In the original design of MAPSTOREFS, all write requests were handled synchronously. Unfortunately, FUSE's restrictive four kilobyte write limit and the network reliant nature of the system wielded an underperforming system. Synchronizing writes forces a user to wait for the return response from MAPSTORE before continuing to the next write operation, vastly underutilizing the network bandwidth. Consequently, writes that do not overlap in their byte ranges for a file are constrained to block unnecessarily.

In order to alleviate unwarranted blocking, MAPSTOREFS submits writes to MAPSTORE asynchronously when their file byte ranges do not overlap. When a user requests a write operation, MAPSTOREFS passes along the request to MAPSTORE, immediately updates the file's corresponding inode cache entry, and notifies the client of the write's completion. As a result, the user essentially avoids any network latency penalty. Furthermore, by parallelizing writes to MAPSTORE, MAPSTOREFS effectively utilizes the available bandwidth of the network to increase the overall throughput of the system.

It must be noted that all writes are not asynchronous in MAPSTOREFS. When submitting write requests asynchronously to MAPSTORE, variable network latencies can reorder their arrival to MAPSTORE. Thus, multiple write requests committed by MAPSTORE may not be ordered in the same fashion as the user submitted them. If the operations are not dependent on an ordering, then MAPSTOREFS safely handles requests asynchronously. Yet, if a write operation is dependent on an outstanding operation, the write must block till it completes. Write and truncate requests must block also if they are requested after a read or a truncate operation to maintain data consistency. MAPSTOREFS abides to these dependencies and applies writes synchronously in such scenarios.

One last behavior that should be mentioned about write requests is that MAPSTOREFS limits the number of outstanding write requests. After the threshold has been reached, write requests will block till outstanding requests complete. The reason for

bounding outstanding write requests is to prevent the trampling of MAPSTORE with an unbounded number of requests, which would deteriorate its responsiveness.

4.6 Write Buffer

To further mitigate FUSE's write size limit of four kilobytes, MAPSTOREFS buffers write requests up to 64 kilobytes before submitting them to MAPSTORE. A larger bound for the write buffer could not be chosen since MAPSTORE's maximum value size is restricted to 64 kilobytes and a write request only adds a single key-value pair to the file log map. The result of write buffer reduces the number of requests to MAPSTORE by 16. As an effect, it allows MAPSTOREFS to submit more write data to MAPSTORE before blocking due to the bounding of outstanding writes.

There are two conditions in which the write buffer flushes its data to MAPSTORE. The typical situation occurs when the buffer exceeds its storage limit and therefore flushes its contents to accommodate new data. The second condition, MAPSTOREFS exploits the FUSE supported API operation, *flush*. FUSE always requests a *flush* operation upon a *close* on a file descriptor. As such, MAPSTOREFS flushes its write buffer to MAPSTORE.

4.7 Journal Operations

File and directory creation and deletion involve two different maps, typically the parent directory map and the map to be created or deleted. In order to provide strong consistency for these operations, modifications across maps must be done atomically. Unfortunately, MAPSTORE only provides atomicity within a single map, so MAPSTOREFS implements a journal mechanism to simulate atomicity across maps.

Before an operation proceeds with its modifications, it writes a journal at all directory maps involved. The journal details all the necessary information for the operation to be replayed in case it fails to complete. Since every directory map only has a single journal entry reserved, an operation only writes to the journal key-value if it is

empty. If the journal entry is not empty, the resident journal must be replayed before placing the new journal. Once all journals are written, the operation may commence. After it completes its changes, all journal are removed to indicate that the operation has been committed.

4.8 Operations

In this section, the several file system operations required by the FUSE API are described in detail on how they coincide with MAPSTORE operations. It should be noted when examining a journal operation, it is implied when it attempts to write a journal with a `MultiOp`, its comparison map contains an empty value for the journal key. This check ensures that another journal operation is not in progress. If the `MultiOp` fails due to a resident journal, the operation stops and returns the resident journal to the file system wrapper service. Once in the hands of the wrapper service, the journal is given to the file system operation service that is capable of handling the replay. After the replay has completed, the wrapper service notifies the pending operation to reattempt its journal operation.

4.8.1 `getattr(path)`

The `getattr` operation retrieves the inode of the given object designated by *path*. MAPSTOREFS attempts to retrieve the inode by probing its metadata cache. In the case where the inode is not found, the requested object's parent directory is looked up in the cache. Since FUSE calls `getattr` on all directories from the root directory to the object during path resolution, it is safely assumed that the parent directory is already cached. Once the parent directory inode is retrieved from cache, `readdir` is called on the parent directory. All of the inode entries returned by `readdir` is cached, and the requested inode is returned.

As an optimization, if `readdir` has been issued on a parent directory in the last second and the requested inode is not found in the metadata cache, then the file

system notifies the client that the inode does not exist. The effect of this optimization is that MAPSTOREFS avoids a `readdir` request to MAPSTORE. We sacrifice some consistency for a second since it is very unlikely that a new inode will be created by another client within the last time that the parent directory was checked. This policy was adapted to MAPSTOREFS in discovering the poor performance encountered during the make phase of openssh. When make starts its compilation, it checks multiple library files that do not exist. Unfortunately, these checks translates to multiple unnecessary `readdir` requests to MAPSTORE to verify their non-existence, crippling the make time. Since MAPSTOREFS already caches existing inodes, it seemed justifiable to absorb some non-existent inode lookups into the metadata cache as well.

4.8.2 `access(path, mode)`

The `access` operation determines if a user has access rights defined by *mode* for the object specified by *path*. It is assumed that the requested object's inode is already cached due to the path traversal behavior of FUSE. Using the object's inode, the credentials of the user is compared with the object's credential set. `access` returns the result of this comparison.

4.8.3 `readlink(path)`

The `readlink` operation returns a symbolic link's target. Similar to the `access` operation, it is assumed that the requested symbolic link's inode has already been cached. `readlink` merely looks up the symbolic link and returns the target.

4.8.4 `readdir(path)`

The `readdir` operation returns all entries of the directory specified by *path*. `readdir` opens the directory's map address and calls `ReadRangeMap` on the map to retrieve all its key-value pairs. Afterwards, it caches and returns these entries to the user.

Table 4.1: Overview of mkdir

Step Number	MAPSTORE Operation	Parameter Details
1	Open (<i>addr, read, write</i>)	addr : parent directory map of <i>path</i> read : true write : true
2	MultiOp (<i>addr, comparisons, writes, removes</i>)	addr : parent directory map of <i>path</i> comparisons : (journal, empty value), (directory name, empty value) writes : (journal, mkdir journal) removes : empty
3	Alloc (<i>addr, readers, writers, owners</i>)	addr : map address of new directory readers : credentials from <i>mode</i> writers : credentials from <i>mode</i> owners : credentials from <i>mode</i>
4	MultiOp (<i>addr, comparisons, writes, removes</i>)	addr : parent directory map of <i>path</i> comparisons : (journal, written mkdir journal) writes : (journal, empty string), (directory name, new inode) removes : empty

4.8.5 mkdir(path, mode)

The `mkdir` operation creates a directory at the given *path* and initializes its permissions based on the *mode* value. An overview of `mkdir` operation logic can be seen in Table 4.1. Further details on each step of operation are explained by the following:

1. Open the parent directory with read and write access.
2. `MultiOp` at the parent directory to journal the `mkdir` operation. The comparison map ensures that the new directory's name is not in use. The write map

includes the journal of the `mkdir` operation, which specifies the new directory's name, the map address that will be allocated, and the credentials that equate to the new directory's permissions. If the `MultiOp` fails on the key of new directory's name because it already exists, the `mkdir` operation returns a status value indicating that fact.

3. `Allocate` the map specified by the journal for the new directory.
4. `MultiOp` at the parent directory to clear the journal and write the inode of the new directory. The comparisons contain the journal written at the beginning the `mkdir` operation to ensure that another client did not replay the current `mkdir` operation. If the comparison fails, the operation must have completed by another user. Therefore, the `mkdir` operation is considered a success as long as it receives a response from the `MultiOp`.

