

Extendable Storage Framework for Reliable Clustered Storage Systems

Sumit Narayan, Ph.D.

University of Connecticut, 2010

The total amount of information stored on disks has increased tremendously in recent years with data storage, sharing and backup becoming more important than ever. The demand for storage has not only changed in size, but also in speed, reliability and security. These requirements not only create a big challenge for storage administrators who must decide on several aspects of storage policy with respect to provisioning backups, retention, redundancy, security, performance, etc. but also for storage system architects who must aim for a *one system fits all* design. Storage policies like backup and security are typically set by system administrators for an entire file system, logical volume or storage pool. However, this granularity is too large and can sacrifice storage efficiency and performance – particularly since different files have different storage requirements. In the same context, clustered storage systems that are typically used for data storage or as file servers, provide very high performance and maximum scalability by striping data across multiple nodes. However, high number of storage nodes in such large systems also raises concerns for reliability in terms of loss of data due to failed nodes, or corrupt blocks on disk drives. Redundancy techniques similar to RAID across these nodes are not common because of the high overhead incurred owing to the parity calculations for all the files present on the file system. In the same way, data integrity checks are often omitted or disabled in file systems to guarantee high throughput from the storage system. This is because not all the files require redundancy or data protection mechanism, and achieving higher throughput outweighs the need to have these computationally expensive routines in

place.

In this thesis, we begin by studying the I/O access patterns of different applications that typically use clustered storage. The study helps us understand the application requirements from a file system. We then propose a framework for an attribute-based extendable storage system which will allow storage policy decisions to be made at a file-level granularity and at all levels of the storage stack, including file system and device managers. We propose to do this by using a file's extended attributes that will enable different defined tasks via plugins or functions implemented at various levels within the storage stack. The applications can set extended attributes for their files, or directories, and extract a complete content-aware storage functionality from the storage system stack. We present a stackable user-space file system which will make it easier to write and install these plugins. Using our stackable file system technique, these plugins can be programmed in user-space and mounted by non-privileged users. We also provide two scenarios where our framework can be used to provide an overall improved performance for a reliable clustered storage system.

Extendable Storage Framework for Reliable Clustered Storage Systems

by

Sumit Narayan

B.E., University of Madras, 2002

M.S., University of Connecticut, 2004

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2010

UMI Number: 3420180

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

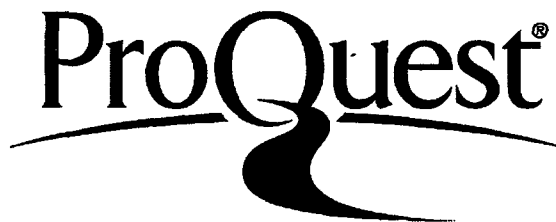
In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3420180

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346


APPROVAL PAGE

Doctor of Philosophy Thesis

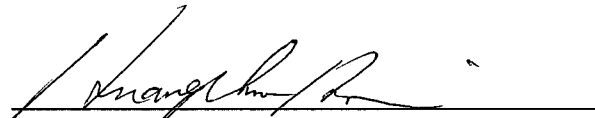
Extendable Storage Framework for Reliable Clustered Storage Systems

Presented by
Sumit Narayan, M.S.

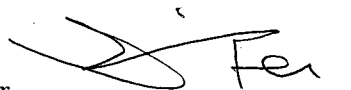
Major Advisor


John A. Chandy

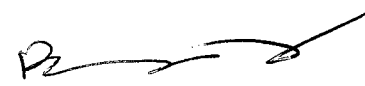
Associate Advisor


Chun-Hsi Huang

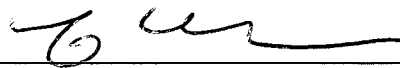
Associate Advisor


Yungsi Fei

Associate Advisor


Bing Wang

Associate Advisor


Robert Ross

University of Connecticut
2010

Acknowledgements

I would like to thank many people who have contributed either directly, or indirectly, in making this thesis a reality for me. I will try to list all of them and I apologize if I have missed out some names.

I would like to begin with thanking and expressing my deep and sincere gratitude to my advisor Professor John Chandy for his constant faith in me and his support every time I tried to pursue a new direction for research. His advice was always insightful and discussions with him always left me enlightened.

My sincere thanks to Chun-Hsi Huang, Yungsi Fei and Bing Wang for serving on my thesis committee. They provided me with valuable insights and comments to improve this work. Special thanks to Rob Ross for his interest, helpful advice and his feedback on this work, not to mention serving on my thesis committee.

Many thanks to Vishal Ravindra for being a wonderful roommate and for all the late-night debates and sport discussions. Thanks to my friends Christian Berger, Javier Areta, Manisha Mishra, Marco Guerrero, Ozgur Erdinc, and William Donat for making my stay at UConn a memorable one. I would also like to thank my colleagues AjithKumar Thamarakuzhi, Jianwei Dai, Mike Kapralos, Shuo Wang, and Weiguo Tang for their help with various things.

Last, but most importantly, I would like to thank my loving family. My parents have played a very big role in getting this thesis done. Their immense love, support and enormous amount of encouragement always allowed me to stay focused on my work. I am grateful to my brother Vineet and sister-in-law Nancy for their love and

encouragement. Their surprise weekend visits always made me happy. I would like to thank Suvasri for her love, affection and understanding. She has been a constant source of inspiration and motivation, without which I wouldn't have been able to complete this thesis.

The financial support of the National Science Foundation is gratefully acknowledged.

Contents

Chapter

List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Outline	5
2 Background and Related Work	6
2.1 Storage Systems	6
2.2 Redundant Arrays of Inexpensive Disks (RAID)	7
2.3 Classification of Storage Systems	9
2.3.1 Direct Attached Storage (DAS)	9
2.3.2 Storage Area Network (SAN)	9
2.3.3 Network-Attached Storage (NAS)	10
2.4 Clustered Storage Systems	10
2.4.1 Parallel Virtual File System (PVFS)	12
2.4.2 Lustre File System	13
2.5 I/O Characterization	13
2.6 File System Properties and Extensions	14

2.7	Redundancy and Failures	17
2.8	Data Integrity	18
3	Application Characterization on a Parallel File System	20
3.1	Applications	21
3.1.1	NPB-BMI Benchmarking Tool	22
3.1.2	OpenAtom	23
3.1.3	FLASH	24
3.1.4	Web Server	24
3.2	Application I/O Characterization	25
3.2.1	Inter-arrival Time	26
3.2.2	Request Size	29
3.2.3	Long Range Dependence	32
3.3	Summary	35
4	Attribute-based Extendable Storage Framework	37
4.1	Storage Policy	38
4.1.1	Redundancy	38
4.1.2	Data Retention and Recovery	39
4.1.3	Performance	41
4.1.4	Security	41
4.1.5	Provenance	42
4.2	ATTEST Architecture	42
4.2.1	Extended Attributes	43
4.2.2	Extendable Plugin Architecture	44
4.2.3	Metadata Handling	47
4.2.4	Migration	47
4.3	Design	48

4.3.1	Attributes and Rules	48
4.3.2	Device Tags	51
4.3.3	Mapping Table	52
4.3.4	Plugins	53
4.3.5	Migration	54
4.4	Results	54
4.4.1	Experimental Setup	54
4.4.2	<i>am-utils</i>	55
4.4.3	IOZone	56
4.4.4	File creation	57
4.5	Summary	58
5	User Space Storage System Stack Modules with File Level Control	60
5.1	Stackable File Systems	63
5.2	File system in User space (FUSE)	64
5.3	Stackable FUSE	66
5.4	Implementation	68
5.5	Results	70
5.6	Summary	73
6	Extendable Storage Framework for Clustered Storage Systems	74
6.1	Implementation	75
6.2	Results	78
6.3	Summary	80
7	Improving Reliability in Clustered Storage Systems using ATTEST	81
7.1	Failure Modes in Clustered Storage	82
7.1.1	Single-node Failure Model	84
7.1.2	Data Corruption	84

7.2	Reliable Clustered Storage	86
7.2.1	Writes Under Failure	88
7.2.2	Reads under failure	90
7.3	Detecting Data Corruption	91
7.3.1	Aligned requests	93
7.3.2	Unaligned requests	94
7.4	Results – Single Node Failures	94
7.4.1	Single Client	97
7.4.2	Multiple Clients	99
7.5	Results – Silent Data Corruption	101
7.6	Results – Reliable Storage using ATTEST	105
7.7	Summary	106
8	Conclusions	108
	Bibliography	111
	Vita	124

List of Tables

Table

3.1	Basic Application Characteristics	25
3.2	Request Sizes for the NPB-BMI application	33
3.3	Hurst Parameter for different workloads	35
4.1	Time to create 10,000 files	59

List of Figures

Figure

2.1	RAID Levels	8
2.2	Parallel file system architecture.	11
3.1	Inter-arrival Time on NPB-BMI ‘C’ Class	27
3.2	Inter-arrival Time on OpenAtom	28
3.3	Inter-arrival Time on FLASH	30
3.4	Inter-arrival Time on Web server	31
3.5	Inter-arrival time on MDS with 16 I/O nodes (Web server)	32
3.6	Variance-time plot for OpenAtom	34
4.1	ATTEST Framework Architecture	46
4.2	File’s Extended Attributes	49
4.3	Rules format	50
4.4	Directory’s Extended Attributes	50
4.5	Device Tags	52
4.6	File system’s block to logical volume mapping	53
4.7	Extent-based mapping	53
4.8	Time to compile <i>am-utils</i>	55
4.9	Read throughput on ATTEST	57
4.10	Write throughput on ATTEST	58

5.1	Stackable file systems	63
5.2	FUSE architecture	65
5.3	Operation flow in <i>sfuse</i>	67
5.4	Throughput comparison for different file system setup	70
5.5	Compile time with ATTEST setup	72
6.1	File's Extended Attributes on Clustered ATTEST.	76
6.2	ATTEST framework incorporated within PVFS	77
6.3	Write throughput comparison of ATTEST on PVFS	78
6.4	Read throughput comparison of ATTEST on PVFS	79
7.1	Parallel file system architecture.	83
7.2	Handling of I/O operation using dirty region database	89
7.3	Storing CRCs in PVFS's <i>bstream</i> files.	93
7.4	Writes on Lustre file system	95
7.5	Reads on Lustre file system	95
7.6	Writes on PVFS file system	96
7.7	Reads on PVFS file system	96
7.8	Write performance with 8 clients on Lustre file system	100
7.9	Write performance with 8 clients on PVFS file system	100
7.10	Write bandwidth using IOR (with sync,single client)	102
7.11	Read bandwidth using IOR (without sync,single client)	102
7.12	Write bandwidth using IOR (without sync,8 clients)	103
7.13	Write bandwidth using IOR (with sync,8 clients)	103
7.14	Data Integrity Checks with ATTEST Framework	105

Abstract

The total amount of information stored on disks has increased tremendously in recent years with data storage, sharing and backup becoming more important than ever. The demand for storage has not only changed in size, but also in speed, reliability and security. These requirements not only create a big challenge for storage administrators who must decide on several aspects of storage policy with respect to provisioning backups, retention, redundancy, security, performance, etc. but also for storage system architects who must aim for a *one system fits all* design. Storage policies like backup and security are typically set by system administrators for an entire file system, logical volume or storage pool. However, this granularity is too large and can sacrifice storage efficiency and performance – particularly since different files have different storage requirements. In the same context, clustered storage systems that are typically used for data storage or as file servers, provide very high performance and maximum scalability by striping data across multiple nodes. However, high number of storage nodes in such large systems also raises concerns for reliability in terms of loss of data due to failed nodes, or corrupt blocks on disk drives. Redundancy techniques similar to RAID across these nodes are not common because of the high overhead incurred owing to the parity calculations for all the files present on the file system. In the same way, data integrity checks are often omitted or disabled in file systems to guarantee high throughput from the storage system. This is because not all the files require redundancy or data protection mechanism, and achieving higher throughput outweighs the need to have these computationally expensive routines in place.

In this thesis, we begin by studying the I/O access patterns of different applications that typically use clustered storage. The study helps us understand the application requirements from a file system. We then propose a framework for an attribute-based

extendable storage system which will allow storage policy decisions to be made at a file-level granularity and at all levels of the storage stack, including file system and device managers. We propose to do this by using a file's extended attributes that will enable different defined tasks via plugins or functions implemented at various levels within the storage stack. The applications can set extended attributes for their files, or directories, and extract a complete content-aware storage functionality from the storage system stack. We present a stackable user-space file system which will make it easier to write and install these plugins. Using our stackable file system technique, these plugins can be programmed in user-space and mounted by non-privileged users. We also provide two scenarios where our framework can be used to provide an overall improved performance for a reliable clustered storage system.

To my family

Chapter 1

Introduction

A data storage device such as a hard-drive or a flash-based drive, that is directly connected to the computer is the most basic example of a storage system. However, these storage solutions are limited to a single machine and cannot be scaled. Improvement in network speeds and increasing demand for data sharing has brought several newer ideas forward by moving the data behind the network. Network file servers are machines that are attached to a network and allow its local file system to be visible to any number of client machines connected to the same network. Several file systems have emerged to distribute data over the network including NFS [113], CIFS [27], AFS [95], Sprite [101], and Coda [114]. These file systems provide users with complete network transparency in that the users are able to make changes to a file as if it is present on the user's local machine. These file systems are typically set by the system administrators who decide on storage policies such as redundancy, security, archiving, etc. All users on these file systems follow the same data storage policy that is set by the system administrator. These policies are based on either some priority, or simple user-based heuristics, or on the regulations put into practice by the institution where the data is stored. Further, these policies are typically set for an entire file system, logical volume or the storage pool. However, this granularity is too large and can sacrifice storage efficiency and performance – particularly since

different files have different storage requirements.

1.1 Motivation

As a motivator for this work, consider a shared network file system like NFS or CIFS accessed by multiple users for general purpose data storage. Some users may be using the file system as a scratch drive and do not really care if the data is lost because of a disk crash or system failure. Other users may expect the file system to retain multiple copies of the data either through versioning or snapshotting, and also to provide very high reliability. Some users may want their files encrypted automatically by the file system, while others may want their data to be verified with data integrity checks. The default choice for a system administrator is the most conservative choice – high reliability, continuous data protection, with data integrity checks and full encryption enabled. Overall performance may potentially suffer because of the need to be overly conservative. Consider another case where different scientific applications use parallel file systems like PVFS [21] or Lustre [15] to store different kinds of data. Some applications dump checkpoint data at frequent intervals without performing any reads from the drive, while others may use the file system for reading input data and writing several intermediate logs and result files. Not only that, some applications run for several days while some run only for a few hours. The system administrator must thus provide protection mechanisms such as data redundancy and data integrity checks in the file system to protect the data belonging to the applications which run for a longer time. This is necessary, despite the high overhead for providing such features, to avoid restarting the application from the beginning in case of a failure or corruption. On the other hand, providing such capabilities will also impact the storage system’s throughput negatively for applications that have a short run-time and can be restarted without trouble or significant loss of time. These applications do not necessarily require strong data redundancy or data protection mechanisms and

the high throughput overheads incurred because of them. The system administrator thus has to decide to either enforce such expensive policies for all files on the storage system, or simply leave the data unprotected for all applications instead of having a mechanism to filter files with specific requirements and limit the feature overhead only to them.