4.8.6 `rmdir(path)`

The `rmdir` operation removes a directory specified by the *path* value. An overview of the operation can be seen in Table 4.2 and is handled by the following `MAPSTORE` operations:

1. `Open` the parent directory map and the target directory map that will be removed with read and write access.
2. `MultiOp` at the parent directory and the target directory to journal operation. The journal specified at the parent directory consists of target directory's name, the map address of the target directory, and the *path*. As for the target directory's journal, it consists of target directory's name, the parent directory's map address, and the *path*. If the inode key fails its comparison at the parent directory, the inode is updated to the current value and the `MultiOp` is reattempted.
3. `ReadRangeMap` for all keys present in the target directory. A directory can only be deleted if the directory is empty. The directory is considered empty if it

Table 4.2: Overview of rmdir

Step Number	MAPSTORE Operation	Parameter Details
1	Open (<i>parent_addr</i> , <i>read</i> , <i>write</i>) Open (<i>target_addr</i> , <i>read</i> , <i>write</i>)	parent_addr : parent directory map of <i>path</i> target_addr : directory map that will be removed read : true write : true
2	MultiOp (<i>parent_addr</i> , <i>parent_comparisons</i> , <i>parent_writes</i> , <i>removes</i>) MultiOp (<i>target_addr</i> , <i>target_comparisons</i> , <i>target_writes</i> , <i>removes</i>)	parent_addr : parent directory map of <i>path</i> target_addr : directory map that will be removed parent_comparisons : (journal, empty string), (target directory, current inode) target_comparisons : (journal, empty string) parent_writes : (journal, rmdir journal) target_writes : (journal, rmdir journal) removes : empty
3	ReadRangeMap (<i>target_addr</i> , <i>empty</i> , <i>empty</i>)	target_addr : directory map that will be removed empty : empty value
4	Free (<i>target_addr</i>)	target_addr : directory map that will be removed
5	MultiOp (<i>parent_addr</i> , <i>comparisons</i> , <i>writes</i> , <i>removes</i>)	parent_addr : parent directory map of <i>path</i> comparisons : (journal, written rmdir journal) writes : (journal, empty string) removes : (name of target directory)

contains at most one key, which can only be a journal key since it is not considered a directory entry. If the directory is not empty, the written journal must be cleared and the operation returns a status that indicates that the directory is not empty.

4. Free directory map.
5. `MultiOp` at the parent directory to clear journal and remove the target directory's inode key. The operation returns success upon the return of `MultiOp`.

4.8.7 `mknod(path, mode)`

The `mknod` operation creates a file at the given *path* and initializes its permissions based on *mode*. The following are the sequences of MAPSTORE operation calls for `mknod`:

1. Open the parent directory map with read and write access.
2. `MultiOp` at parent directory to write the operation's journal. The journal comprises of the file's name, the map address that will be used for the checkpoint map, the inode value that will be used for the file, and the credential sets that reflect the permissions of *mode*.
3. Concurrently, `Allocate` the checkpoint and log maps.
4. `MultiOp` to clear the journal at the parent directory using the written journal for comparison and the journal key for the remove list. On the return of the `MultiOp`, `mknod` returns success.

4.8.8 `symlink(from, to)`

The `symlink` operation creates a symbolic at the given *from* path. The *to* path is a relative or absolute path that will be evaluated by FUSE at run-time. The following are the MAPSTORE operation calls implemented for `symlink`:

1. Open the parent directory map with read and write access.
2. `MultiOp` at parent directory. The comparison map contains empty values for the keys of the journal and symbolic link's name. By asserting that no journal operation is in progress, no other operation will be creating an inode with the same

name as the `symlink` operation. If this comparison were left out, a possible race condition could lead to a corrupted inode when operations attempt to write to the same inode key. As for the comparison value for the symbolic link's name, the empty value indicates that the key is not in use by another file, directory, or symbolic link. If the comparison fails for the symbolic link's name, then the `symlink` operation will return a status value indicating that the name is already in use. Upon a successful `MultiOp` call, the `symlink` returns success.

4.8.9 `unlink(path)`

The `unlink` operation removes a symbolic link or file specified by the *path*. The following are the sequence of MAPSTORE operation calls for `unlink`:

1. Open the parent directory map with read and write access.
2. `MultiOp` at the parent directory. Depending if the object is a symbolic link or a file, the `MultiOp` operation will do the following:
 - **Symbolic Link.** The comparison map contains the journal key with an empty value and the symbolic link's inode key with the current value of the inode. The remove list contains only the inode key value. If the comparison fails on the inode key because it doesn't exist, the `unlink` operation returns a status indicating that the symbolic link does not exist. If the comparison fails because the inode key is out of date, the inode is updated and the `MultiOp` is reattempted. Upon a successful `MultiOp` call, `unlink` returns a success status value.
 - **File.** The comparison map contains the journal key with an empty value while the write map contains the journal of the operation. Specified in the `unlink` journal is the file's name. The operation continues to the following steps.
3. Concurrently, `Free` the checkpoint and log maps of the file.

4. `MultiOp` to clear the journal at the parent directory using the written journal as a comparison and the journal key for the remove list. On the return of the `MultiOp`, `unlink` returns success.

4.8.10 `rename(from, to)`

The `rename` operation changes the name of an object (file, directory, or symbolic link) when both the *from* and *to* path have the same parent directory. In the case that the parent directories differ, the operation copies the inode from the *from* path to the *to* path and deletes it from the *from* path. The following are the `MAPSTORE` operation calls for the `rename` operation:

1. Concurrently, `Open` the parent directories of the *from* and *to* paths.
2. Depending if the the *from* and *to* path have the same parent directories or not, the operation has two possible courses of action:
 - **Same Parent Directories.** `MultiOp` at the parent directory with a comparison map containing an empty value for the journal key and the *to* key. In addition, the comparison has an entry for the *from* key with the current inode of the *from* object. These comparisons ensure that the source object exists while the destination object does not. In order to copy the source object to the destination, the write map contains the the current inode value for the *to* key and the remove list contains the *from* key. If the comparison fails on the *from* key, the inode of the *from* key is updated and the `MultiOp` is reattempted. In the case that the *to* key fails, the `rename` operation returns a status indicating that the destination object already exists. Otherwise, the operation returns a success status value.
 - **Different Parent Directories.** `MultiOp` at both parent directories to journal the operation with the comparison maps setting the journal key as empty. Both journals consists the map addresses and paths of *from* and *to* as well as the inode of the *from* object. In addition, the comparison map of *from*'s

parent directory contains the inode of *from* to guarantee its existence for the `rename` operation. The comparison map of *to*'s parent directory contains an empty value for the *to* key to ensure that target key does not exist. After both `MultiOp` completes, proceed to the next step.

3. `MultiOp` at *to*'s parent directory to write *from*'s inode to the *to* key. The comparison contains the previously written journal to ensure that no one else has completed this `rename` operation. The write map contains the *from* inode data for the *to* key.
4. `MultiOp` at *from*'s parent directory to clear its journal and remove *from*'s inode. The comparison contains the written journal from the beginning of the operation for the same reason as the previous step. As for the write, it contains an empty value for the journal key to clear the journal. As for the remove list, it consists of the *from* key to remove *from*'s inode.
5. `MultiOp` at *to*'s parent directory clear its journal. The comparison map comprises of the written journal and the write map has an empty value for the journal key. Upon the return of the `MultiOp`, `rename` returns a success status value.

4.8.11 `chmod(path, mode)`

The `chmod` operation changes the permissions of the object (file or directory) specified by the given *path* based on the *mode* value. Permissions are equated from *mode* into MAPSTORE's credentials. Since MAPSTORE does not have an equivalent UNIX group permissions, any changes to group permissions are ignored. The following are the MAPSTORE operations calls for `chmod`:

1. Open the parent directory map with read and write access.
2. `MultiOp` at the parent directory. The comparison map consists of an empty value for the journal key and the object's current inode value for the key of the object's name. If the `MultiOp` comparison fails on the key of the object's name

because the inode is out of date, the inode data is updated and the `MultiOp` is reattempted. Otherwise, the `chmod` returns a status value indicating that the object does not exist. The journal is composed of the object's name, the map address of the object, and the credentials to reflect the permission changes.

A journal is used for the `chmod` operation to serialize operations that involve inode data, most notably the `rename` operation. For instance, imagine a scenario where a `chmod` operation is followed by a `rename` operation. It is dependent that the `chmod` completes before the `rename` operation proceeds. If the `chmod` operation did not journal its operation, the `rename` operation could remove the inode that the `chmod` depends on, causing the `chmod` operation to fail. With a journal, the `chmod` operation guarantees that its operation must complete before any other operation affecting its inode data can proceed.

3. `SetPermissions` on the object's map address (the directory map for a directory and the checkpoint and log maps for a file) to reflect the change in permissions by the `chmod` operation request.
4. `MultiOp` to clear journal and update the inode's data at the parent directory. The comparison map contains the written journal and the previous inode key-value pairs. The write map contains an empty value and the updated inode value for the journal and inode keys, respectively. Upon the return of the `MultiOp`, the `chmod` operation returns a status value indicating success.

4.8.12 `open(path, read, write)`

The `Open` operation opens a file specified by *path* for reading and/or writing indicated by the *read* and *write* flags. The operation calls `Open` on the file's checkpoint and log maps to cache their *MapDescriptors* for future operations and return a corresponding file handler, which contains the map address of the checkpoint. In subsequent file operations, the user will pass along the file handler for each request.

4.8.13 `write(path, mapAddr, buf, offset)`

The `write` operation writes to the file specified by *path* the contents of *buf* at the given *offset*. The *mapAddr* is actually retrieved from the file handler from a previous `Open` request, so no path traversal is necessary to locate the file's log map. To begin the operation, `MAPSTOREFS` will `Open` the file's corresponding log map. Fortunately, the *MapDescriptor* of the log map is cached from a previous `Open` request, avoiding a network request to `MAPSTORE`. Afterwards, the operation calls `Append` on the log map in order to commit the request's write log, completing the operation.

4.8.14 `truncate(path, mapAddr, size)`

The `truncate` operation will either truncate or extend a file depending if the *size* value is less than or greater than the file's current size, respectively. When truncating to a size less than the current file's size, the extra bytes will be removed from the file. In the case of extending the file, the extra bytes will be defaulted to zero values. The operation logic proceeds exactly like `write`, but instead of appending a write log, it commits a truncate log in its place.

4.8.15 `read(path, mapAddr, offset, size)`

The `read` operation reads *size* bytes from the file specified by *path* beginning from *offset*. As like the previous `write` and `truncate` operations, the *mapAddr* of the file is the retrieved from the file handler of the `Open` request of the file. The operation details are the following:

1. Concurrently, `Open` the checkpoint and log maps of the file.
2. Concurrently,
 - `ReadList` all corresponding keys corresponding the read request's byte range (determined by *offset* and *size*) from the checkpoint map

- ReadRangeMap the log map to gather all logs that have not been applied to the checkpoint map.
3. Apply logs one by one only to the pertaining byte range of the `read` request. Once all logs are applied, return the result.

4.9 Applying Logs to the Checkpoint

The read performance of MAPSTOREFS is tightly dictated by the number of logs present in the file's log map, depicted in Figure 4.3. Processing of a read request requires gathering all logs from the log map and applying them locally to the checkpoint's data for every request. The more logs present, the more local processing is required to handle the request. Since read requests are divided into 128 kilobyte chunks by FUSE, logs are redundantly applied to the checkpoint's data for each read request since the logs remain uncommitted to the checkpoint map and present in the log map. Therefore, to avoid unnecessary processing and to increase performance, the logs need to be permanently applied to the checkpoint and expunged from the log map while maintaining data consistency.

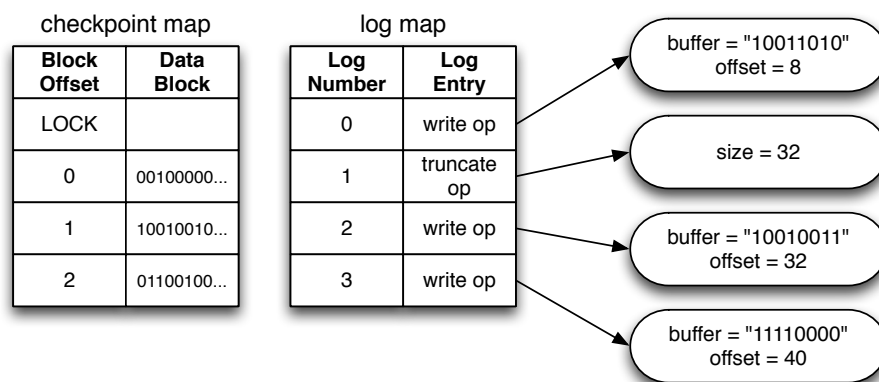


Figure 4.3: Detailed Example of File Maps

In order to minimize the growth of the log map, MAPSTOREFS periodically commits logs to the checkpoint map as a background process. Specifically, a timer is

set after each write request. When the timer expires or after the completion of a read request, logs are applied to the checkpoint and removed from the log map. As logs are being applied, clients can still read the current file since reapplying logs is an idempotent operation.

Consequently, multiple users concurrently applying logs to the same checkpoint map can corrupt the file's data. To prevent users from trampling over the file, a lock mechanism is employed to allow only a single user at a time to apply logs. Associated with each checkpoint map is a reserved key that acts as a file lock, which has a value consisting of a user's IP address along with a timestamp. Before retrieving logs, a user must `CompareAndSet` the lock key to lock the file. Anytime the user updates the checkpoint, a `MultiOp` must be used to guarantee that the user still possesses the file's lock. The reason for the lock's timestamp prevents the file from being lock indefinitely. When a `CompareAndSet` or `MultiOp` fails due to the lock key and the resident lock's timestamp exceeds a fixed duration, the user is allowed to overwrite the lock with his own. Otherwise, the user waits till the timestamp expires and reattempts to lock the file.

MAPSTOREFS applies logs to the checkpoint map in the background through the following procedure:

1. Concurrently, `Open` the file's checkpoint and log maps.
2. Concurrently, `Open` the file's parent directory map and `CompareAndSet` the file's checkpoint to lock the file for log application. The `CompareAndSet` asserts that the current lock value is empty to ensure no other client is currently applying logs to avoid contention.
3. Concurrently, `ReadRangeMap` the log map to retrieve a fixed number of logs and `Read` the file's parent directory to acquire the file's current inode. If there no logs present, the log application ends.
4. `ReadList` to retrieve keys from the checkpoint map pertaining to the log entries.

5. Apply the log entries one by one in order to update the keys of the checkpoint map and update the size of the file's inode. Concurrently, `MultiOp` the checkpoint and parent directory map to update the file's data and inode, respectively. The comparisons for each map pertain to their previous values as well as the lock for the checkpoint map.
6. `MultiOp` to remove the recently applied log entries from the log map. The comparison map consists of the log entries' key-value pairs and the remove map comprises of the log keys.
7. Proceed to Step 3.

In Step 3, the `ReadRangeMap` reads a fixed number instead of all the logs to prevent from over-saturating the network. Because the log application involve multiple requests to `MAPSTORE` while writes typically require one, the rate at which logs are added can overwhelming outpace the rate in which they are removed. Regrettably, an unbounded `ReadRangeMap` could request an immense amount of data from `MAPSTORE`. We therefore bound the number of logs retrieved at a time from the log map.

As an optimization, instead of retrieving the keys pertaining to single log entry, multiple logs are used. Batching multiple logs reduces the round-trip requests with the user and `MAPSTORE` to update the file, which in effect speeds up the log application process. This optimization prevents the degraded performance that occurs when a user writes a large amounts of data to single a file and immediately reads the file afterwards. By the time the read request begins processing, the log application will be vastly behind in committing and removing the logs from the file. As a result, the read request would have to apply a tremendous number of logs, forcing a noticeable degradation in read performance.

Chapter 5

Initial Design

In the original design of MAPSTOREFS, a file consisted on a single map that was similar to the checkpoint map in the current design, but reserved a special file size key-value pair. The file system wrapper service delegated file operations to the OBJECTSTORE Mace service, which supported a interface similar to that of Amazon's S3. OBJECTSTORE was entirely responsible for writing, truncating, and reading files. In addition to handling file requests, OBJECTSTORE maintained a data cache.

Dividing responsibilities between the file system wrapper and OBJECTSTORE services inadvertently introduced some cache consistency problems. Because the file system wrapper maintained a metadata cache, the file size is consequently cached in both the metadata and the file cache. Updating the size in both caches became a difficult task to manage because cache management was not centralized in a single service.

Initially, it was assumed that applications would usually cache block-sized segments of data before writing data to the file system, and for the large part this assumption held. Under this model, it was assumed that serializing writes to the file would result in a minor, tolerable increase in latency as a result of limited contention. However, certain programs (in particular `gcc`) perform a great deal of writes that are usually less than one kilobyte long. To allow for writes to proceed as fast as possible, writes were modified to be asynchronous. These two factors resulted in dozens of concurrent writes to individual file blocks that needed to be serialized; this serialization essentially

crippled the system's write performance under reasonable workloads.

Not only were metadata and data inconsistent in cache in the initial architecture, they had the potential to be inconsistent in MAPSTORE as well. During debugging of the initial architecture, writes to a file's inode were sometimes committing out of order, causing the sizes of a file as listed in its inode to be inconsistent with its corresponding file map.

Through these tribulations, MAPSTOREFS's design transitioned from a single to a two map file abstraction. The design emphasizes on alleviating the cost of updating a file's inode data that was prevalent in the original design. Due to the difficulty of atomically updating both size values in the file's inode and data map, all metadata information is centralized in only the inode. As for allowing asynchronous writes while maintaining an orderly updates to a file's inode, an append map became prudent for performance. By only appending logs for a write request, a file's corresponding inode no longer needs to be updated. To retrieve the current value of the file's inode, a user must apply all logs to the file. Effectively, only a single map is involved when writing to a file instead of two from the initial design (the parent directory map containing the file's inode and the file map itself). The repercussion of this design is that the complexity of reads increases for the performance gains of writes. Fortunately, if the file system can apply and purge logs at a timely pace, degradation in read performance should be negligible.

Chapter 6

Performance Evaluations

In this chapter, multiple aspects of MAPSTOREFS performance is evaluated. In each experiment, MAPSTORE consists of 12 machines, portraying a three data center setup with four machines each, as shown in Figure 6.1. All machines are provisioned with a 2.13 GHz Quad-Core Intel Xeon X3210 processor and 4 GB of RAM. Debian Linux 4.0 with the 2.6.24 Linux kernel are installed on each machine. MAPSTOREFS itself is mounted on a separate machine with the FUSE 2.7.4 kernel module. All machines are connected by a local network. In all experiments, the values presented are averages of ten executions.