Apart from the variety of storage policy options available to an administrator, another motivator for this work is the presence of a wide range of storage devices with very different properties in the market and the choice of different system configurations possible with them. New emerging technologies such as flash translation layers (FTLs) that allow random accesses several magnitudes faster than traditional magnetic drives are gaining popularity in large storage systems. Traditional RAID techniques like striping, mirroring, and parity are other ways where disk access times vary significantly and are preferred depending on the workload. These diverse storage devices can all be collected and placed under a logical volume manager such as Linux’s LVM [28, 68] and viewed as a single device. However, doing that also hides the ability of applications to take advantage of their individual features. For example, an application with a random-seek read-only workload can use flash drives to store the input data while storing the log files, checkpoint files, or intermediate output files on a magnetic disk. However, currently, there is no way to tell a volume manager to place the data blocks belonging to a particular file on a particular drive. To deal with this problem, we require a means for applications to transfer some knowledge about the data to the file system and the underlying devices. The approach should relay the requirements to all the layers in the storage stack, which may be present locally through a direct connected storage device, or, across the network in case of a network-based file system.

We believe that selecting storage policies at a large granularity or hiding device properties by representing multiple storage devices as a single logical volume with

the help of device managers limits many optimizations that could be done to make more efficient use of the available storage. As storage size and administration costs go up, a single policy model to fit all users and file characteristics is not practical and a different approach is needed. In this work, we hope to motivate a case to consider storage policies at the file level and enable efficiencies at multiple levels of the storage hierarchy in a clustered storage system. Administrators can make global decisions about file storage policies based on file classes – for example for all `.c` files, all `.o` files or all `.mp3` files. In addition, end users are given the capability to make file specific, or directory-based, storage policy management decisions based on their inherent understanding of file’s characteristics. Thus, both system administrators and users have input into the storage policy decisions on a particular file or directory. This work will also present an extendable mechanism to enable what has been called application-aware storage [2] as a way to provide smarter storage and lead to reduction in administrative time and cost of ownership.

In this thesis, we first study the I/O access patterns of applications that typically take advantage of clustered storage systems. The study is used to observe the application’s behavior and understand the requirements from storage systems. We then provide an attribute-based extendable storage framework called ATTEST for these clustered storage systems. This framework provides applications, or users, a way to communicate their data storage requirements in terms of performance, redundancy, security etc. to the storage system stack. This can be done by simply adding the requirement in a file’s extended attributes. The ATTEST framework also has provisions to introduce stackable plugins in the storage stack. The kernel-space plugins can be written using File System Translator language FiST [140], while the user-space plugins can be written using FUSE-like stackable user-space file system *sfuse* (Chapter 5). These plugins can be added to the system, either on the client, or on the storage nodes which can then be enabled, or disabled based on the attributes set by the applica-

tion. Applications can use this to take advantage of all available features within the storage system from a single mount point instead of having several mounted devices on the system. The ATTEST framework provides applications an attribute-based access and control of the storage system’s internal functionalities. This framework fully leverages the capabilities of a parallel file system and allows using a variety of storage devices present in the system. As an example, we show how this framework can be used to provide fault tolerance or data integrity solutions for important files in the system without placing any significant overhead on the remaining files.

1.2 Thesis Outline

This thesis is organized as follows. Chapter 2 provides some background and earlier work done in this area of research. Chapter 3 studies the I/O access patterns of applications using parallel file systems. Chapter 4 describes our attribute-based ATTEST framework in detail and Chapter 5 explains the stackable user-space plugin architecture. Chapter 6 describes how ATTEST framework can be extended to clustered file systems. In Chapter 7, we provide examples to show how ATTEST framework can be used to provide redundancy and data integrity solutions in clustered storage systems. We end with a chapter on conclusions and our plans for future work.

Chapter 2

Background and Related Work

In this chapter, we provide some background on storage systems and also discuss related work done previously in this field.

2.1 Storage Systems

A data storage device is a device that is connected to a machine for storing user's data persistently. Some examples of storage devices include tapes, hard disk drives, CDROMs, flash drives etc. Of all the storage devices, hard disk drives are in most common use. A file system is an operating system layer that uses these storage devices to provide a persistent medium for applications to store its data. Various operating systems support many different types of file systems. Examples of file systems that can be installed on these devices include EXT2/3/4 [7, 91], JFS [74, 44], XFS [72, 124], ZFS [122], HFS [71], etc. These are single node file systems and are local to the machine. They cannot be shared with users who are not present on the system. Specific file systems known as network file systems or distributed file systems allows multiple users to access files from multiple hosts that share the same network. In these file systems, while the clients do not have direct access to the underlying storage device, an interface similar to a local file system and transparent to the user can be set up, irrespective of the kind of network or file system protocol

used. NFS [113] and CIFS [27] are examples of distributed file system protocols that can be used to share local file systems across the networks.

2.2 Redundant Arrays of Inexpensive Disks (RAID)

Hard disk, because of their mechanical nature, are usually the least reliable component in a computer system. Redundant Arrays of Inexpensive Disks (RAID) is a technology that involves two or more disks to provide storage reliability through data redundancy [102]. RAID is based on the idea of dividing data across multiple low-cost disks instead of storing it on a single large expensive disk. These disks are seen as a single logical unit using a special hardware, or a software. RAID arrays are designed based on three key concepts: mirroring, striping, and parity-based error correction. Figure 2.1 shows some of the popular RAID organizations.

In striping, the data is distributed across all the disks present in an array without any redundancy. This arrangement is also referred as RAID-0. It is typically used for improving the throughput of the system. All I/O operations can be performed in parallel across all the disks in the array. However, the failure of any disk in the array will result in the loss of all data.

Mirroring, or RAID-1 refers to complete replication of the data across multiple disks. If one of the disk fails, the data can be recovered from the other disk which contains exactly the same copy of the data. The cost of recovering the data is very low since the data is already present in its complete form. However, the cost of maintaining such RAID arrays are very high due to the space utilization factor of only 0.5.

Parity-based error correction codes are used to provide an intermediate reliability solution to redundancy and performance. Parity is generated for every stripe of data and stored on a separate disk. Thus, if any of the data disks fail, the data can be regenerated using the parity. There are many different configurations possible

D00	D10	D20	D30	D40
D01	D11	D21	D31	D41
D02	D12	D22	D32	D42
D03	D13	D23	D33	D43
D04	D14	D24	D34	D44

(a) RAID-0: Striping

D00	D00'
D01	D01'
D02	D02'
D03	D03'
D04	D04'

(b) RAID-1: Mirroring

D00	D10	D20	D30	P0
D01	D11	D21	D31	P1
D02	D12	D22	D32	P2
D03	D13	D23	D33	P3
D04	D14	D24	D34	P4

(c) RAID-4: Parity

D00	D10	D20	D30	P0
D01	D11	D21	P1	D41
D02	D12	P2	D32	D42
D03	P3	D23	D33	D43
P4	D14	D24	D34	D44

(d) RAID-5: Distributed-parity

Figure 2.1: RAID levels: Data blocks are striped or mirrored across different devices present in the RAID array. P_0, P_1, \dots refer to the parity blocks.

depending on where the parity is stored. RAID-4 and RAID-5 are two formats that are frequently used.

2.3 Classification of Storage Systems

Depending on the way the storage devices are accessed from a system, they can be classified under three categories – Direct Attached Storage (DAS), Networked Attached Storage (NAS), or Storage Area Network (SAN). This section will touch briefly on each of them.

2.3.1 Direct Attached Storage (DAS)

Direct Attached Storage (DAS) is the simplest storage solution in which the storage device is directly connected to a server or a workstation through host bus adapter (HBA). These devices typically include tape, hard drive, or a flash-based device. They do not have network present between the system and the data storage device. DAS connections are made through one of many popular protocols—ATA, SATA, eSATA, SCSI, or Fibre Channel. Direct attached storage are however limited to a single machine and thus cannot be scaled or shared with other servers without having interfaces that will allow multiple concurrent accesses present on the device.

2.3.2 Storage Area Network (SAN)

Storage Area Network (SAN) is an architecture where remote storage devices such as disks or tapes appear as locally attached devices to the operating system. Partitions on these devices are visible as virtual hard drives with a unique logical unit identification number (LUNs). Operating system usually connect to these devices using SAN protocols like SCSI, Fibre Channel, iSCSI, ATA over Ethernet (AoE), HyperSCSI etc. and install their own file system on these virtual drives.

2.3.3 Network-Attached Storage (NAS)

Network-Attached Storage (NAS) is a file-level data storage connected to a network and shared across clients present on the network. It provides both, storage and a file system. This is different from SAN which provides a block-based storage and allows the client to install its own file system. NAS servers are dedicated machines that only handle file system requests, thus provide very high throughput. This throughput however depends on the speed and traffic on the network. NAS servers typically provide a NFS or CIFS based file system interface.

2.4 Clustered Storage Systems

A clustered system is a group of independent compute nodes capable of distributing workload among itself and managing transfer of workloads between the nodes. For large clustering systems, a shared storage architecture such as storage-area network (SAN) is necessary. These clustered storage solutions are distributed in nature and typically allow several, in the order of thousands, simultaneous connections. Clustered storage is comprised of low-cost, low-performance and less-reliable storage servers which distribute data and work with appropriate redundancy and reliability algorithms among itself to provide a comprehensive solution to high-performance, high-reliability storage system. Each individual node is capable of maintaining its own form of data storage, such as RAID, SSDs, tapes, etc. These storage systems are also highly scalable. These systems are in contrast to the classic enterprise storage model which are monolithic and very expensive [30].

There are many parallel storage file systems which have been developed recently and are in common use [65, 46, 64, 125, 131]. Most of these file systems are based on the idea of separating the metadata from the data. Keeping the metadata separate removes the management overhead such as the file type, ownership, permissions and directory hierarchy, thus giving the user direct access to the data. In order to operate

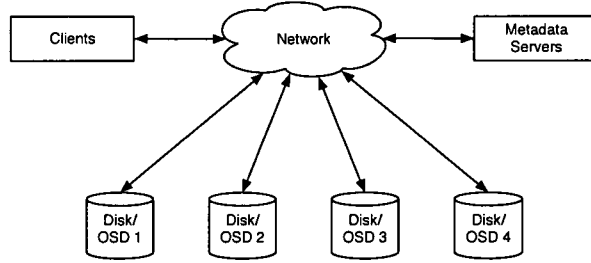


Figure 2.2: *Parallel file system architecture.*

on a file, the client must first obtain the file metadata from a metadata server. This metadata information will include the location of the file data, i.e. information about which storage servers stores which part of the file. These file systems can achieve high throughput by striping the data across many storage servers. Metadata is spread across one or more metadata servers and many clients, all of which are connected by a shared network, perform file system activities as shown in Figure 2.2.

In most parallel file systems, the data organization unit on the storage nodes is in the form of an object. This is different from a local disk-based file system where the organization unit is a block. An object comprises the data (or subset of the data) of a file but its allocation is managed by the target device and a metadata server. Each target device can be considered as a smart disk, but the key differentiating feature of an object-based architecture is the metadata server that is separated from the data targets.

Typical examples of file systems that are deployed to manage large clustered storage systems include Parallel Virtual File System (PVFS) [21], Lustre [15], PanFS [125], IBM’s GPFS [116], Google File System [64], etc. In this thesis, we primarily work with two file systems—PVFS and Lustre.

2.4.1 Parallel Virtual File System (PVFS)

Parallel Virtual File System (PVFS) is an open source implementation of a parallel file system that has enabled clusters of high performance PCs to address applications with fast I/O requirements [21]. PVFS consist of three components – a compute/client node, a metadata server node and storage nodes. A compute node can serve as a metadata server, or a storage node, or both. It is based on a client/server architecture where the server daemon and client side libraries are all present in the user space. The files are striped across nodes based on the rules set by the user. PVFS maintains file metadata using a BerkeleyDB database and file data in a directory tree, both present on the node's local file system. This keeps the design of PVFS simple and provides easy portability across a variety of systems.

PVFS provides multiple application programming interfaces including UNIX/-POSIX and MPI-IO. MPI-IO is part of the standard programming interface for parallel applications, the message passing interface MPI [16]. The MPI layer calls PVFS's functions directly to access data. Using MPI-IO shows a great benefit in comparison to the POSIX interface, in that the applications can extract the underlying file system parallelism which is usually concealed by the UNIX/POSIX-based file systems. MPICH2 [17], which is the most commonly used implementation of MPI can be easily configured to support PVFS using ROMIO [23], a high-performance input/output implementation for MPI. ROMIO libraries, which are linked with MPICH2 and run on a variety of parallel architectures, provide an abstract I/O device (ADIO) [1] layer for PVFS. This ADIO layer provides the link between MPI and parallel file systems like PVFS. All I/O requests made using MPI libraries are sent directly to the PVFS file system. PVFS also provides a kernel module for integration with Linux's VFS to run applications which require a POSIX interface to the file system. The kernel module, however, is dependent on the version of Linux kernel present on the system and requires loading the module before use.

2.4.2 Lustre File System

The Lustre file system is another open source implementation of parallel file system available from Sun Microsystems [15]. It is comprised of three main components, all present as a loadable Linux kernel module – an object storage client (OSC), a metadata server (MDS), and an object storage target (OST). Similar to PVFS, a client, in order to operate on a file must first obtain the file’s metadata from the MDS. This metadata information will include the location of the file’s data, i.e. target information corresponding to each object, as well as the authorization token to access those objects. Once this information is acquired, the client need not approach MDS for any subsequent operations. Since all components of Lustre are present within the Linux kernel, it can take advantage of Linux kernel’s VFS cache and other kernel optimizations. It uses a distributed locking mechanism to maintain concurrency of the data. With the metadata separated from the actual data, and with this data striped across several nodes, the Lustre file system can easily obtain very high throughput.

2.5 I/O Characterization

With disks being the slowest component in the storage system, it is important to know and understand the nature of I/O requirements from high-performance applications. An application’s I/O access pattern gives us an insight of how the storage nodes behave under real workload and how tuning certain file system’s parameters can change the system’s overall throughput.

There has been significant early work on I/O analysis of parallel applications [90, 107, 108, 93, 83, 54]. For the most part, these studies have been done on message passing distributed memory parallel systems such as the Intel Paragon, Intel iPSC and Thinking Machines’ CM-5. Much of this work is several years old and may not represent recent usages of parallel systems as usage of MPI and MPI-IO has increased. In [52, 53] Corbett et al. provided mechanisms in their Vesta file system to dynamically

change the partitioning of data based on the intended access pattern to enhance the performance. Their work was based on access patterns of applications existing at that time and may not match with recent developments in the field. Smirni et al. in [120] studied the evolution of I/O access patterns of scientific applications and identified patterns that belonged to the application and not artifacts of optimizations made by the developers.

Other recent works have looked at parallel I/O on the Cray XT [136]. Alam et al. in [33] characterized scientific workloads on multi-core processors, but without considering the effects of the file system. Their work evaluated the performance of the system based on the bandwidth and latency of the operation when run in different configurations. Kunkel and Ludwig [85] evaluated the performance of PVFS2 architecture under synthetic workloads concentrating on software layers of PVFS2 instead of disk I/Os. In 2007, Ahmad [32] published results which included disk I/O performance, block size and locality details, however they used synthetic workloads over ZFS and UFS file systems.

Our work presented in Chapter 3 differs from the previous work in that we evaluate the workloads from the parallel file system point of view, specifically looking at the effect on individual I/O nodes under both synthetic and actual applications. We present details on how different configuration of servers on storage nodes effects the frequency of request on the file system, and how much variation we get on those parameters by changing the number of processors.

2.6 File System Properties and Extensions

On single node systems, logical volume managers such as Linux's LVM help administrators manage disk drives and other similar storage devices as physical volumes under a single large volume. The volume could then be partitioned in different sizes for mounting as different file systems, or be used as a single partition under a single file

system. Although this technique of using several devices under a single volume simplifies storage management, it hides the storage mechanism under it, thus preventing the file system or higher level applications from becoming aware of the characteristics of the underlying storage. Similarly, the underlying storage is also unaware of the characteristics or the nature of the data that it is storing. The ATTEST framework (Chapter 4) allows the data characteristics, using attributes, to be pushed all the way to the underlying storage layer so that the system can distribute or optimize the location of the data storage if it can, all under a single mount point.

Sun's ZFS file system [122] subsumes much of the logical volume management functionality into the file system. This allows the file system to make striping and redundancy decisions at a higher level and on a file-by-file basis. However, the storage system itself is not made aware of the file characteristics, and it is not possible to enable particular features such as compression or encryption through plugins. UnionFS [105] is a popular mechanism to merge different file systems. UnionFS allows a user to merge different directories across different file systems into a single mount point. The unification under a single file system is independent of the underlying file systems. UnionFS forms the back-end for Live CDs of variety of Linux's distros. UnionFS gives the user and applications extreme flexibility in managing files from a single mount point. However, the placement of files is based on the order of mounts, or access permissions of the underlying file systems. Thus, the user does not have the freedom to select the exact file system to place the file.

Some of the other solutions that merge different file systems and also provide the ability to control the location of the data include RAIF (Redundant Array of Independent Filesystems) [78] and UmbrellaFS [63]. Under both mechanisms, the user has the ability to define rules to determine distribution policies. While RAIF provides users with redundancy characteristics over different file systems, UmbrellaFS gives user ability to select partitions based on rules, and each partition in itself could

have independent redundancy devices under it. These mechanisms however do not allow underlying storage systems to be aware of the rules or characteristics. Further, they do not propagate the rules across the network in case of network attached storage devices.

Various researchers have investigated enabling storage management decisions on a file by file basis, but primarily for redundancy policy. For example, RAIF allows individual files to be spread across different file systems with each file having different distribution and redundancy characteristics [79]. Users are allowed to specify desired RAID levels through the use of *ioctl*s [10]. Similarly, many clustered and object-based file systems have taken this approach to redundancy by allowing different files or objects to have different RAID levels [46, 48, 134]. In Lustre, these decisions are made by the user through the use of *ioctl*s. Peer-to-peer file systems also use file based granularity to replicate files based on file popularity or node reliability [84, 112, 57]. Other aspects of storage policy management that have been done at the file level include distributed consistency using user plugins [128], dynamic compression through elastic quotas [141], and transparent encryption using extended attributes [92]. However, none of these methods allow user specified mechanisms across multiple storage policy dimensions as we have suggested.

Application-aware storage has been suggested by Anderson and Griffioen [37]. Their work uses name-based hints to allow applications to specify persistence requirements to the storage system. This approach is similar to ATTEST in that the hints are pushed down to the storage system but is not a general approach for a variety of system policies. ATTEST allows applications to set storage policy attributes as required by the application. For example, in managing a database, the application may have a better understanding of files created by it and its storage requirements rather than the data administrator.

2.7 Redundancy and Failures

Parallel file systems use several storage nodes to stripe data and take advantage of parallelism. However, using more nodes for storage also raises concerns for reliability or failures. A higher number of independent components in the system reduces the system's overall *mean time to failure* (MTTF), thus increasing the down-time frequency. This makes designing a reliable storage system a complex problem. Very few file systems provide redundancy and data integrity checks and those which do, bear a high overhead. Most of the sites simply use the simpler but costly fail-over mechanism as a workaround to add reliability in the storage system. Similarly, disk backups are utilized in cases where one requires device-level data corruption recovery.

Many commonly used high performance parallel file systems such as Lustre and PVFS have ignored reliability issues like storage node failures or data corruption. But as the cluster size increases, these failures will become more common and more important. Distributed storage systems that have used replication include Echo [123], Petal [86], Slice [35] and peer-to-peer file systems such as FarSite [45, 57], Freenet [50], PAST [58], CFS [56], Ivy [97] and Pangaea [112]. Xin et al. in [134] present reliability techniques to handle non-recoverable read errors and drive failures by replicating data. IBM's GPFS [116] provides support for software-based data replication along with on-line fail-over. It uses client-based replication algorithm and hence requires distributed locking schemes. Lock-less mirroring techniques have been proposed by Settlemyer et al. in [117]. Ceph [131, 130] provides redundancy by replicating data between storage devices using primary-copy technique and splay replication. Arpaci Dusseau et al. in [40] proposed the River environment in which they used a modified version of chained declustering [73] to provide data mirroring. However, the problem with replication is the significant overhead costs in terms of storage space. OceanStore [84] uses erasure codes to reduce the storage overhead while providing better availability guarantees, but at the expense of significant computation overhead.

Using parity-based redundancy can reduce the storage overhead, but is expensive in terms of performance in a networked storage system. Among the earliest parallel storage systems to support parity based redundancy was the Swift/RAID system [89]. Swift/RAID used transfer plans that are specified at the application level to determine RAID levels and striping characteristics. In spite of using optimized data transfer plans they were able to achieve write performance that was only half of the non-RAID performance. Log-structured file systems such as Zebra [67] and xFS [36] can amortize the cost of parity calculation at the expense of garbage collection complications. Pillai et al. have designed a modification of PVFS called CSAR which uses a hybrid RAID scheme that mirrors partial stripe writes on the server [103]. These mirrored data blocks can incur significant storage overhead since they are not cleaned up and the scheme still incurs a 20–25% performance overhead for both large and small writes. Amiri et al. have discussed concurrency strategies when developing RAID protocols for shared storage [34]. One of the few works that discussed the availability of the system during failure was by Yu and Vahdat. In [135], they presented *Om* which can achieve high availability in peer-to-peer, wide-area storage system through online automatic regeneration techniques. Their work was however developed for files spread over a wide-area network and required considering network congestion and node overloads. Our work presented in Chapter 7 provides redundancy algorithms for parallel file systems present on local-area networks and do not have to take network latency into account.

2.8 Data Integrity

Significant work has also been done on identifying data corruption in large-scale storage environments [42, 41, 75], each of them concluding that silent data corruption is a real problem. Gopalan et al. in their paper [119] provided a detailed survey of different integrity assurance mechanisms that are in use today to tackle the problem.

However, little work has been done on studying the real cost of providing checks to verify integrity of the data, particularly in the context of a parallel file system. Single-node journaling file systems such as Ext4, JFS and XFS provide user-level tools such as *fsck* to check file system consistencies. These tools are provided to ensure proper system startup after unclean shutdown, by keeping the file system’s metadata consistent, but they do not perform data integrity checks. ZFS [122] and Btrfs [4] are examples of file systems that provide both data and metadata verification by using checksums. The IRON file system is a prototype file system that provides in-disk checksumming [104] to verify the integrity of the data. This work demonstrates that checksums can be applied to local file system without a large overhead; however, the work is limited to direct attached storage.

Most distributed file systems provide integrity checks only for transmitted data. Lustre [46], for example, has provisions for checksum verification for all blocks transmitted over the network. Any corrupt data detected by that mechanism can be requested again from the sender. However, it fails to provide integrity checks once the data is stored on the disk. Ceph [131] provides integrity checks by using Btrfs as its back-end file system. GPFS [116], PFS [121], PanFS [125] and GFS [64] are other popular distributed file systems that provide integrity checks, but only as an option during mount for persistent checks. None of these file systems provide any insight into how expensive checksum computations are, mostly for commercial reasons. Moreover, most of these file systems provide integrity checks only as a mount option although all studies stress the need for having such a feature available by default.

In our work, we use a parallel file system and provide redundancy and reliability for clustered storage system based on the rules set by the user. Our work also can be extended to support and take advantage of variety of emerging technologies like flash-based disks, or object-based storage devices.

Chapter 3

Application Characterization on a Parallel File System

Rapidly increasing processor speeds and the ease with which they can be connected and converted to form a clustered computer environment has given rise to a new world of parallel computing. These parallel machines run applications which operate on the principle that large problems can always be divided into several smaller ones, and then each smaller problem can be solved concurrently [20].

However, disk I/O on these large machines continues to be a challenge. According to Amdahl's system balance rule [70], each MIPS (million instructions per second) must be accompanied by one megabit per second of I/O. However, today's disks are not able to meet this rule especially in light of the modern cluster computing or supercomputing capabilities. To overcome this, parallel file systems such as PVFS [21, 49], Lustre [46] and GPFS [116, 77] are typically used. Scientific computing often requires large applications doing several noncontiguous access of small regions of data [43, 54, 99, 126]. However, the frequency and size of requests vary across applications. Due to the diverse computational and data access patterns of different applications, it is essential that file system designers understand how an application accesses data. The standard approach of striping data in a parallel file systems can become a bottleneck if the file distribution parameters such as stripe size,

caching duration, or prefetch length do not fit the access patterns of the applications. Workload characterization, thus, plays an important role in systems design. It is one of the necessary and most important steps in recognizing the fundamentals of the parallel file system. It allows us to understand the state of the system under different applications. We ran a variety of parallel applications and have collected their I/O patterns on the parallel file system PVFS. In this chapter, we provide an analysis that can characterize these applications and their access patterns.

The rest of this chapter is organized as follows. Section 3.1 provides details of the applications that were run on PVFS and in Section 3.2 we present details on our experiments and results. We provide the summary of our experiments in Section 3.3.

3.1 Applications

Input/output characterization of an application code ideally includes access patterns and performance data from the application, its input/output libraries, file system, and device drivers [54]. Some of these are handled by cache at different levels of the system. Physical disk access patterns provide the ultimate evaluation of system's response. Disk accesses made by the application generally depend on its input/output modules, size of data being handled, temporal spacing and spatial patterns generated from its computation libraries and certain file system's optimizations (eg. prefetching or caching). Monitoring the I/O access patterns gives us the real response of the system under that application and gives the file system designers an insight into the application's requirements. At the same time, it is also a good resource for application developers to maximize the use of file system policies.

Most parallel applications use one of five major models of I/O as described below.

- Single output file shared by multiple nodes by ranges
- Large sequential read by single node at beginning of computation and large

sequential write by single node at end of computation

- Checkpointing of state
- Metadata and read intensive - small data I/O, frequent directory lookups for reads
- Each node outputs to its own file

There are other I/O access patterns, but most applications use one or more of these patterns for the majority of their I/O. In this study, we selected a sample of parallel applications that use one of these forms of I/O in order to represent a wide range of parallel I/O applications. The popular benchmarking tool NPB-BMI uses the single output file model. The scientific application OpenAtom is largely distinguished by its I/O on several input states files during the computation. FLASH, another common scientific application uses frequent checkpointing of state to an output file, and finally, web server usage is characterized by heavy metadata usage.

We shall briefly discuss the properties of each of these applications in this section.

3.1.1 NPB-BMI Benchmarking Tool

The NAS Parallel Benchmark (NPB) (BMI version 3.3) is a parallel benchmarking tool developed by the NASA Advanced Supercomputing Division [76, 133, 18]. This tool was formerly known as BTIO. It presents a block-tridiagonal (BT) partitioning pattern on a three-dimensional array across a square number of compute nodes. Each processor is responsible for multiple Cartesian subsets of the entire data set, whose number increases with the square root of the number of processors participating in the computation. The I/O requirements and verification tests for the tools are as follows. After every five time steps, the entire solution field, consisting of five double-precision words per mesh point, must be written to one or more files. After all time steps are finished, all data belonging to a single time step must be stored in the

same file, and must be sorted by vector component x-coordinate, y-coordinate and z-coordinate respectively. It uses MPI for communication and MPI-IO for I/Os. In our experiments, we ran NPB-BMI’s ‘C’ class of “full_mpio” application which uses collective I/O to combine data accesses of multiple processes into large, regular I/O requests.

We chose this application because it provides an example of a parallel application in which each processor writes into a shared output file the data for which it is responsible, thus potentially contributing to file system fragmentation.

3.1.2 OpenAtom

OpenAtom is a highly scalable and portable parallel application for molecular dynamics simulations at the quantum level [19]. It is written in Charm++ [80] which is a parallel object-oriented programming language based on C++ and was designed with the goal of enhancing programmer productivity by giving a high-level abstraction of a parallel program to deliver good performance on the underlying system. OpenAtom attempts to solve important problems in material science and chemistry. Their approach uses Car-Parrinello ab initio molecular dynamics (CPAIMD) which involves a large number of inter-dependent phases with high-communication overhead including sparse 3D Fast Fourier Transforms (3D-FFTs), non-square matrix multiplies and dense 3D-FFTs. OpenAtom’s I/O requirement consists of a large sequential read at the beginning of simulation to read the input data. At the end of simulation, the output is flushed as a large file thus initiating a sequential write. In addition, there are several intermediate checkpoint flushes which can be controlled using pre-defined parameters. OpenAtom requires POSIX access to file system and hence uses PVFS’s virtual file system interface.

3.1.3 FLASH

FLASH is a block-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [62, 8]. The “Flash” problem is centered on simulating the accretion of matter onto a compact star, and the subsequent stellar evolution including nuclear burning either on the surface of compact star, or in its interior. The computational domain is divided into blocks which are distributed across the MPI processes. A block is a three dimensional array with an additional four elements as guard cells in each dimension on both sides to hold information from its neighbors. The I/O uses HDF5 [29], a higher level data abstraction which allows data to be stored along with its metadata in the same file. HDF5 is built on top of MPI-IO. FLASH flushes its data at checkpoint intervals over multiple checkpoint files and is a completely write-dominated workload. It hence provides us a different perspective of scientific applications.

3.1.4 Web Server

Since the metadata server is a prime component of a parallel file system such as PVFS, we chose a metadata heavy workload for our final application. We used the real web server traces of the 1998 Soccer World Cup web site [26]. The traces were collected on 33 different web servers at four geographic locations. The real measurement was done over a period of 88 days during which it received 1.35 billion requests [39]. However, for our experiments, we ran the trace of 4 peak hours on one of the busiest days observed by the web server. The access rate averaged roughly 800 HTTP requests per second. Unlike the scientific applications that we have discussed above, the web server is very metadata intensive as it requires numerous directory lookups and file opens and closes. Thus, running these web server traces will give us an insight of the impact of real-world heavy metadata workloads on parallel file

Table 3.1: *Basic Application Characteristics*

Application	Files Accessed	Data Accesses	Read %	Metadata Accesses	Run Time (sec)
NPB-BMI	1	30,736	50	1	59
OpenAtom	135	174	91.8	135	297
FLASH	10	480	0	10	934
Web Server	4,965	11,469,846	100	2,433,391	14,524

systems.

3.2 Application I/O Characterization

We conducted our experiments on 16 dual-core AMD Opteron dual-processor machines each with 2GB RAM and a 80GB SATA disk drive. EXT3 served as the base file system on each of these machines.

We modified PVFS (version 2.7.0) to collect traces about each request sent by the client and received by the server. The information which was logged included timestamp, nature of request, size of request (in case of read/write), server number and file ID. This information which was added as a configurable option to PVFS would dump the trace to a separate log file, in a temporary directory. In our experiments, apart from tracing, PVFS was mounted with default options using the TCP protocol.