6.1 Tuning Variables

The performance of MAPSTOREFS is depended on many tuning variables, such as how many outstanding write requests are permitted and various timers for locks. The following sections explain how these values were experimentally determined. The subsequent micro-benchmarks and macro-benchmarks are configured with the tuning variables established in these experiments.

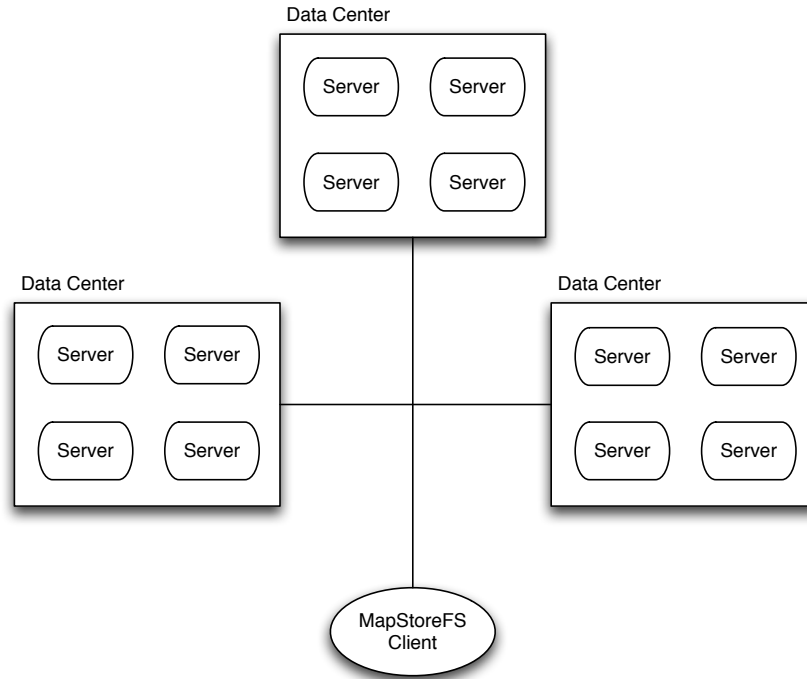


Figure 6.1: Evaluation Setup

6.1.1 Outstanding Writes

Writes in MAPSTOREFS are asynchronous, but MAPSTOREFS bounds the number of outstanding write requests to MAPSTORE at any given time to prevent overloading MAPSTORE. When the file system reaches this bound, succeeding write requests block till outstanding requests are serviced. To determine this value, a single client wrote a 10 megabyte file to the file system. The number of outstanding write requests permitted was varied and the throughput was analyzed. Logs were not applied to the checkpoint in the background in order to not disrupt the raw write performance. The results can be seen in Figure 6.2.

As the bounding size increases, the throughput of increases until file system bounds to 50 outstanding writes. The throughput flattens out from this point to about 31 MB/s. The cause of this plateau is likely due to FUSE's breaking write requests into four kilobyte chunks, forcing multiple context switches.

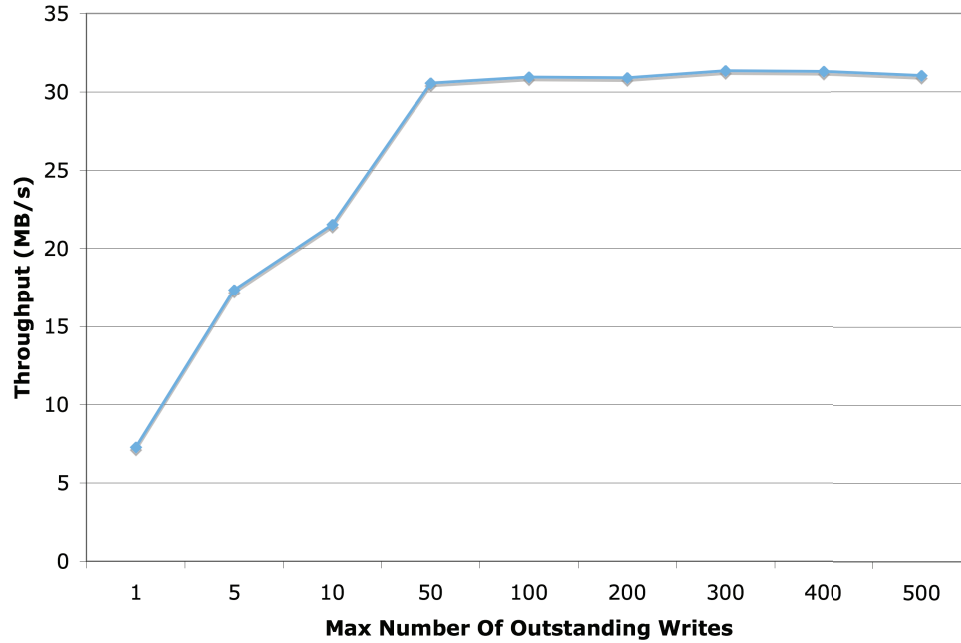


Figure 6.2: Outstanding Writes

Based on these performance numbers, we configured MAPSTOREFS to allow only 50 outstanding writes before it blocks subsequent write requests. Increasing this value does not significantly improve the write throughput further and requires less state information to be maintained by the system.

6.1.2 Applying Logs in Batches

When applying logs to a file's checkpoint, MAPSTOREFS attempts to retrieve and to apply a fixed number of logs at a time. In order to determine the log batch size, a single client wrote a 20 megabyte file and the batch size is varied between executions. A 20 megabyte file was used instead of 10 megabyte file so the log application process occurs concurrently with more write requests. This approach simulates the log application process with a concurrent load, causing it to share CPU resources. By increasing the batch size to the point of CPU saturation, CPU contention with write requests will be more evident. We want to avoid this contention while maintaining a fast log application rate. In this experiment, throughput is defined as the number of bytes applied from the

logs to the checkpoint per second. The results of the experiment can be found in Figure 6.3.

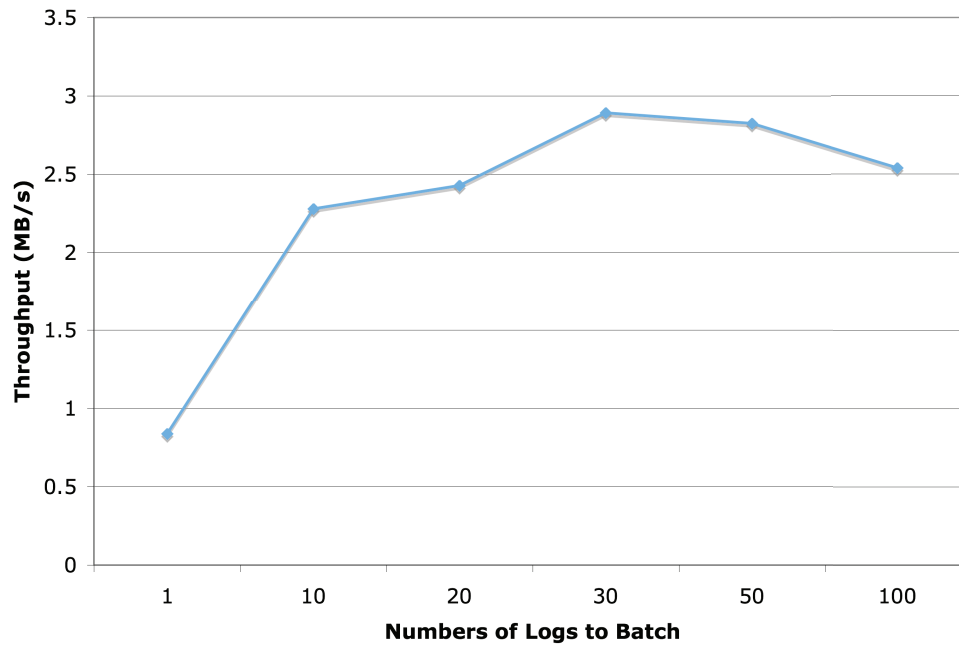


Figure 6.3: Applying Logs in Batches

The throughput increases as the batch size reaches to 30 logs. Afterwards, the overall throughput decreases. The reason for the decline is that when applying more than 30 logs at a time, more CPU times is required for processing logs while handling concurrent write requests. Compared to a batch size of 30 to 50 and 100, the CPU less often peaks to 100 percent usage. As a result, MAPSTOREFS was configured to a batch size of 30, which applies logs to the checkpoint at approximately 3 MB/s. Furthermore, this value was chosen to reduce stress in the system so that it does not affect write performance. We want the log application process to quickly apply logs, but as the same time to be unobtrusive to concurrent file system requests.