For the Web server experiment, we took the World Cup trace and partitioned the trace so that the workload was load balanced across the available Apache servers. We configured from 2 to 16 Apache servers and split the requests among them. PVFS was configured with multiple metadata servers with each MDS colocated with an Apache server.

We analyze our results based on three important parameters. They are inter-arrival time, request size and long-range dependence of requests. Table 3.1 also shows other basic characteristics of the applications including number of files accessed, num-

ber of data accesses, percentages of data accesses that are reads, number of metadata accesses, and run time for 16 processors using 16 I/O nodes configuration.

3.2.1 Inter-arrival Time

Inter-arrival time is defined as the time difference between successive I/O requests on a node. It is a useful measure to help identify the load on the storage nodes and determines the required service from them. It can be used to determine the intensity of traffic observed by the node and hence can help design and configure both hardware and software components of the node.

Each application was run with 2 to 16 compute nodes and 2 to 16 I/O nodes, except for NPB-BMI where the number of compute nodes were 4, 9 and 16 since for NPB-BMI the number of processors must be a square. In the graphs in Figures 3.1–3.4, we show the cumulative inter-arrival time distribution seen by all I/O nodes.

As seen in Figure 3.1, NPB-BMI has very frequent accesses – more than 90% of the requests are within 2 milliseconds of the previous request in all cases. We see heavy load on storage nodes in case of 16 I/O nodes with over 90% of requests having inter-arrival time of less than a millisecond. However, increasing the number of storage nodes reduces the frequency of requests. This is because the data is shared between more nodes. It can also be seen that increasing the number of compute nodes increases the frequency of requests, which is because the new processors are sharing the I/Os and hence the storage node sees requests from more compute nodes.

The FLASH application (Figure 3.3) and the web server (Figure 3.4) show similar characteristics to NPB-BMI. For FLASH and the web server, it can be seen that with fewer processors, the frequency is reduced, which is similar to what was seen with NPB-BMI. Thus, as we scale up in processors we must be careful to scale up I/O capacity to match with the increase in computation. But scaling up I/O does not simply mean increasing the number of I/O nodes. As seen in the figures, distributing

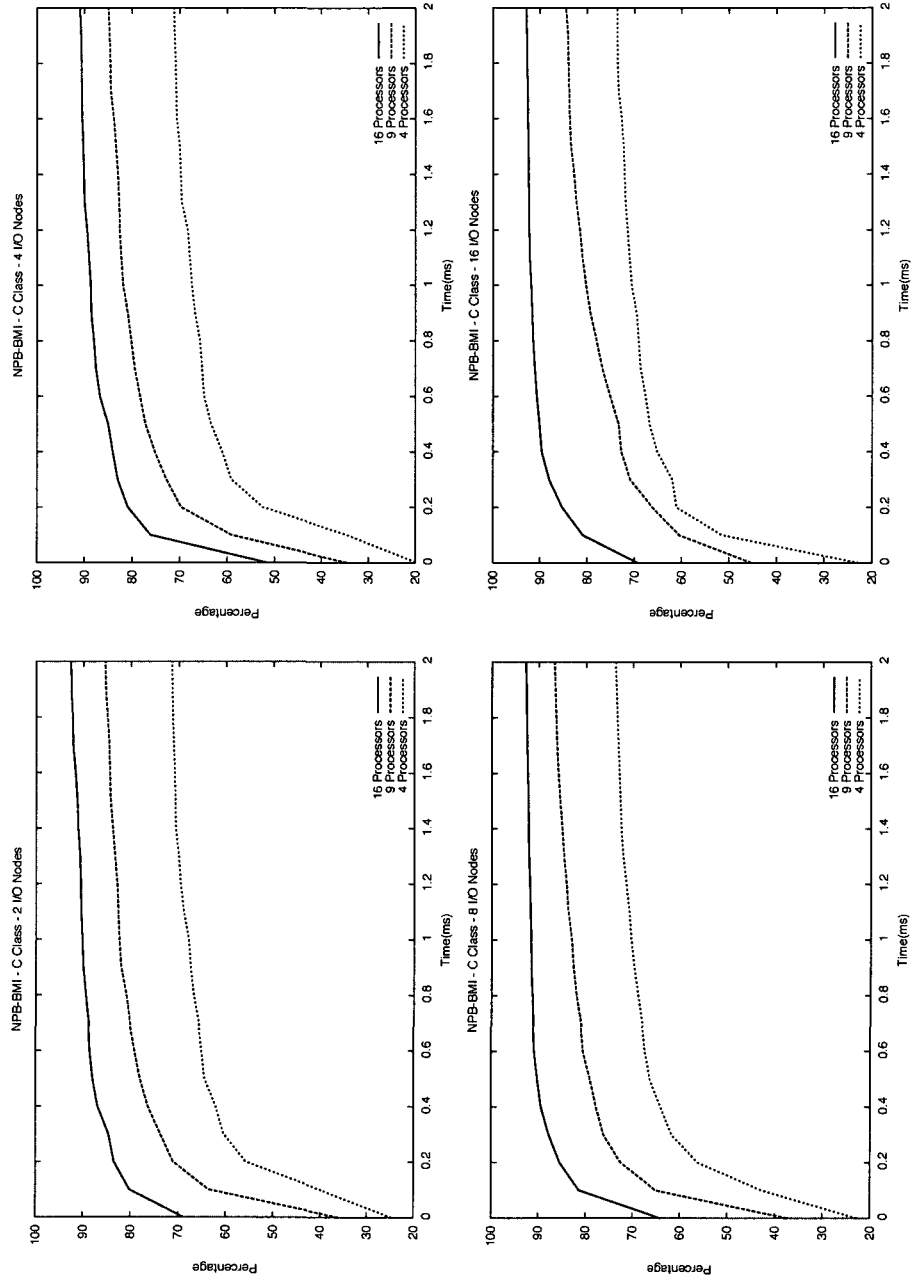


Figure 3.1: *Inter-arrival Time on NPB-BMI 'C' Class*

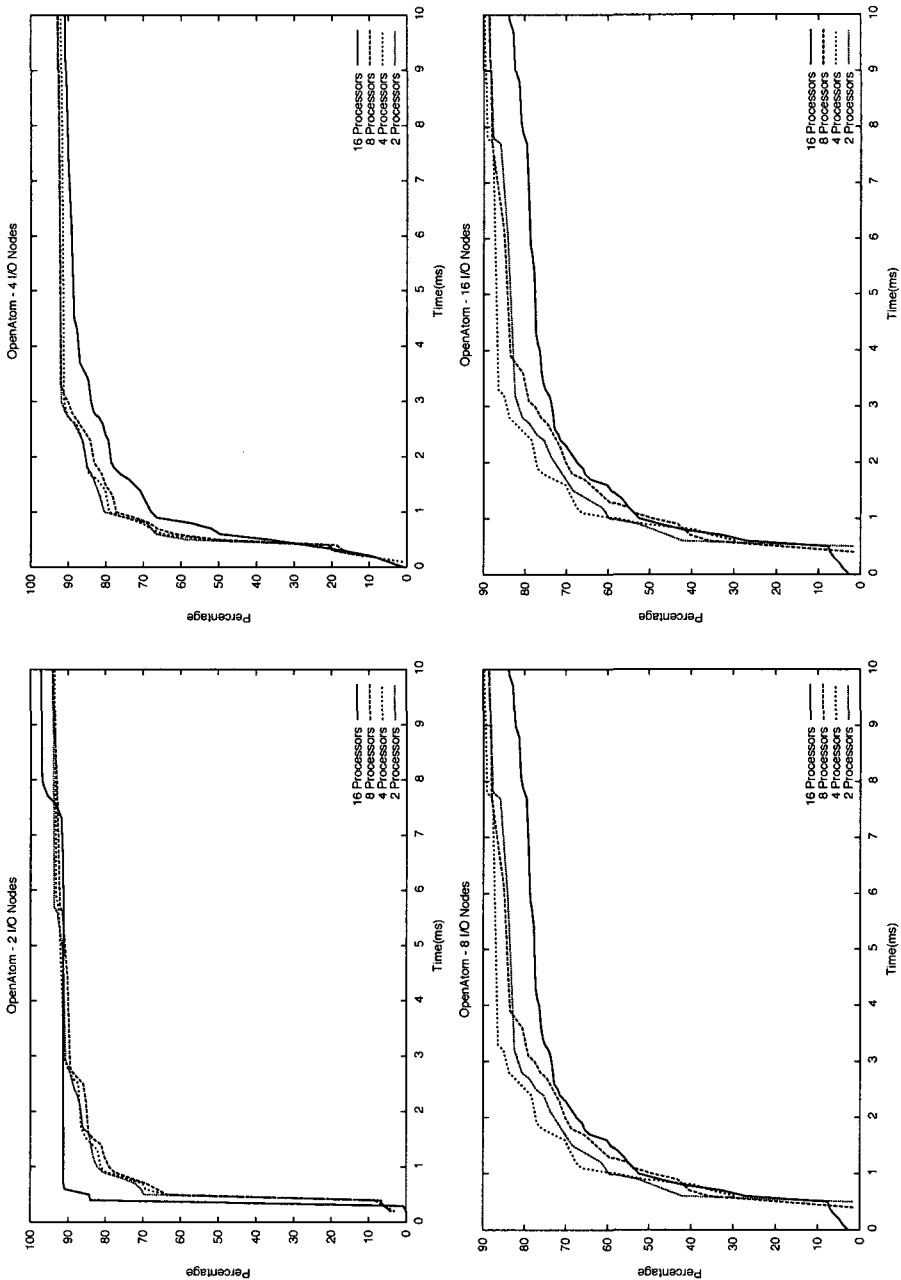


Figure 3.2: Inter-arrival Time on OpenAtom

the I/O across more nodes does not decrease the inter-arrival times because the files are striped across all nodes which causes any read or write to access all nodes. If the latency is being affected because of the frequent accesses, it may be appropriate to change the striping so that files are distributed across only different subsets of the I/O nodes.

OpenAtom, on the other hand, does not show any significant difference in the inter-arrival rates as we change the number of processors (Figure 3.2). Increasing the number of processors reduces the inter-arrival time only by a small factor. This is because OpenAtom does most of its I/O through a single compute node. Although OpenAtom dumps checkpoint data at regular intervals, it is also done through a single compute node. Thus, the number of processors does not change the frequency of the I/O. From Figure 3.2 it is also clear that increasing the number of I/O nodes lengthens the inter-arrival times. This is simply because the data is distributed sequentially across more I/O nodes. As a result, striping across all nodes is a preferred option to increase throughput at the start and end, and latency is not likely to be an issue during the actual run of the application.

Figure 3.5 shows the average inter-arrival time of each metadata server using the web server application. For relatively few HTTP servers, the frequency of access to the metadata servers is not that great, but as we increase the number of HTTP servers the inter-arrival time decreases significantly. With 16 HTTP servers, the inter-arrival time distribution is nearly the same as the data I/O distribution. Even with the load distributed across 16 metadata servers, the each server can see significantly increased activity.

3.2.2 Request Size

In this section, we examine the request size distribution, a measure of the number of blocks in each request sent to the disk (Table 3.2). The larger the request the