6.1.3 Log Application Timer

The file system applies logs to a file's checkpoint map under two events. The first event occurs when a file is read and the file system notices that there are logs present

for the file. After the read operation completes, the file system immediately schedules the logs to be applied to the file's checkpoint. The prompt response prevents subsequent reads from being penalized from applying logs again. In the second scenario, when a write request completes, a timer is set to apply the logs at a later time. Instead of applying the logs immediately, the file system allows further logs to be appended to a file's log map so it can retrieve logs in large batches. This section determines the value of this timer.

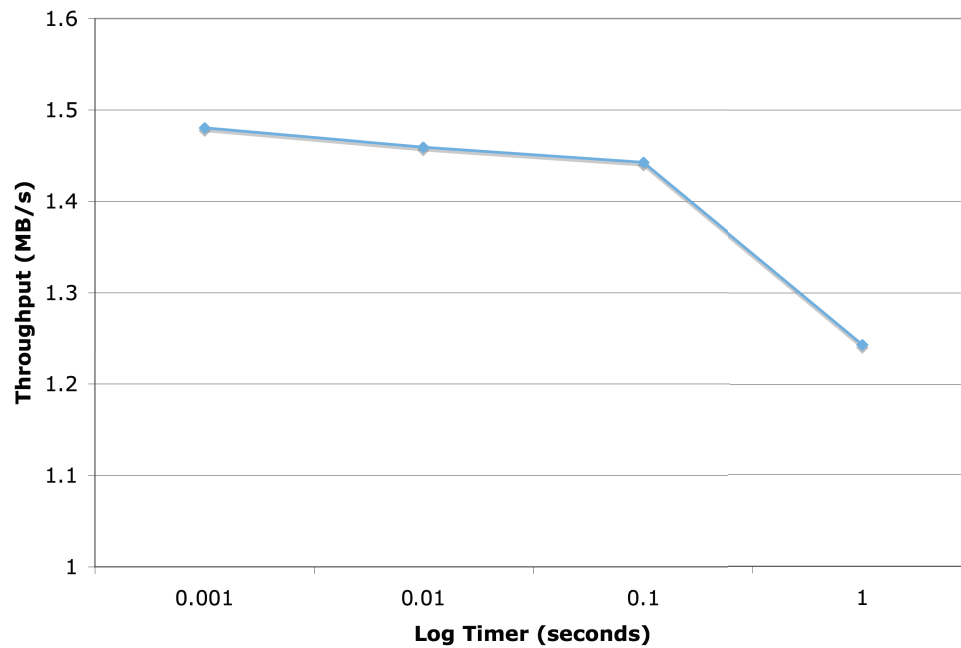


Figure 6.4: Write Log Timer

In this evaluation, a 10 megabyte file is written and immediately read with a variety of log timers. The read throughput is recorded to determine how effectively logs can be applied to the file's checkpoint using different timers. The outcome of the evaluation can be found in Figure 6.4.

When the write log timer is between 0.001 to 0.1 second, the throughput of the immediate read request is about 1.45 MB/s. With a 1 second timer, the throughput drops to about 1.24 MB/s. Given that a timer between 0.001 to 0.1 roughly contributes the same read throughput, we opted to use a 0.1 second timer in order to allow ample

time for logs to accumulate on the log map. Since logs are retrieved in batches during the log application process, the 0.1 second timer prevents MAPSTOREFS from prematurely acquiring a small batch.

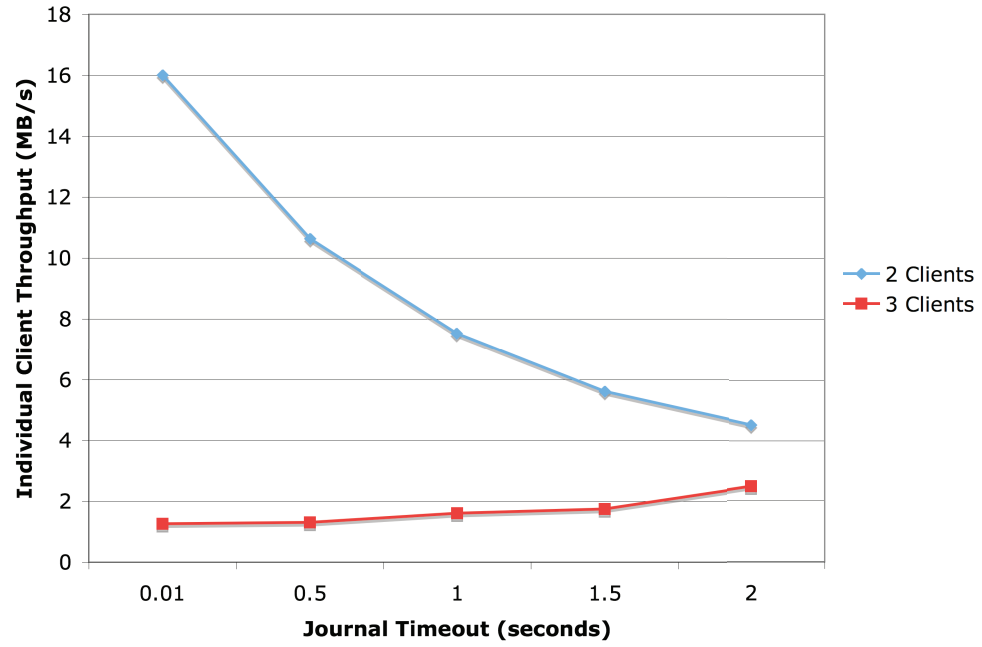
6.1.4 Journal Timer

When a client requests a journal operation, MAPSTOREFS verifies that no journals reside in all directories associated with the operation before proceeding. If a journal is present, MAPSTOREFS waits a fixed time period to allow the other client to finish its journal operation before it attempts to resolve the journal itself. In this section, several journal timers are used to optimize concurrent journal operations. To determine the most suitable journal timer, multiple clients created 100 files each in the same directory. Figure 6.5 shows the overall individual and aggregate file creation throughput with two and three concurrent clients.

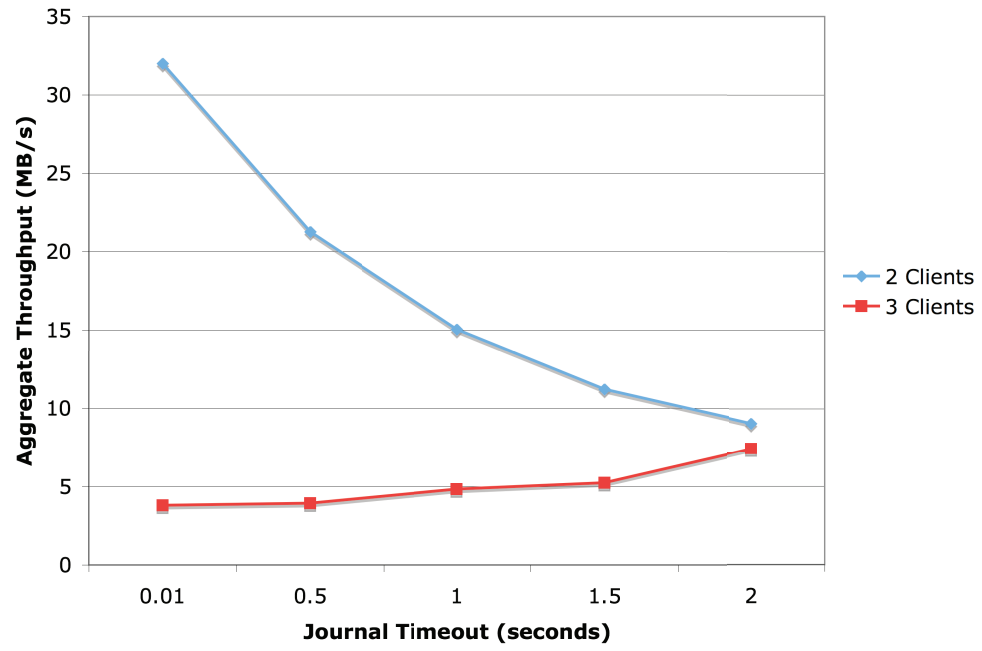
When two concurrent clients create 100 files in the same directory, the overall file creation throughput increases as we increase the granularity of the journal timer. By utilizing a smaller timeout period, clients do not excessively wait idly for another client. This observation is prevalent when the journal timeout period is 0.01 seconds. However, when we introduce three clients, the contrary effect occurs with the overall file creation throughput increasing with the journal timeout period. With more clients, there is more contention for the journal key and consequently, a smaller timeout period causes longer wait times. The performance gain by a longer timeout period is not drastic though. As a result, we opted to use a journal timeout of 0.01 seconds with the assumption that typically only two clients would be manipulating metadata in the same directory.

6.2 Micro-Benchmarks

In the following sections, we examine MAPSTOREFS's metadata, read, and write operations. MAPSTOREFS's metadata creation and deletion operations perform inadequately due to its strong consistency guarantees. Because metadata creations and



(a) Individual Client File Creation Throughput



(b) Aggregate Client File Creation Throughput

Figure 6.5: Journal Timer Performance

Table 6.1: Metadata Performance

Files Created Per Second	15.18
Files Deleted Per Second	9.95
Directories Created Per Second	9.43
Directories Deleted Per Second	9.49

deletions journal their operations, it is unavoidable for MAPSTOREFS to reduce multiple round-trips with MAPSTORE without reducing consistency guarantees. On the other hand, MAPSTOREFS's read operations behave sufficiently well with a throughput about an order slower than the local file system could achieve. Read performance is mainly restrained by the excessive copying that is needed to process and construct the read buffer for the user. For instance, when retrieving the key-values associated with a read request, MAPSTOREFS concatenates all values together to construct a contiguous read buffer for the user. Furthermore, this read buffer is copied once again from the user file system module to the kernel module. Fortunately, if data is cached, read performance yields a throughput comparable to the local file system. As for write operations, they do not deviate far from the local file system speeds since most of the network latency is masked by its asynchronous nature. Write throughput is mainly restricted by FUSE's four kilobyte write request limit. Large write requests are divided into several smaller ones, forcing multiple context switches between the kernel and the userspace file system module. Inevitably, the context switches degrade the overall write throughput.

6.2.1 Metadata Performance

File creation and deletion were evaluated by timing the rate at which the file system creates and deletes 100 files. Directory creation and deletion was evaluated in the same manner in the context of directories. The outcome of the evaluation is summarized in Table 6.1.

Metadata creation and deletion do not fare well in MAPSTOREFS due to its strong consistency guarantees. MAPSTOREFS requires multiple round-trips with MAPSTORE to journal operations before actually proceeding with creations and deletions.

An interesting observation is the performance discrepancy between the file creation and deletion rates. These two rates should be nearly identical since they consist of the same number of MAPSTORE operations, but the file deletion rate is lower. After closer examination, it appears that the response times of `MultiOp` requests during file deletion were significantly slower than during file creation. The reason for this anomaly requires further investigation into MAPSTORE. Unfortunately, this issue occurs during directory creation and deletion, but is masked by MAPSTORE’s performance discrepancy between its `Allocate` and `Free` operations. The latency observed by clients for an `Allocate` compared to a `Free` request is longer because an `Allocate` request requires consensus between MAPSTORE’s data centers before responding to the client. In contrast, a `Free` request to MAPSTORE immediately returns to the client and the request is handled asynchronously.

6.2.2 Uncached Reads

In this section, the read performance is measured in the scenario where all logs have been applied to the checkpoint map. This evaluation shows the optimal read throughput of MAPSTOREFS since no logs need to be processed. We assess read throughput by writing a 10 megabyte file, waiting till all logs are applied in the background, and then reading the file. A wide range of file sizes were examined. The results can be found in Figure 6.6.

The read throughput steadily increases as the file size reaches 5 megabytes. Afterwards, the performance flattens out to about 26 MB/s. The cause of this plateau is most likely due to the excessive copying from MAPSTOREFS to its FUSE module. When MAPSTOREFS receives its data from MAPSTORE, it copies key-value pairs to a contiguous read buffer. Afterwards, the FUSE module copies the buffer to its own buffer for the FUSE kernel module to return to the client. In addition to the copying issue, the 128 kilobyte read size limit also exacerbates the issue.

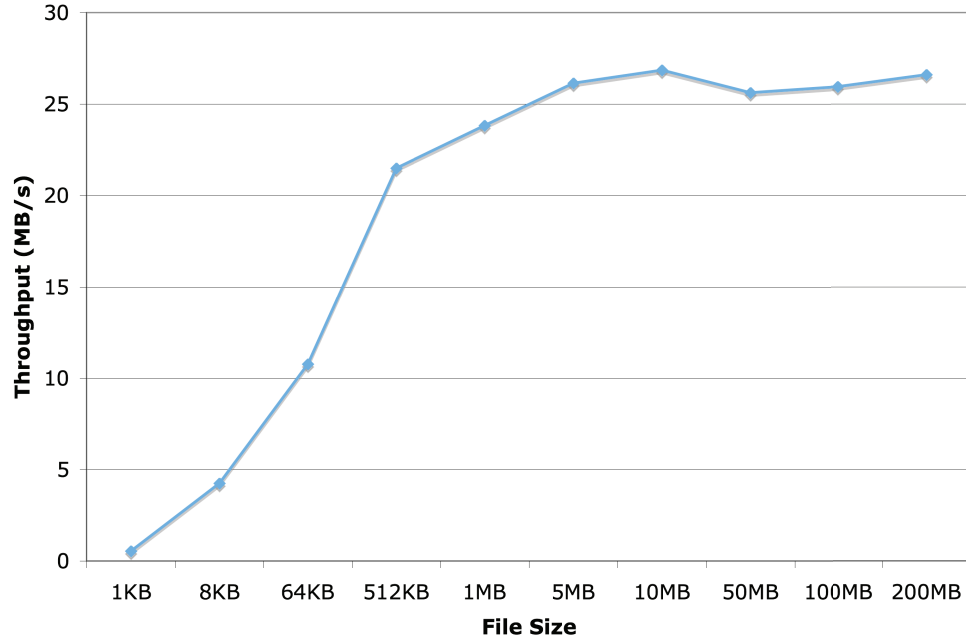


Figure 6.6: Uncached Read Performance

6.2.3 Cached Reads

This section demonstrates the caching performance of FUSE itself. Similar to the previous evaluation, a file is written then read after logs have been applied. A second read request is issued afterwards since FUSE would have the file's data cached into the kernel. The outcome of the experiment is summarized in Figure 6.7.

The caching mechanism of FUSE performs quite well in comparison to MAPSTOREFS when directly fetching data from MAPSTORE. The throughput increases as we increase the read input, but it eventually decreases as it reaches to a 50 megabyte file. At this point the throughput flattens to about 290 MB/s. The cause of this increase is mostly due to the large data size which causes some of the cache data to be vacated.

6.2.4 Writes

To demonstrate the MAPSTOREFS's write performance, a file is written and the throughput of the write is recorded. The same file size range from the previous

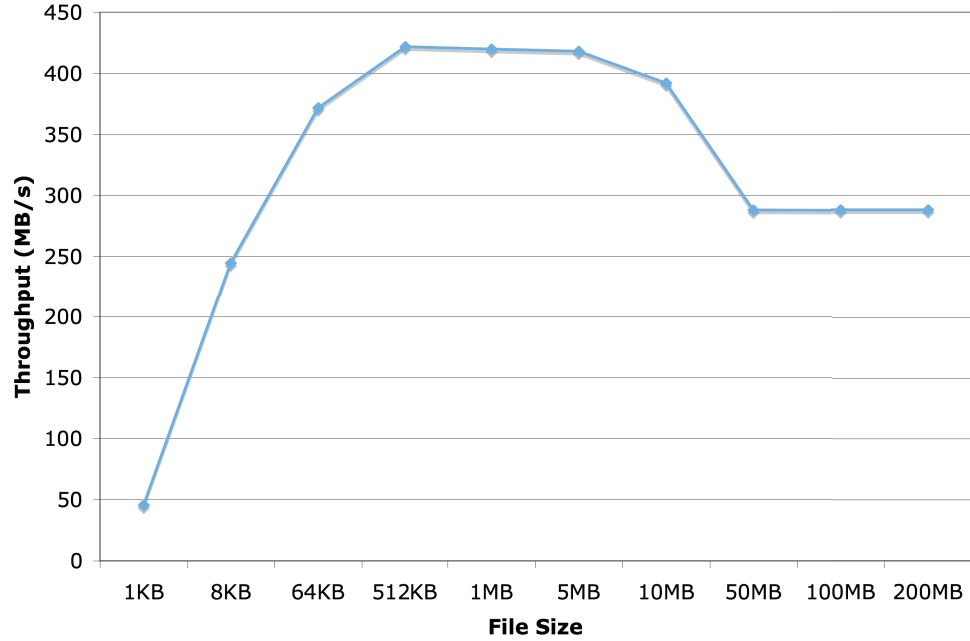


Figure 6.7: Cached Read Performance

experiments were used. The outcome of the writes requests is shown in Figure 6.8.

The write throughput steadily increases with the file size. The peak throughput reaches to approximately 28 MB/s when writing a five megabyte file. Unfortunately, the write throughput drastically decreases when writing 10 megabytes and more. After closer examination, the cause of this decline is due to MAPSTORE's `Append` operation. When determining the next sequential key to associate with the `Append` data, MAPSTORE finds the first key in sorted order associated with the requested map. Thereafter, MAPSTORE iterates over all keys to the last key, incrementing the last key to determine the new key for the `Append` request. Consequently, this process takes linear time. The diminished write performance is more evident at larger requests due to the log application process. The rate at which logs append to the log map vastly outpaces the log application process from removing those logs. As a result, `Append` operations constantly iterate through an ever increasing key set for the log map. Currently, MAPSTORE cannot avoid this iteration because it stores its data using the Berkeley DB, which does not provide an operation that efficiently determines the last key of a map.