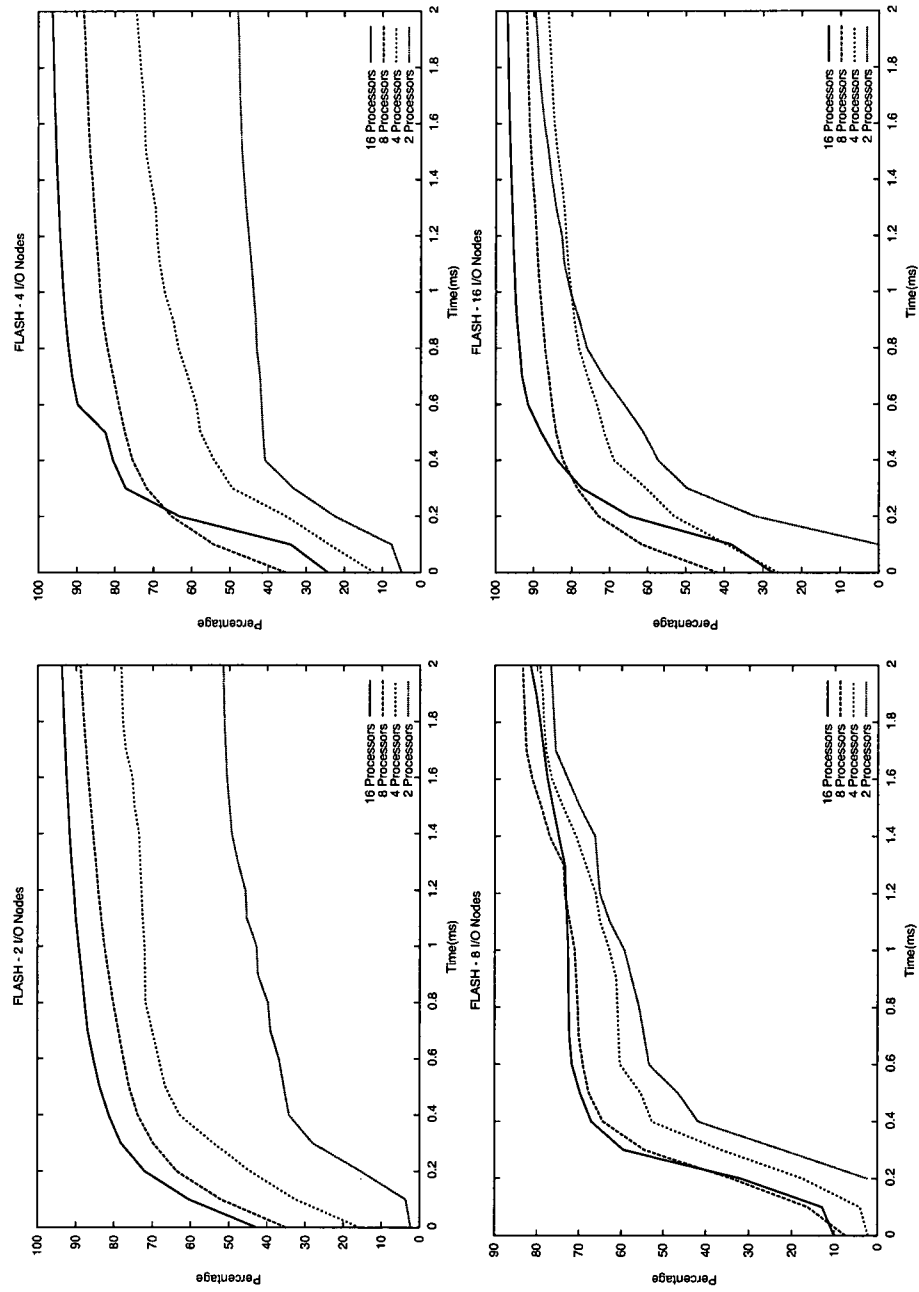


Figure 3.3: *Inter-arrival Time on FLASH*

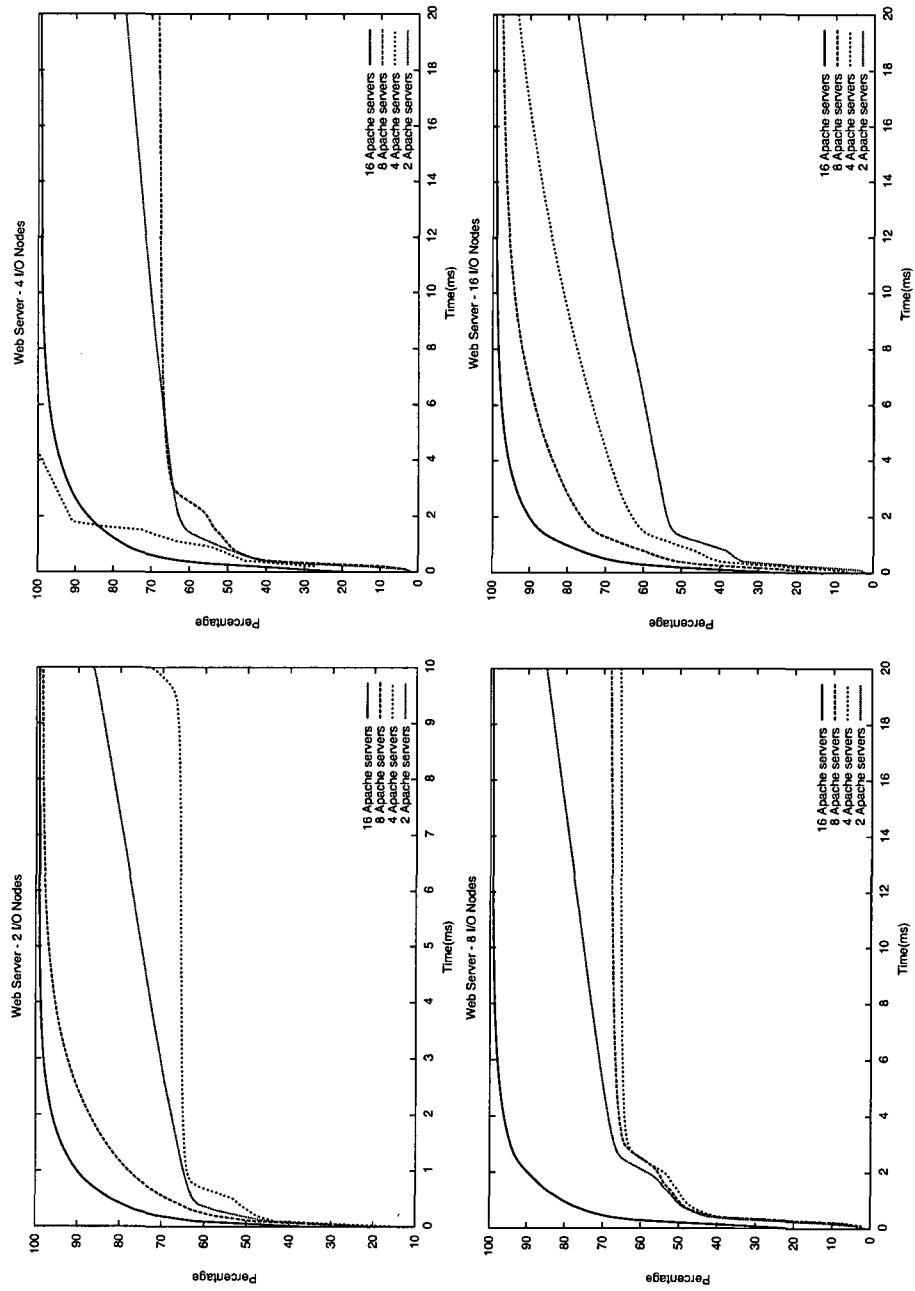


Figure 3.4: Inter-arrival Time on Web server

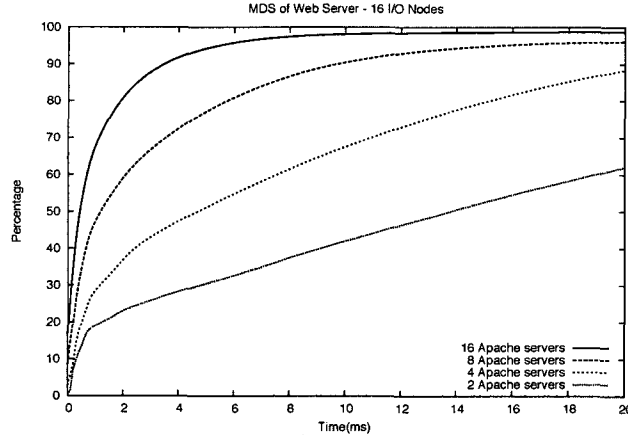


Figure 3.5: *Inter-arrival time on MDS with 16 I/O nodes (Web server)*

more likely that the underlying file system can deliver the parallelism required for performance.

In the case of NPB-BMI, we observed a fixed request size across different storage node counts. The request size however decreased as the processor count increased. This implies that the new processors are sharing the I/Os.

The request size for the FLASH, OpenAtom and web server workloads were fixed at 4K in size. This could be easily explained since all three of them do sequential reads of data through a single processor, they do not do very large I/O. Further, OpenAtom and the web server submit requests using the POSIX interface which uses Linux kernel's VFS and thus limiting the request size to 4K. An increase in VFS's page size limit could possibly help storage nodes see larger requests and at a slightly lower frequency thereby improving overall I/O performance.

3.2.3 Long Range Dependence

Many analytical studies of I/O assume a Poisson model for I/O requests. Poisson models presume that there is no dependence between a request and subsequent requests. However, real I/O does exhibit some long range dependence - i.e. request

Table 3.2: *Request Sizes for the NPB-BMI application*

Application	Number of Processors	I/O Nodes			
		2	4	8	16
NPB-BMI	4	15.4 MB	15.4 MB	15.4 MB	15.4 MB
	9	7.05 MB	7.05 MB	7.05 MB	7.05 MB
	16	3.34 MB	3.34 MB	3.34 MB	3.34 MB

patterns that do not behave like the requests generated by a Poisson process. Markov I/O models that assume a probabilistic dependence also do not accurately reflect real I/O behavior. In our work, we calculate the Hurst parameter to describe self-similarity or long-range dependencies on the traffic generated by the workload as seen on the storage nodes. Self-similar traffic behaves the same when viewed at different degrees of traffic. This information could be used to understand the “burstiness” of requests. We draw on techniques to estimate the Hurst parameter as described in [51].

Let $X = (X_t : t = 0, 1, 2, \dots)$ be a covariance stationary stochastic process with mean μ , variance σ^2 and autocorrelation function $r(k), k \gg 0$. Assume $r(k)$ of the form

$$r(k) \sim k^{-\beta} \quad , \quad k \rightarrow \infty$$

where $0 < \beta < 1$. The process X is called (exactly) second-order self-similar if for all $m = 1, 2, 3, \dots$, $\text{var}(X^{(m)}) = \sigma^2 m^{-\beta}$ and

$$r^{(m)}(k) \sim r(k), k \geq 0.$$

and called (asymptotically) second-order self-similar if for k large enough,

$$r^{(m)}(k) \rightarrow r(k), m \rightarrow \infty$$

holds true.

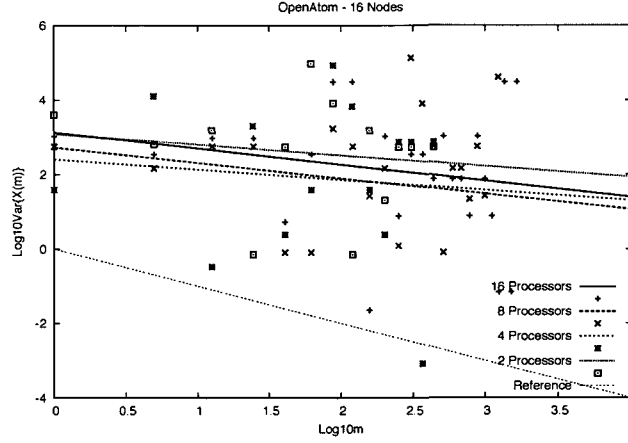


Figure 3.6: Variance-time plot for 4 configurations on 16 I/O nodes for OpenAtom

The Hurst parameter, H is characterization of long range dependence. To calculate the Hurst parameter and determine self-similarity, we first plot $\text{var}(X^{(m)})$ as a function of m . The variance-time (variance vs. m) plot is made on log-log scale. The relationship between the Hurst parameter and β is given by $H = 1 - \beta/2$, where $\beta = -\rho$ from the equation of the line $y = \rho x + b$ for each processor configuration. ρ is the slope of the line. Any process characterized by a slope less than 0 and greater than the reference line exhibits long range dependency and has an H parameter value of $1/2 < H < 1$.

Figure 3.6 shows the variance-time plot for OpenAtom with 16 I/O nodes. The data for each processor configuration series is fitted to determine the $y = \rho x + b$ line. The graph shows the slowly decaying variance of a self-similar series. It can be observed that the slope for all processor configurations has a slope between 0 and -1 . Thus, for OpenAtom, the requests on the storage nodes exhibit Long Range Dependence. We found similar behavior for the other applications and the Hurst parameter for these applications is shown in Table 3.3.

The Hurst parameter, H , takes on values from 0.5 to 1. A value of 0.5 indicates the data is uncorrelated and purely random, while values closer to 1 indicate high degree of

Table 3.3: *Hurst Parameter for different workloads*

Application	Processor Count				
	2	4	8	9	16
NPB-BMI	-	0.8850	-	0.8700	0.8245
OpenAtom	0.7831	0.7932	0.8628	-	0.8552
FLASH	0.6474	0.7660	0.7871	-	0.6013
Web Server	0.9439	0.9831	0.7707	-	0.9438

persistence or long-range dependence. In our work, we obtained the Hurst parameter by calculating the slope of the best-fit line on each configuration. From this, it can be observed that all four applications have self-similar inter-arrival properties on the storage nodes. Thus, the traditional approach of using Poisson models to characterize I/O behavior is not adequate. NPB-BMI and Flash do I/O at regular intervals and OpenAtom does major I/O at the beginning and the end as well as checkpointing at regular intervals. Thus, these I/O accesses are clearly repetitively patterned and that is reflected in the high Hurst parameters. Web server I/O shows significant self-similarity because of the popularity of groups of files. A Hurst parameter value closer to 1 for the web server workload indicates that significant performance benefit could be achieved by using additional memory or caching on the storage nodes.

3.3 Summary

In this chapter, we presented a survey of different parallel applications and their impact on I/Os as seen on storage nodes and metadata servers in a parallel file system. From our study on the inter-arrival times, we see that most parallel applications that do significant I/O during the run increase the I/O frequency as we increase the number of compute nodes. However, scaling I/O nodes alone will cause problems because the increased load is transferred to each I/O storage node in the common case of striping across all storage nodes. Thus, care must be taken with striping as you increase the

number of I/O nodes. Other applications which do significant sequential I/O can and should use striping across as many nodes as possible to increase throughput. Our study on self similarity shows that most parallel applications exhibit significant long range dependencies. This result shows that I/O access models that assume independence or randomness between requests are not valid.

Chapter 4

Attribute-based Extendable Storage Framework

There has been an ever increasing need to store data, but managing this data has become a challenge for system administrators. Storage administrators spend a significant amount of time determining and enforcing storage policy and provisioning storage for these different and often conflicting policies. At the device layer, one has to decide what level of RAID to use to provide required levels of reliability and availability. At the logical volume level, the primary requirement is to allocate or provision sufficient storage capability from a pool of storage buckets for user and application demands. With modern SAN systems, these allocations can be made dynamically and the process is fairly routine to change the amount of storage available in a logical volume. In addition to provisioning and redundancy, file system decisions must be made with regards to quotas, how often to backup systems (daily, weekly, etc.), where to backup data to (tape, disk, off-site, etc.), whether to snapshot, whether to encrypt data, etc.

The difficulty in assessing the different options is that the granularity of storage policy is too large – disk, logical volume, file system, etc. However, it is often the case that these policy decisions should be made on a file or directory basis. In this chapter, we present ATTEST (ATtributes-based Extendable STorage), an extendable storage

system stack architecture that is evolved from stackable file system [69, 139, 137, 140] and object-based storage technologies with attributes-based plugins in order to facilitate a file-based storage policy model. ATTEST is a rule-based architecture which allows the flow of attributes through every layer of the storage system. Availability of storage attributes at all layers of storage stack allows each layer to make its own decisions on the best way to enforce a particular policy. These attributes vary from selecting between different stackable file system modules [138, 38] to selecting between different redundancy levels at the device level.

The remainder of this chapter is organized as follows. Section 4.1 discusses some aspects of storage policy and the inefficiencies of large grain policy administration. Section 4.2 describes ATTEST, the architecture for an extendable storage system stack driven by user-directed file attributes. Section 4.3 explains the design and Section 4.4 shows the results. We end with a section on summary.

4.1 Storage Policy

Every organization is required to have storage management policies and practices developed to ensure that the data is properly managed. Examples of these policies include data retention, data protection from both failure and external attack, etc. Storage system administrators are required to ensure that these are enabled with the best performance and minimum cost to the organization. In this section, we shall discuss how these policies impact today's storage systems and how ATTEST can provide a strong framework to help improve these systems.

4.1.1 Redundancy

For several years, RAID [102] has been the most popular and effective solution for improving the reliability and availability of disk subsystems. However, the different levels of RAID require dedicated extra storage space for redundancy and can also

incur a performance penalty, particularly for RAID5. Often times, there are many files on a storage system that do not require strict guarantees on reliability, so it may be possible to reduce the amount of redundancy required in return for higher performance and less storage overhead. However, current storage systems do not allow these decisions to be made on a file-by-file basis. Consider, for example, the HP AFRAID system which improves the performance of RAID5 by delaying parity calculation, but at the risk of leaving the data unprotected for a small time frame [115]. Preferably, we would like to make this optimization on a file or directory basis, so that important files are not potentially compromised. Similarly, we may wish to turn off redundancy completely for temporary or unimportant files such as web browser caches or audio/video files. This is not possible with existing non-commercial storage architectures since all files in the same file system or disk subsystem must share the same reliability and availability requirements.

4.1.2 Data Retention and Recovery

Storage administrators must always worry about the retention of critical data and the recoverability of that data and, thus how to ensure proper backup mechanisms. Sarbanes-Oxley [24], HIPAA [11], and other regulatory mandates have made data retention a costly and time-consuming endeavor for most storage administrators. Among the different options are backup to tape, backup to disk, snapshotting, and continuous data protection (CDP). One of the primary deterrents to frequent backups is the time and space required for the backup regardless of the mechanism. This is particularly problematic with large file systems with terabytes of data and tape systems not getting appreciably faster in recent years.

However, backup times could be improved significantly if “unimportant” files were not backed up. To give an idea of the number of “unimportant” files on a typical disk, according to a Sun Microsystems storage assessment, less than 41% of a disk

contains useful data [127]. Similarly, up to 40% of user data is regenerable, e.g. object files that can be recreated from source or old tar files [141]. This number probably underestimates the amount of regenerable data, since user data typically does not contain system and application files which can be reloaded from CD or downloaded from the Internet. On UNIX systems, system and application files may already be on a separate / or /usr file systems. Unfortunately, most Windows or MacOS systems by default have a single partition for all files, both system and user. In such cases where we have vast amounts of useless or regenerable data, we can eliminate backup for these files or reduce the recovery guarantees. For example, if 40% of a storage system does not need to be backed up, it can cut the backup time and space by nearly half. Similarly, turning off snapshots or CDP for unessential data can also reduce storage overhead significantly.