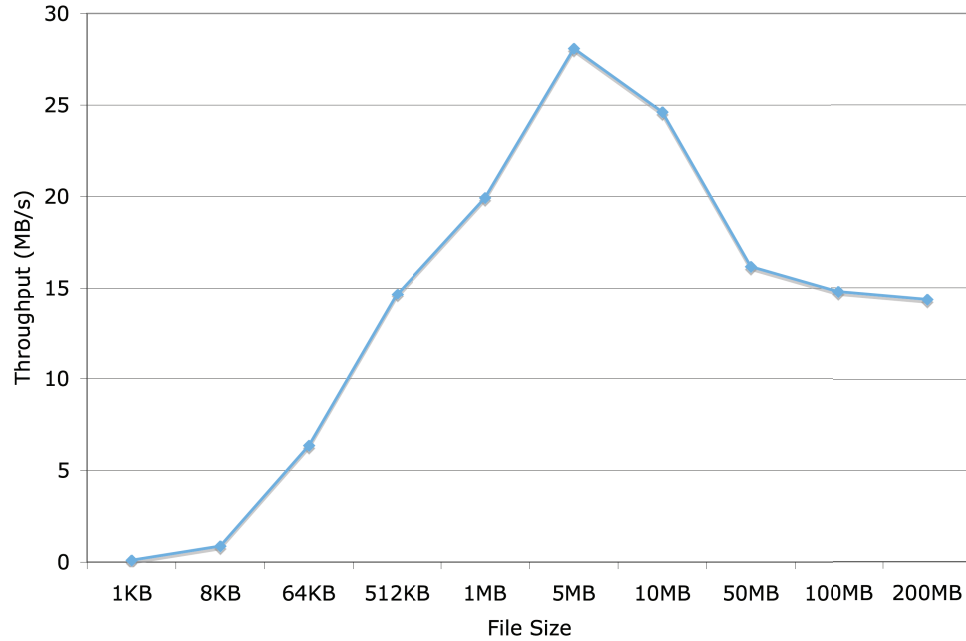


Figure 6.8: Write Performance

6.3 Macro-Benchmarks

The following sections evaluate real world workloads. First, to evaluate meta-data intensive applications, the Network Appliance's PostMark benchmark was ran on MAPSTOREFS. Secondly, OpenSSH [11] was compiled on MAPSTOREFS. Both benchmarks were ran on the local file system of the client machine for comparison.

6.3.1 PostMark

The PostMark benchmark simulates an application that manages small file sizes, such as an email server. In this experiment, version 1.51 of PostMark was used with the default parameters. The benchmark consisted of 764 file creations, 243 read operations, 257 append operations, and 764 file deletions. PostMark also mixes file creation and deletion periodically with a read or append operation, which it dubs as a *transaction*. Of all file creations, 264 of them were mixed with a transaction. As for file deletions, 236 were mixed with transactions. The total amount of data read was 1.36

Table 6.2: PostMark Evaluation

	MAPSTOREFS	Local File System
Total Time (seconds)	111.1	2.1
Total Time Spent For Transactions (seconds)	33.9	1
Transactions Per Second	15	500
Files Created Per Second (With & Without Transitions)	6.5	369.2
Files Created Per Second (Without Transactions)	13.1	500
Files Created Per Second (With Transactions)	7.7	264
Read Operations Per Second	7.2	243
Append Operations Per Second	7.5	257
File Deletions Per Second (With & Without Transitions)	6.5	369.2
File Deletions Per Second (Without Transactions)	14.7	528
File Deletions Per Second (With Transactions)	6.9	236
Read Throughput (KB/s)	12.85	675.33
Write Throughput (KB/s)	41.88	2197.50

megabytes and the total written was 4.45 megabytes. The output produced by PostMark while running on MAPSTOREFS and the local file system is summarized in Table 6.2.

Examining PostMark’s evaluation of MAPSTOREFS, it is evident that MAPSTOREFS does not perform well in metadata creation and deletion operations, as previously seen in the micro-benchmarks. When isolating the file creation operations alone without PostMark transactions, the throughput performed similar to the micro-benchmarks. However, the file deletion rate showed improvement, which is likely caused by MAPSTORE’s `MultiOp` response times behaving well under PostMark, contrary to what occurred during the micro-benchmarks. Further investigation in `MultiOp`’s performance behavior is needed. When mixed with transactions, the throughput drastically drops to approximately to a half of its value for file creation and deletion. In this scenario, many of MAPSTOREFS optimizations are underutilized since

Table 6.3: OpenSSH Compilation

	Configuration Time (seconds)	Make Time (seconds)	Total (seconds)
MAPSTOREFS	275.57	50.95	326.52
Local File System	24.35	29.19	53.54

small read requests cannot be parallelized by FUSE. In addition, file creation and deletion requests block other metadata operations till completed when managing files in the same directory. To alleviate this issue, we need to increase the journal locking mechanism granularity. Instead of reserving a single journal key at each directory, directories would contain multiple journals. This mechanism has yet to be implemented.

6.3.2 Compiling OpenSSH

The procedure in which OpenSSH was compiled on MAPSTOREFS consisted of copying the source directory to the MAPSTOREFS mount point, running the configure script, and immediately executing the make script. Therefore, logs could be potentially running from the configure script while make executes. The version of OpenSSH compiled is 4.7p1. The results of the compilation on MAPSTOREFS and the local file system is summarized in Table 6.3.

Needless to say, the local file system outperforms MAPSTOREFS. MAPSTOREFS suffers from the file creation and deletion portions of the configuration and make process during compilation. Metadata creation and deletion is more evident in the configuration process since it creates and removes many small files compared to during make. Most of files generated during make are only a few kilobytes while the largest binaries consists of about 10 megabytes. MAPSTOREFS yields greater write throughput with large contiguous requests, but compilation involves many small write requests. Consequently, MAPSTOREFS cannot fully maximize its write throughput performance.

6.3.3 Hypothetical OpenSSH Compilations

To fully comprehend the poor compilation performance of OpenSSH, we examined the latency times of MAPSTOREFS and MAPSTORE operations. Unfortunately, we discovered the latency times of multiple MAPSTORE operations were inconsistent and widely variable. For instance, the variable latency times caused `mknod` operations to fluctuate between 20 and 100 milliseconds with an average latency of 55 milliseconds. Since `mknod` mainly consists of three MAPSTORE operations (`MultiOp`, `Allocate`, `MultiOp`), each MAPSTORE operation consumed approximately 18.33 milliseconds. We believe that MAPSTORE should be able to achieve an average latency time significantly lower. Therefore, we concluded there were issues in the MAPSTORE implementation causing these discrepancies.

In this section, we provide hypothetical runtimes of the configuration and make phases of OpenSSH if the MAPSTORE operation latency times improve. Specifically, we adjust the `mknod` and `unlink` average latency time with the assumption that the `MultiOp`, `Allocate`, and `Free` operations all execute with the same latency times. We emphasize these two operations since they consumed most of the time during the configuration phase under MAPSTOREFS. Improvements in other MAPSTORE would coincide and reduce these hypothetical results even further, but for simplicity, we focus on these two operations.

Based on logs, the configuration phase requests 1421 `mknod` and 1413 `unlink` operations. In the make phase, 64 and 62 operations were `mknod` and `unlink`, respectively. We use the following formula to compute the hypothetical runtimes:

$$Runtime = (Base\ Time) + (\#\ of\ mknod/unlink\ Operations) \times (Avg.\ Latency)$$

The base time consists of the time that our configuration and make phases spent without `mknod` and `unlink` operations. Assuming an average `mknod` and `unlink` latency time of 55 milliseconds during our original benchmarks, the base time for the configuration and make phases are 119.70 and 44.02 seconds, respectively. Table

Table 6.4: Hypothetical OpenSSH Compilation for MAPSTOREFS

Mknod/Unlink Avg. Latency (milliseconds)	Configuration Time (seconds)	Make Time (seconds)	Total (seconds)
3	128.20	44.21	172.41
15	162.21	44.98	207.19
30	204.72	45.94	250.66
45	247.23	46.60	294.13
55	275.57	50.95	326.52

6.4 lists the possible runtimes of compiling OpenSSH if the average latency of `mknod` and `unlink` improve.

Not surprisingly, MAPSTOREFS performs much more effectively, especially with average latency times of 3 and 15 milliseconds. Therefore, MAPSTOREFS is capable of providing more compelling benchmarks, but requires MAPSTORE operations to reduce its latency times.

Chapter 7

Related Works

In this chapter, we discuss related systems that have influenced MAPSTORE and MAPSTOREFS. We also compare and contrast the various characteristics of these systems.