With single user PCs, backup software allows users to selectively mark files for periodic backup thus reducing the amount of data being backed up. Most current backup systems do support the capability to do selective backups, but since enterprise backup processes are controlled by central storage administrators, they are not aware of the necessity of specific files, and as a result they are overly conservative and backup everything. The ability to do selective backups is not used since the end user can not contribute the information needed to activate the feature. Some Enterprise storage management software such as Tivoli do allow for user selective backup, but these features are accessed through complex tools distinct from the file system. ATTEST allows these attributes to be set in the normal file system using the same rules to set other storage policies. If users are able to specify in a simple format the important files that require backup as well as frequency of backup and the recovery objectives, storage management tools can then enable tape backup, snapshot, or CDP as required on a per-file basis.

4.1.3 Performance

We have given some examples of how loosening redundancy requirements on certain files can improve storage system performance. In addition to redundancy optimizations, other opportunities for performance tuning are available to storage administrators. At a high level, this tuning involves identifying bottlenecks and introducing load balancing or provisioning more devices. At a lower level, storage administrators can also choose caching/persistence policies, striping sizes for parallelism, block transfer sizes, amongst others. Once again these decisions are highly influenced by the type of file and type of access. For example, temporary files do not have to persist to disk but database files do. User input on the type of file can assist storage systems in determining optimal performance strategies.

Nearline storage provides administrators another option while deciding on policies for managing data. It is used to store data which does not change over time as often, but is not ready to be archived to a tape. Nearline storage architectures provide near-primary storage performance at a significantly lower cost by utilizing disk drives and removable storage (tapes, CDs etc.) in a hierarchy for storing data. A typical example includes large software development repositories which are compiled every night. While the software trees are large, they do not see much change across daily builds [6]. Storage administrators could use knowledge of these properties of data to allocate resources for storing data in a more cost-effective manner.

4.1.4 Security

With modern enterprise systems and broader network connectivity, the security of sensitive data is a major concern. Protecting this data is usually done through authorization mechanisms such as file permissions and access control lists. Typically, the data is stored in the clear on the raw disk, so any one with direct access to the disk, through theft or root privileges, can inspect the data. For users, there are a variety

of tools available to encrypt and decrypt data including the standard UNIX `crypt` and products such as PGP. However, these methods do not scale to large systems and are not transparent to applications. In response, there has been work in volume and file system level encryption so that the decryption is invisible to the application. However, the encryption granularity is so large that significant computation is wasted on the encryption of files that don't need to be encrypted.

We performed an analysis of a subset of the University of Connecticut's engineering server's user directories and found that only 19% of the user files had user-only read permissions. Moreover, those files only comprised 10.7% of the storage space. If these files serve as an upper bound approximation of files that users would want encrypted, we can reduce the encryption costs by roughly 90% with user input. Microsoft's Encrypting File System (EFS) provides the capability to encrypt on file-by-file basis such that the process is application transparent.

4.1.5 Provenance

Provenance-aware storage systems (PASS) [96] can also benefit from the ATTEST architecture. In PASS systems, the lineage or complete history of a file is automatically stored and managed. This includes data such as environment variables of the system during compile, macros defined in compiler, etc. While it is important to have provenance available for some files, not all files may require provenance. Preserving provenance information for all files requires large amounts of storage space which is unnecessary and wasteful. The ATTEST system can enable PASS for just the necessary files.

4.2 ATTEST Architecture

In the previous section, we gave examples of improved redundancy cost in terms of storage overhead and performance but to the detriment of availability or reliability

in a particular aspect. One can use static means to determine the importance of availability or reliability for different files, but ideally we would like to determine them more dynamically. The AutoRAID system from HP uses this concept as it dynamically moves blocks from high performance mirroring to lower performance parity storage and vice versa [132]. As a side effect, it ends up moving blocks from higher reliability to lower reliability and vice versa.

The difficulty with dynamic mechanisms is that the only hint with respect to the importance of a file is the frequency of access – in the sense that files that are frequently accessed are presumed to require higher performance and in the case of AutoRAID also higher levels of redundancy. In practice, this is not often true. For example, application files are accessed frequently and require high performance but they do not require high levels of data protection in a non 24x7 environment, since they can easily be re-installed. Thus, instead of mirroring we may want to put these files on a RAID0 system. An analysis of a large scale network file system in a corporate environment showed that the majority of opened files were of an unknown type [87]. The study also revealed that most file types did not have any common access patterns. This implies that the administrator cannot make decision simply based on file types. The only reliable source for importance of files is the user who is able to make judgments of cost in terms of disk space and performance vs. availability, reliability, and recovery guarantees.

4.2.1 Extended Attributes

The natural mode of expression of a system administrator or user's wishes to the file system is by setting attributes for a file or directory. Modern file systems support extended attributes which can convey features other than standard file metadata. We propose that these extended attributes be used to specify storage policy decisions such as reliability and availability levels, recovery times, backup intervals, performance

characteristics, etc. However, since many of these decisions are implemented at layers of the storage system that are far removed from the file system, it is necessary to push this attribute information down the storage system stack. With current virtualized storage systems which abstract away much of the underlying storage architecture, this may be difficult, but it is absolutely necessary in order to fully expose to the user the costs and benefits of each reliability choice.

With these per-file attributes, it becomes possible to implement storage policies on a much smaller granularity. Some files could thus be encrypted with high reliability while other files may have no redundancy guarantees at all. The key is that the tradeoffs are made visible to the user so that the user can control the choice. Using extended attributes gives user the freedom of moving files within and across systems using ATTEST without worrying about the policies, since these extended attributes become part of the file's metadata.

4.2.2 Extendable Plugin Architecture

Currently, operating systems do not expose the attribute information at all levels of the storage stack. The ATTEST architecture sets up a flow of plugins along the storage stack so that the user defined policies are available from the file system layer to all storage layers of the kernel, including the logical volume manager, device drivers, and the disk layer. These policies are then used by the lower layers to make intelligent decisions such as defining appropriate redundancy and availability policies. These attributes could either be set by the user individually or by rules based on file name, *uid*, or other metadata. By doing so, the user has more flexibility to decide on the best match of reliability and performance that the file might require.

With object-based disks and intelligent disks potentially appearing commercially in the near future, these policy attributes can be passed all the way to the disk layer as object attributes. In active storage systems or object-based storage systems the

plugin stack can be expanded to the object store layer as well. OSD architectures easily support expanded metadata that could contain policy information at the object level. With the use of intelligent object storage targets, the policy decisions can then be pushed to the object store as well. We have begun work with the PVFS [49] file system to support such policy management.

The basis of the ATTEST architecture is to use plugins to implement the various policy functions if the underlying storage does not provide the feature. The plugins are arranged at the top level of the stack as shown in Figure 4.1. The plugins are enabled or disabled based on the extended attributes for a file or directory. The extended attributes also include options that can parameterize the plugin policies on a per-file basis. As discussed earlier, different files on a single file system requires different levels of redundancy, backup policies, performance etc., and this can be achieved through use of stacked arrangement of plugins of different policies within the file system. Each plugin functions independent of other plugins and the flow of operation would be determined dynamically based on the attributes set for the file (by the user/application).

For example, a file that has been marked for retention may enable the compression, backup, and encryption plugins assuming the underlying storage system does not support these features. Setting respective attributes on the file can guide the *backup* plugin to choose the backup policy that is to be enforced – tape or disk backup, daily or weekly, etc. Performance critical files could be cached by a *cache* plugin and saved on a high-performance storage device. RAID at different levels, or single large-capacity disk storage could be chosen based on the importance and size of file. Exposure to so many combinations of policies using attributes and plugins gives administrators and also application designers several benefits to make use of.

The preliminary implementation of ATTEST draws on the FiST [140] stackable file system technology to provide the plugin stack. As the file is operated on, control

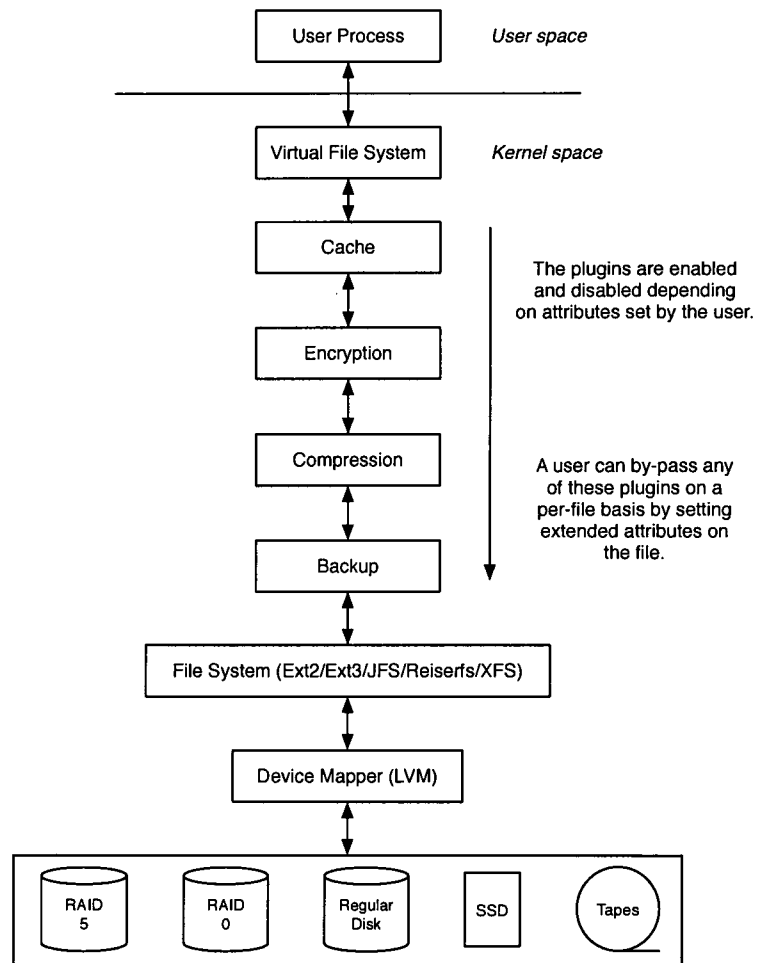


Figure 4.1: *ATtributes-based Extendable Storage (ATTEST) Architecture. The different plugins shown inside dotted box can be enabled or disabled dynamically based on information passed through file's extended attribute.*

and data flow passes through each plugin but the plugin processes the data only if it is enabled for the file. The key difference with stackable file systems is that the stack is dynamically configured on a per-file basis.

A key aspect of the ATTEST architecture is its flexibility and extendibility. The use of plugins allows a variety of storage policies to be defined at all levels of the storage stack. In addition, the parameterization of modules allows optional features to be enabled or ignored as desired.

4.2.3 Metadata Handling

Metadata is the most important component of a file system. For any file system to be accessible by the user, it is very important that a complete and consistent copy of metadata is available. Since ATTEST is capable of handling different levels of redundancy under a single file system, it becomes mandatory that a file's metadata is available at all times. ATTEST allows users to modify the redundancy level of the file during the lifetime of the file. In order to avoid moving metadata across different disk volumes, we designed ATTEST to always place the metadata (superblock, inodes, etc.) on the volume with the highest redundancy level.

4.2.4 Migration

ATTEST also allows users to modify their file's attributes. This permits the user to have control of their data not only during the creation of the file, but also throughout the existence of the file. However, this feature comes at the cost of delay due to migration. When the attributes are modified for a file, all blocks of the file are re-read from the disk and written to disk again, to pass through the stacks and device layer again, based on the new attributes.

In the case of directories, if the rules attribute, *user.attest.rules*, is updated, the new rules are applied on all new files created in the directory. In case a new stack is

added to the directory, the metadata in the directory passes through the stack, while the data within the directory is left unchanged. For example, if encryption is added to a directory's stack, its inode table and dentries' are encrypted, while the files that were already present in the directory are not encrypted. All new files created after this will however be encrypted. We chose to not apply the new directory rules to existing files because of the potential cost to migrate these files. If the user's intent is to apply rule changes to all files in a directory, it is easy enough to apply the rules through a `find . | xargs setattr XXX` type call.

4.3 Design

As discussed before, ATTEST provides the user with the capability to control their data at a finer granularity in terms of storage policies. ATTEST allows users to set rules for files in the file system through two methods. A rules attribute on a directory defines attributes for files created in that directory and the rules propagate to all sub-directories as well. The user also could manually override the default rules by setting attributes on each file or directory. Section 4.3.1 provides a detailed explanation on the format of rules and attributes and how to set them.

ATTEST is currently implemented using Linux 2.6.24 as a configurable option during compile. To give ATTEST portability across variety of file systems, the changes were restricted to the VFS layer and the device mapper layer of the kernel. Thus, ATTEST is not limited to any particular file system and can support all devices including SCSI and IDE as long as the Logical Volume Manager (LVM) can create physical volumes on them.

4.3.1 Attributes and Rules

Attributes are set on each file and each directory to convey storage policy decisions to the storage system. These attributes are set using the *setfattr* or *attr*

```
user.attest.redundancy='10'  
user.attest.compress='1'
```

Figure 4.2: *File's Extended Attributes*

commands. Figure 4.2 shows how the extended attributes are used to define policies – in this example, the user has requested compression and high redundancy. Since the devices present under LVM are hidden from the user for the most part, the user only provides an abstract storage policy without regard to implementation. For example, the redundancy rating on a file is given as a numerical value in the range of 0-10. This scale, from the user's view ranges between not important (lowest value) to medium availability (mid-value) to highest-availability and high performance (high value). The LVM layer based on its knowledge of underlying devices, will decide how to map a redundancy rating to the actual storage devices. The number however is not a definite indication of the underlying devices. The LVM uses this number only to get information on how important the block is, and which is the best available device for that block in the system. For example, a system need not always contain a RAID0, or mirrored (RAID1) devices to provide high availability. In the absence of RAID1, all blocks with redundancy value 10 could be placed on RAID5. This way, the user need not be aware of the complexities of underlying hardware.

Since it may be impractical to expect users to set attributes for each individual file, the user and system administrator can set rules that define the default attributes for a file at creation. An example set of rules is shown in Figure 4.3.

The rules follow the format of `< pattern >:< attributes >`. Rules allow users to set attributes for newly created files based on regular expression matching on the filename. Thus, in the example given, all `.c` files in `/home/user1/prj_code` will be given policy attributes that indicate highest redundancy, while all `.mp3` files on the system will have backup enabled. When the user creates a new file, it is checked in

```
Form: <pattern>:<attributes>

/home/user1/prj_code/*.c:redundancy=10
/tmp/*:redundancy=0
*.c:compress=1,redundancy=10
*.o:redundancy=5
*.mp3:backup=1
```

Figure 4.3: *Rules format*

VFS with all the rules set by the user for a match. The attributes of the file are decided based on rules with which a match was found. In case there is a conflict due to different rules matching on the same file, ATTEST uses the most conservative interpretation, i.e. the rule with highest redundancy is chosen. The attributes generated based on the rules can be overridden by manually setting the extended attribute directly. These manually set rules will always override the rules, thus giving users a simple way to have more control on their data. Although the rules are currently name based, with minor implementation changes one could also set rules based on other file metadata such as size, ownership, permissions, etc. Attributes specifies the storage policies for a file, and ATTEST uses this information to implement the policy either through stack plugins or through underlying storage functionality.

```
user.attest.redundancy='10'
user.attest.compress='1'
user.attest.rules='
*.c:compress=1,redundancy=10;
*.o:redundancy=5;
*.mp3:backup=1,redundancy=2;
*:redundancy=0'
```

Figure 4.4: *Directory's Extended Attributes*

Rules are defined in two ways: either through a system administrator defined set of rules or directory-specific rules. The system administrator can create a global rules

file that overrides all rules on a system. This file is read by the kernel when a ATTEST system is mounted. The rules file is read only once and is cached in the kernel to reduce repeated disk access. A reload of this rules file after modifications can be forced by simply setting a `/proc` parameter. Directories, along with regular policy attributes, also hold the rules which are applicable to all files created in it (Figure 4.4). Any new file in the directory will automatically inherit the rules from the parent directory, unless manually overridden. Similarly, sub-directories will propagate the rules, unless modified. In addition, a user could set his own rules apart from administrator's rules. These rules, as per rules and priority set by the administrator, could supersede the admin's rules or be ignored by the system. ATTEST gains from this multiple rules setup in that it provides the administrator control of files for which rules are not defined by the user. The user, on the other hand, is empowered with more control on how his data is saved on the system. Even if the administrator picks default conservative rules which give importance to reliability over performance for all files on the system, the user could apply relaxed rules on his files to get higher I/O bandwidth out of the system.

4.3.2 Device Tags

While we would ideally like to see the system identify the devices and redundancy level provided by them automatically, it is not possible without tagging. Since it is important for ATTEST to know the kind of devices available in LVM, we use tags to convey this information. Before ATTEST is set up, the physical volumes added to the LVM are tagged to represent their capabilities. This can be accomplished by using the existing tagging feature of LVM. All the volumes are then placed under a single volume group tagged 'ATTEST'. These tags are read on boot using a script and passed to the device mapper layer using `/proc`. This information can be read from ATTEST's folder created by the kernel module.

An example of device tags is shown in Figure 4.5.

```
# cat /proc/attest/devices
/dev/sdb  RAID5
/dev/sdc  RAID0
/dev/sda1 DEFAULT
```

Figure 4.5: *Device Tags*

4.3.3 Mapping Table

At the LVM layer, we map all requests from the file system to blocks on the devices according to the device type requested through tags. To do this, we have added a mapping table to the LVM. This mapping table maps the blocks visible to the file system to the real blocks on the lower disk drives. Since the file system is not aware of the underlying devices present in the system, it conveys information regarding the desired redundancy level to the LVM through tags. LVM, being aware of the connected devices, can simply map the blocks to the device that fits the required policy/functionality. For example, if File1 in Figure 4.6 requires the highest redundancy level, blocks from those files will be mapped to the highest redundancy volume available in the system; in this case, RAID5. Each shaded block in the figure refers to the allocated blocks in the file system, and each of them are mapped to lower volumes based on their tags.

Adding this new mapping table to the LVM layer brings some level of complexity to the system. Since this re-mapping information is not sent back to the file system, a permanent mapping is required at the LVM layer. Further, the LVM does not have any information about which block is being used by the file system and which is not. Hence, a persistent copy of mapping of both, in-use blocks and free-blocks needs to be maintained in LVM. This mapping table is read during file system mount and cached in the kernel. The table is also flushed to the disk at regular intervals.

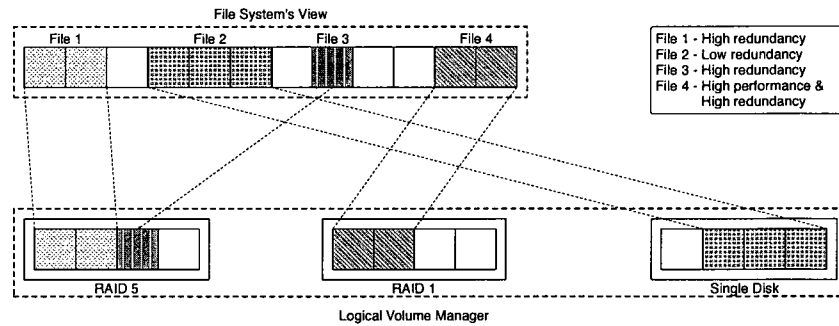


Figure 4.6: *File system's block to logical volume mapping*

```
struct att_extents {
    ...
    sector_t start_block;
    sector_t length;
    ...
};
```

Figure 4.7: *Extent-based mapping*

To minimize the size of this mapping table, we save the maps as extents. Extents are contiguous area of file system mapped to contiguous area on the device. Thus, instead of saving information regarding mapping of each and every block, we only save the starting block number and the length of buffer. Shaded blocks present together in the file system view as shown in Figure 4.7 is an example of extent. This reduces the number of elements in the red-black search tree, and thus reducing search time and memory overhead on the system. If the file system at any later point frees a block which is part of the extent, the extent is resized, or split, depending on where the free block is present in the extent.

4.3.4 Plugins

Different stackable file systems are loaded as plugins in the ATTEST architecture as explained in previous sections. These plugins are enabled or disabled depending

on the attributes set by user. We currently use FiST modules to provide plugin functionality. We have modified the source code generator in FiST (fistgen 0.2) to support our controllable stack/plugin architecture and include checking of attributes. If the attribute for a particular plugin was not set, the request is allowed to directly flow to the next level of the stack.

4.3.5 Migration

One of the major challenges in rule-based allocation policy is how to provide users the flexibility to modify attributes at any time during the life of the file. Modifying attributes such as backup, and snapshot may be easy to control or change; however, redundancy levels are not since a change to the redundancy level of a file necessitates a migration of the data to a different redundancy device. We can do this by moving the file from one redundancy volume to a volume with another redundancy. This practice can also be seen as a move of file across partitions with different redundancy levels. However, this move brings with it some complications. A file which requires high redundancy also demands that all its parent directories are protected at least at same level. To solve this problem, we place all metadata of the file system on a volume with the highest redundancy from the beginning. This way, we will never have to worry about moving or protecting the path in which an important file exists.

4.4 Results

4.4.1 Experimental Setup

We conducted our experiments on dual-core AMD Opteron 1.8 MHz dual-processor machines with 2 GB RAM, five 80 GB, 7200 RPM SATA disk drives and an Areca RAID controller. Four drives were set up as RAID-5, providing a 240 GB volume capacity while one drive was used as regular volume. RAID-5 was set up with a 64 KB stripe size. We used the EXT2 file system throughout our experiments as the

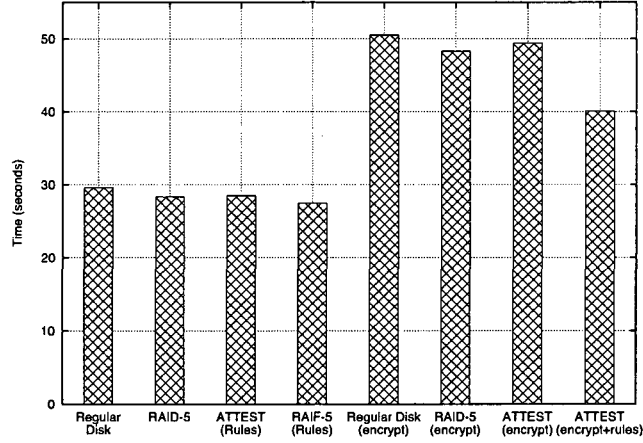


Figure 4.8: *Time to compile am-utils (shorter bars are better)*

underlying file system. The following experiments evaluate the impact of ATTEST on the performance of the system. All results are the average of five runs.

In all experiments, we compared the performance of ATTEST with a base single disk EXT2 file system and with a four disk RAID5 EXT2 file system. We also compare ATTEST with RAIF [79, 78] which also uses rules to perform file level volume allocation. RAIF differs from ATTEST in that rules are used to determine placement of files on different file systems that are unioned to appear as a single file system. The RAIF rules do not allow for low-level storage optimizations. For our experiments with RAIF, we modified its release version 1.1 to run on Linux’s kernel 2.6.24.

4.4.2 *am-utils*

A potential use-case of ATTEST is in software build environments where `.c` files require high redundancy and binary files do not. As a measure of this environment, we tested ATTEST by compiling Berkeley’s Automounter utilities [3]. The *am-utils* package contains over 500 files with roughly 50,000 lines of C code in several dozen small files. Upon compilation, eight binaries are built.

We measured performance of the system in eight different scenarios. First, by

compiling *am-utils* on a logical volume comprising a single disk and then again on a logical volume comprising a 4-disk hardware RAID5 system. Subsequently, we ran it under ATTEST with rules where all the source code files were given high redundancy and placed on the RAID-5 volume and binary files (generated as a result of compilation) were placed with low redundancy on the single disk volume. RAIF was set up with 5 file systems. Four of them were combined to simulate RAIF-5 (RAID level 5) and the fifth disk was used as a single volume to place binary files. From Figure 4.8, it is clear that ATTEST does not have any significant performance penalty. Our experiments also show that RAIF has a slightly better performance, mainly because RAIF can benefit from caching available in the virtual file system layer.

We also repeated the experiments with *cryptfs* - an open source, computation heavy, encryption-based stackable file system available with the FiST package[138, 140]. On ATTEST, we ran the experiment with two different conditions, one where all files were encrypted while passing through the stack, and the other where only *.c* files were encrypted. The *.c* files comprise about 22% of the files in the *am-utils* build. We were not able to use RAIF with *cryptfs*. From Figure 4.8 it can be seen that using rules with ATTEST provides a performance gain when compared to both regular disk and RAID with encryption. This clearly demonstrates the benefit of having file-based storage policy rather than using the same policy for the entire volume.

4.4.3 IOZone

We also tested a similar set up using the file system benchmarking tool IOZone[14], which performs synthetic read/write tests to determine the throughput of the system. We ran IOZone experiments on a freshly formatted file system with default set up and without rules or placement policies. We ran the experiments with a file size of 2 GB and a record size of 64 KB. The graphs in Figures 4.9 and 4.10 show the

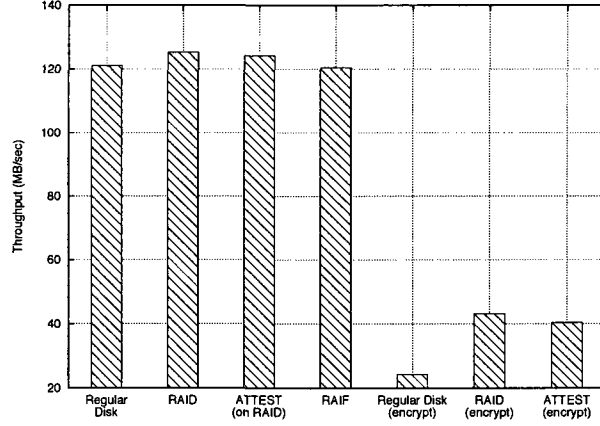


Figure 4.9: *IOZone - Read (longer bars are better)*

throughput of the system in megabytes per second. It was again seen that ATTEST's performance was on par with RAID controller for reads. For writes, we observed a slight hit on performance primarily because of the mapping table at the LVM layer. Since ATTEST remaps the file allocation at the LVM layer, optimizations that the file system makes to keep file metadata and data near each other is lost. As a result, inode metadata ends up at the beginning of the disk and data is interspersed throughout the disk, and hence, every write requires head movement to the beginning of the disk before writing data, thus resulting in slightly poorer performance. The data retains spatial and temporal locality and as a result does not suffer in reads. We are investigating methods to allow the file system to make the LVM and storage layers aware of the optimizations that it is trying to perform when doing block allocation.

4.4.4 File creation

A potential issue with the ATTEST architecture is its effect on metadata performance. We are primarily concerned with file creation as rules must be evaluated at multiple levels of the stack. In order to measure the stress of the creation of large numbers of files within a directory, similar to a large shared file system, we timed a

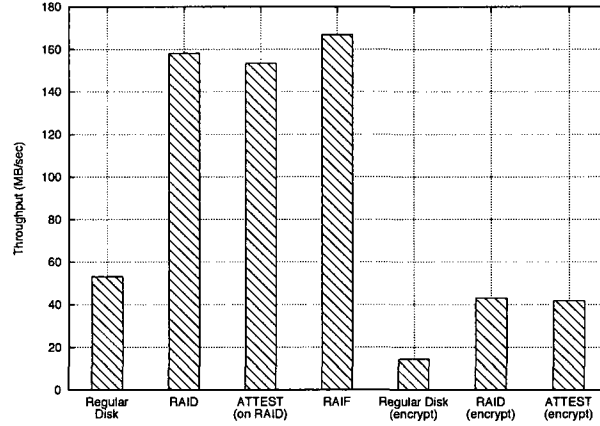


Figure 4.10: *IOZone - Write (longer bars are better)*

simple application which created 10,000 files in a single directory. The rules were set such that half of the files go to a RAID5 volume and the other half go to a single disk volume. Table 4.1 shows that ATTEST's performance was almost the same as that of regular disk and RAID controllers while RAIF was very expensive. The reason that RAIF suffers is that it has to deal with metadata management on multiple file systems, on RAIF and on the underlying file systems. In contrast, ATTEST deals with only a single file system metadata management.

Further, to test if ATTEST's performance varied with an increase in the number of rules, we ran the same create experiment with over 650 rules. However, we did not see any change in performance from those reported in Table 4.1.

4.5 Summary

In this chapter, we presented our motivation for extending storage system policy administration to the file level thereby providing opportunities for efficiencies in terms of storage overhead and performance at several levels of the storage hierarchy. User-directed storage policy through the use of extended attributes can assist storage administrators in determining optimal storage usage and allocation. We have also

Table 4.1: *Time to create 10,000 files*

Setup	Time (in seconds)
Regular Disk	7.259
RAID	6.902
ATTEST	6.914
RAIF	28.718

outlined ATTEST, an extendable storage technology that can enable this file based storage policy model. As storage administration costs go up, we believe the one size fits all policy model no longer works and extendable policy models such as what we have presented are required. These file based storage policy rules can be set by applications, administrators or end users allowing flexibility in decision making.

Chapter 5

User Space Storage System Stack Modules with File Level Control

The file system is often seen as one of the most critical parts of an operating system. It handles the task of storing and organizing user files and their data on the underlying storage devices. It is comprised of very complex C kernel code which takes several months to develop and stabilize and is usually written for a particular operating system platform. The file system code must interact with the operating system's virtual file system manager to receive system calls from the user-space, with virtual memory manager for page allocation and memory management within the kernel and with the virtual device layer to communicate with the storage devices and store data. This makes the file system code remarkably complex to understand and very hard to develop. An average modern file system is comprised of around 50,000–60,000 lines of code and supports a variety of features, such as B-tree based search, flexible data extents, access control lists, extended attributes, etc. [142]. This low-level kernel code is very difficult to program and is often the origin of bugs in a storage system [66, 104]. To add any new feature in a file system, a programmer needs to have a thorough understanding and working of the file system. Apart from programming, providing support and maintenance for such large and complex file systems with several features and diverse mount options is also very hard. Thus,

file system development and maintenance is always considered to be the work of the select few who have a very deep knowledge of the file system and also the operating system.

Several techniques have been suggested to simplify the process of file system development. To address the need to quickly develop and incorporate new features in an existing file system, the Linux kernel has provisions for implementing stackable file systems. Stackable file systems [111, 69] give developers a quicker way to add new features to a file system through an extensible file system interface. It reduces the complexity of developing a newer file system, in that it allows features to be added incrementally in steps instead of creating a new file system from scratch, or modifying an existing one. However, to obtain the best performance, these file systems are tightly integrated into the Linux kernel or are designed and developed to run as a kernel module, thus requiring the uphill task of understanding the kernel before starting to develop a file system.