7.1 Log-Structured File System

MAPSTOREFS is strongly influenced by the Log-Structured File System (LFS) [12] for its design of storing data permanently as logs. Any modification to the file system is stored in a log and sequentially appended to disk. Essentially almost all seeks are eliminated during the write process as a result. Furthermore, LFS buffers file system changes and commits them in fixed-size extents, called *segments*, to disk as a single write operation. In order to identify inodes, LFS maintains an *inode map* to locate inode blocks on disk. The inode map itself is divided into blocks and located by a fixed checkpoint region on disk. MAPSTOREFS follows a similar approach by appending logs when committing writes to the file system. Unlike LFS, MAPSTOREFS's inodes reside only in a single region of the file system, which is the parent directory of file, directory, or symbolic link. The write logs only need to be read to update a file's size attribute. Fortunately, there is no need for an inode map in MAPSTOREFS.

Over time the free space of the disk will be fragmented as files are created and deleted. Consequently, write throughput decreases as large contiguous writes will not

be possible. LFS avoids this issue by periodically copying used data blocks from segments and writing them back into contiguous blocks. This process creates contiguous free space on disk as well as aggregates associated data blocks. MAPSTOREFS has an analogous problem with the growth of a file's log entries. The log application was introduced to reduce log growth and improve read performance, similar to LFS's segment cleaning.

7.2 Frangipani

Frangipani [14] is a scalable distributed file system very similar to MAPSTOREFS. The system is divided into a two-layer system: a lower level providing data storage and higher level providing file system code. The lower layer is Petal, a distributed storage system providing virtual disks. Frangipani itself is the higher layer that leverages Petal as its storage layer, inheriting many of its attributes, such as scalability and availability. Frangipani runs in the kernel so it is not easily portable like MAPSTOREFS that uses the FUSE kernel module. In order to synchronize updates to its files, Frangipani utilizes a distributed lock service, locking at a per file basis.

Similar to MAPSTOREFS, Frangipani maintains a redo log on Petal for pending changes to the file system. In case of a failure by a client, another Frangipani server can recover the file system to a consistent state. It must be noted that these redo logs only maintain information for recovering metadata and not file data itself. As opposed to Frangipani, MAPSTOREFS's journal mechanism and write logs can recover both metadata and file data.

As for security, Petal itself does not have a security mechanism for authentication. Any Frangipani client can read or write to any block of Petal's virtual disks. In contrast, MAPSTORE requires a client to provide his credentials before accessing its maps.

7.3 Dynamo

Amazon's Dynamo [4] is a highly available data storage with a key-value interface comparable to MAPSTORE. Dynamo provides two operations for its objects: `get(key)` and `put(key, context, value)`. The `get` operation returns an object or a list of objects if conflicts occurred during an update along with a *context*. The *context* value contains metadata information about the object, such as its version, to resolve update conflicts. The `put` operation stores the *value* to the associated *key* with its *context*. Instead of resolving conflicts itself, Dynamo leaves conflict resolution to the application services. Analogously, MAPSTORE leaves conflict resolution to the client with the `MultiOp` operation. By comparing a set key-value pairs, a client provides a context for an update.

Dynamo uses a variation of consistent hashing to distribute its key-values across its servers. Instead of a server being only responsible for single node in the consistent hash circle, a server is responsible for multiple "virtual nodes" to improve load distribution. Dynamo replicates its data *N* times, where *N* is a tunable parameter. In addition to the single node that is responsible for a key range, its *N* - 1 successor nodes maintain replicas. When updating a value, the replicas are notified asynchronously of the change so delays can cause a subsequent `get` operation to have an older value. As a result, Dynamo provides eventual consistency for its data. Compared to Dynamo, MAPSTORE uses Paxos consensus to update its replicated data across and within data centers. Updates may not be as efficient as Dynamo, but MAPSTORE provides stronger consistency guarantees across its servers.

7.4 Sinfonia

Sinfonia [1] provides a storage layer service similar to MAPSTORE. It implements its service within a cluster of machines for single data center rather than MAPSTORE's environment of machines disperse across data centers. Data is stored across memory nodes where each node supports a linear address space for its data. Memory

nodes either maintain their data in RAM or stable storage.

In order to manipulate Sinfonia’s data, it provides a single operation, which it designates as a *minitransaction*. Analogous to MAPSTORE’s `MultiOp`, Sinfonia’s minitransaction consists of sets of compare, read, and write items. Each item lists a memory node, an address range, and other relevant data pertaining to the requested operation. Only when the compare items match the current state of the involved memory nodes can the read and write items be acted upon. In essence, minitransactions provide an atomic operation across memory nodes.

The authors implemented a cluster file system, SinfoniaFS, on top of Sinfonia. Because Sinfonia data abstraction diverges not too far from a local disk’s data abstraction, the notion of superblocks, inodes, and data blocks are easily translated to Sinfonia. Data blocks merely require an additional attribute of a memory node id to identify its location within the cluster. Since an inode’s data blocks can be allocated across multiple memory nodes, SinfoniaFS attempts to colocate these files as much as possible. MAPSTORE, on the other hand, does not have a notion of locality and consequently, cannot optimize data based on node locality.

MAPSTOREFS distinguishes itself from SinfoniaFS because its data is meant to be spread across multiple data centers. Its storage layer, MAPSTORE, does not expose node locality. However, MAPSTORE simplifies data access by abstracting its data into a single address space. It is not fair to compare these two file systems in performance since both target different environments, but they do have similar characteristics in their storage layers as already mentioned.

7.5 Cooperative File System

The Cooperative File System (CFS) [3] is built on a peer-to-peer read-only storage system, utilizing a distributed hash table, DHash, along with the Chord [13] lookup service. The DHash service maintains the data blocks of the file system across multiple servers. It is also responsible for replicating data blocks between other servers

as well as caching blocks. As for Chord, it supplies routing tables to locate data blocks for the file system. Analogous to MAPSTOREFS, the combination of DHash and the Chord lookup service is CFS's storage layer.

Since CFS is a read-only storage system, it distinctly targets two types of users: publishers and consumers. Publishers produce data and store them within CFS for consumers to read. Only the publisher of a data block in CFS can modify it. By distinguishing user roles, CFS essentially supports a single-writer, multiple-readers policy for its data. In contrast, MAPSTOREFS allows multiple-writers, multiple-readers policy for its content. Consequently, MAPSTOREFS must handle more data consistency issues for write and read requests that CFS avoids. As a result, CFS more freely caches its data blocks than MAPSTOREFS since data is not actively updated by multiple users.

Chapter 8

Conclusion

MAPSTORE addresses the growing demand of highly available and strongly consistent storage layers. In order to be a plausible solution, MAPSTORE's map interface must articulate a wide range of applications. This thesis exemplifies that MAPSTORE's maps can undoubtedly express complex systems such as a file system.

MAPSTOREFS has been shown to be a capable file system running on top of MAPSTORE. Although MAPSTOREFS's performance is not overwhelming compelling, multiple pending improvements in MAPSTOREFS and MAPSTORE potentially can prove otherwise. For instance, concurrent metadata creations and deletions within the same directories are currently serialized because directories only possess a single journal for multi-map atomicity. The single journal design simplified the replay of an uncompleted operation and did not consider the performance ramifications of concurrency within a single directory. To alleviate this problem, MAPSTOREFS must support multiple journals within a directory. When journaling a `mknode` operation, for example, instead of writing a journal at a reserved key, MAPSTOREFS writes the journal into the file's inode key. Effectively, the inode is speculative till the journal is removed from the inode value.

Further work needs to be allocated in improving MAPSTORE itself. As discovered in the macro-benchmarks, inconsistencies in MAPSTORE's operation latencies consequently degraded the performance of MAPSTOREFS. If the MAPSTORE latency

times improve, it would unquestionably refine MAPSTOREFS's performance.

Although MAPSTOREFS's metadata performance overwhelmingly underperforms compared to a local file system, it promises strong consistency between multiple machines while providing data replication and fault-tolerance. Establishing consensus between replicas in MAPSTORE is inherently latency intensive and therefore a penalty MAPSTOREFS cannot avoid. In order to yield higher performing metadata operations, the consistency constraints must be relaxed to be competitive to a local file system. Nonetheless, this thesis demonstrates that MAPSTORE's API is capable of an expressing a file system.

Bibliography

- [1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.
- [2] Amazon.com. Amazon Web Services. 2008. <http://s3.amazonaws.com/>.
- [3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.
- [4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.
- [5] FUSE. Filesystem in Userspace. 2008. <http://fuse.sourceforge.net/>.
- [6] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. *SIGPLAN Not.*, 42(6):179–188, 2007.
- [7] C. E. Killian, J. W. Anderson, R. Jhala, and A. M. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. *NSDI*, 2007.
- [8] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix (revised july 27, 1983). Technical report, Berkeley, CA, USA, 1983.
- [10] OpenSolaris. ZFS. 2008. <http://opensolaris.org/os/community/zfs/>.
- [11] OpenSSH. OpenSSH. 2008. <http://www.openssh.com/>.
- [12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. Technical report, Berkeley, CA, USA, 1992.

- [13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.
- [14] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 224–237, New York, NY, USA, 1997. ACM.