File System Translator (FiST) [140] is a file system generation tool that simplifies the task of creating stackable file systems by generating most of the code from a standard file system template. The programmer is required to provide code only for the main functionality of the file system, which is then fed to the FiST file system generation tool. The resulting code can then be inserted into a live system as a loadable kernel module. However, this simplified coding requires learning a new file system template language.

Developing kernel-space file systems is very difficult and suffers from several drawbacks. They cannot be ported across different platforms and they also do not provide any options for non-privileged users to mount a file system. File system in User Space (FUSE) is another solution to simplify writing a file system and can be ported across different operating system platforms. It has been integrated into the Linux kernel tree and has ports available for other major operating systems. FUSE exports all file

system calls within the kernel to the user-space through a simple application programming interface (API) by connecting to a daemon that is running in the user-space. FUSE provides an easy way to write virtual file systems, in that the file systems do not store any data themselves. Writing a file system in user-space is much easier than compared to writing a kernel-space file system. FUSE also has provisions to permit non-privileged users to mount FUSE-based file systems. These user-space file systems however come with a small overhead due to context switches and memory copies made during the data transfer operations [138, 139].

UserFS [61] was an idea proposed in 1993 which exported file system requests to the user-space through a file descriptor. *puffs* is an export of FUSE-like library on the NetBSD operating system [22]. However, these solutions lack in their allow multiple file systems to be stacked on top of one another in the user-space. Our work differs from all previous work done in that we provide stackability for file systems that are present in the user-space. This approach has a major advantages, in that it can be mounted by any user on the system and does not require learning any new language.

As discussed above, FiST and FUSE are two very common solutions to simplify the process of writing file systems. However, with these existing systems, performance, portability and availability to non-privileged users, cannot be achieved together. In this work we propose a stackable FUSE architecture that will allow the development of stackable file systems in user-space. In order to limit the overhead due to context switching between the kernel space and user-space, we propose to combine this stackable FUSE design with ATTEST, attribute-based storage framework discussed in Chapter 4.

The remainder of this chapter is organized as follows. Section 5.1 discusses the stackable file system model, while Section 5.2 describes the FUSE architecture. In Section 5.3, we provide details of our design for stackable FUSE module, Section 5.4 gives a brief description of our implementation approach and Section 5.5 shows the

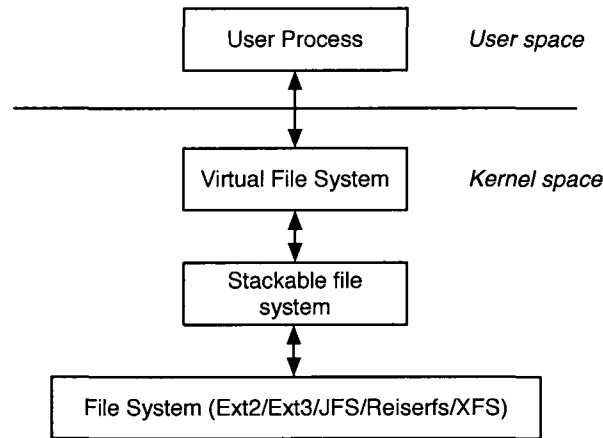


Figure 5.1: *Stackable file systems.*

performance results. We end the chapter with a section on summary of this work.

5.1 Stackable File Systems

The idea of stackable or layered file systems was adapted from the *vnode* interface first implemented on SunOS in 1984 [82]. Stackable file systems [110, 111, 137, 118] are stand-alone file systems that can be mounted on top of an existing file system mount point. Figure 5.1 shows the typical arrangement of a stackable file system present between the Virtual File System (VFS) and a lower-level file system, which may or may not be a device-based file system. The advantage of developing a stackable file systems is that they can be used to extend the functionality of an existing file system without changing the code of the original file system. A stackable file system creates a *vnode* with its own operations that is inserted on top of the *vnode* belonging to the underlying file system. This allows a stackable file system to perform operations in between the VFS and the lower file system calls. For example, an encryption process can take place before the data is written on the lower file system, or, a decryption function can be run after the data is read from the lower file system. Stackable file systems can be used to add many functionalities such as compression, encryption,

caching, etc. to an existing file system. Other examples of stackable file systems include WrapFS [139], UnionFS [105], RAIF [78], AVFS [94], etc.

fistgen is a tool that allows a developer to create a stackable file system by only describing the core functionalities of the file system using the File System Translator (FiST) language [140]. The file system generator tool, generates the code for a file system that can be directly loaded as a kernel module into a live Linux system. To add some of the functionalities in FiST, however, requires learning a new language. Since loading a kernel module in a system is restricted to privileged users only, file systems generated using FiST can only be used if inserted into the system previously by an administrator or a privileged user.

5.2 File system in User space (FUSE)

Filesystem in Userspace (FUSE) is a combination of a user-space library and a kernel module for Unix-like operating systems that allows non-privileged users to create their own file systems without editing the kernel code [9]. This is achieved by running the file system code in user-space, while the FUSE module only provides a bridge to the actual kernel interfaces through a set of APIs. FUSE's kernel module simply redirects the Virtual File System (VFS) calls to the user-space daemon. Figure 5.2 shows the internal architecture of FUSE. Several FUSE-based file systems are already in common use. FUSE can be particularly useful in providing a POSIX interface for files which are accessible over the network through different network protocols. Some of the file systems based on such a design are *sshfs* [25], *httpfs* [12], *CurlFtpFs* [5], etc. FUSE file systems are easier to maintain since they run in user-space. They are also easier to code and debug compared to the kernel file systems. Running file system in user-space also implies access to more libraries. Thus, FUSE file systems can be written in any language that has a binding to the FUSE libraries, including Ruby and Python.

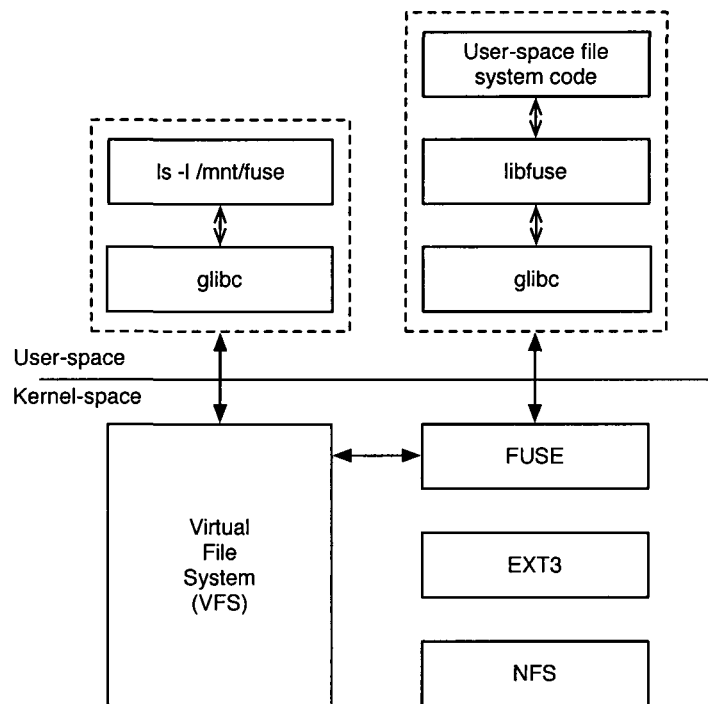


Figure 5.2: *FUSE architecture showing kernel module and user-space library.*

However, file systems created using FUSE are always the lowest file system in the storage stack. This means that all FUSE requests must return from the FUSE layer to the user-space applications without going to the lower-level file system within the kernel, similar to a stackable file system. Thus, FUSE, in its current form is not a solution for developing a stackable user-space file system.

5.3 Stackable FUSE

As already mentioned, FiST and FUSE are techniques to design new file systems with a lower learning curve as compared to writing a standard kernel-level file system from scratch. FiST helps by extending the capabilities of an existing file system while FUSE allows easy programming and maintenance of a file system in the user-space. FUSE also provides the ability for non-privileged users to mount file systems and use it.

In this work, we propose a new stackable file system module called *sfuse* that will provide users with a FUSE-like interface in user-space to write their own file system in the user-space. The added advantage of *sfuse* and the difference compared to FUSE is that it will provide stackability similar to that available using FiST. Thus, data in all I/O operations will be sent to the user-space, copied or modified, and returned to kernel-space to be pushed to the lower-level file system. Since a FUSE-based file system has a cost due to context switching and memory copies, we also extend the idea of ATTEST and limit the overhead only to files that require the user-space functionality implemented in the file system.

There are several advantages of porting file system stackability to the user-space. One of the main advantages is that such a scheme would allow any user on the machine to mount a stackable file system without the need of administrator privileges. Along with stackability, per-file control on the files will allow the user more control on how the files are treated. *sfuse* also avoids the need for the user to understand the FiST

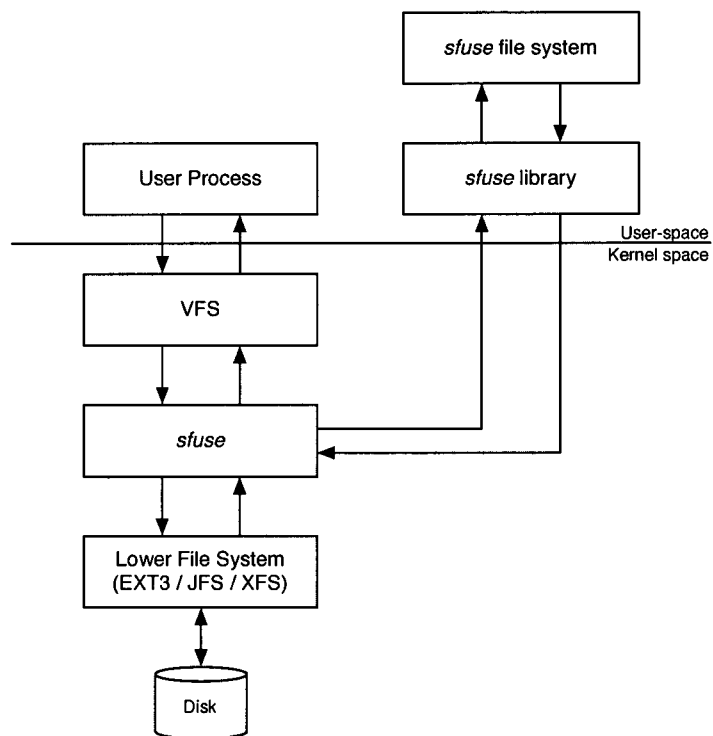


Figure 5.3: Operation flow in *sfuse* with a user-space *sfuse* file system.

language in developing a stackable file system. Figure 5.3 shows the operation flow with a *sfuse*-based file system present in the user-space. In case of READ, the data would flow away from the lower file system, while during WRITE operation, the data would flow towards the lower-level file system. Section 5.4 provides more detail on the internals of the *sfuse* file system module.

5.4 Implementation

We implemented our stackable FUSE-like file system *sfuse* on Linux kernel version 2.6.24. We started by first creating a stackable base file system *basefs* using FiST. We used a patched source of FiST version 0.2.1 to create the stackable file systems which was compatible with the Linux kernel version present on our machine. *sfuse* is designed to export all file system operations to the user-space daemon, similar to the default FUSE module. To export I/O functions to the user-space daemon, we modified the user-space FUSE library to receive requests from the kernel even without any previous file OPEN operation. All I/O requests are forwarded to the user-space file system, irrespective of whether an OPEN operation was performed on that file. This is in contrast to FUSE, where an I/O operation can be performed only after an OPEN call is made. This step is required in FUSE to open the actual file on the ported file system and obtain a file handle in the user-space. The file handle information is later used in identifying the file on which I/O needs to be performed. In *sfuse*, the file is actually opened within the *sfuse*'s Linux kernel module and multiple operations on the same file are handled within the kernel module.

A user may, however, also opt to also use the exported OPEN function in the user space, depending on the requirements of the stacked file system. As an example, for a simple encryption file system, encode and decode functions can be implemented without having a file handle to hold state. In another case, if the stackable file system is designed to count number of times a file is opened, an OPEN function will have

to be implemented in the user-space in order to create a file handle to hold the file count.

To implement stackability in FUSE, we also require the user-space FUSE library to return the request data buffer back to the kernel after performing the stack function. This is done in the same way as any write operation would be performed in FUSE module. One can force the user-space function to be called, both, before and after calling the lower-level file system operations. For example, a user can implement a file system where during a READ operation, the user controls if the READ should succeed by allowing the READ operation to fail or succeed from the user-space before it goes to the lower-file system. A failure from the user-space will return the data flow immediately, thus resulting in a failed read.

File systems developed based on *sfuse* are mounted in the same way as a FUSE-based file system. The mount binary file requires two parameters – the mount point directory and the directory which needs to have the stacked functionality on top of it. In the kernel, data structures for storing the file system’s private information must include information regarding the lower-level file system along with the connection pointers of the user-space FUSE daemon.

The user can control the files which must be exported to the user-space by using ATTEST. The user can lay rules or set policies for each file by directly setting the file’s extended attributes, or by including the rules in the ATTEST config file. More details on how to set the rules and policy in an ATTEST framework is explained in [98]. Stackable FUSE also allows attributes set by the ATTEST framework to be passed from the kernel-space to the user-space as tags along with any I/O request. This will allow the user to perform dynamic ordering of multiple stacked functions in the user-space without going back into the kernel-space.

Our current implementation only supports synchronous operations. This means that all operations can return to the kernel space only after the user-space functions

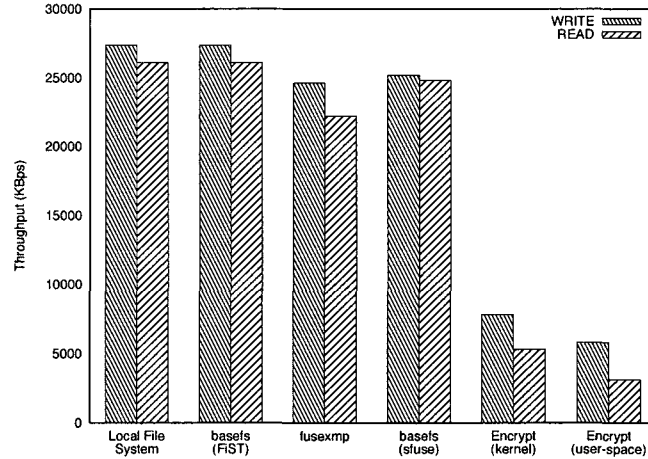


Figure 5.4: *Throughput comparison for different setup of file system using IOZone benchmarking tool on a 2 GB file and record size of 128 KB.*

have returned. As part of our future work, we plan to support asynchronous operations in *sfuse* library, which will allow the requests to be appended to a queue in the user-space file system. This queue will be cleaned by a thread running continuously on the system. One place where such a mechanism can be very useful is in performing lazy data backup and deduplication on a per-file or a per-directory basis. With the assistance of ATTEST rules, the user can also define policies, such as, if the files need to be backed up after compression, or encryption, or neither.

5.5 Results

We evaluated the performance of *sfuse* by running IOZone [14], a popular benchmarking tool that performs synthetic read/write tests to determine the throughput of the system over a variety of file system configurations. We conducted our experiments on a 1.8 GHz dual-core dual processor AMD Opteron machine with 2 GB RAM and two 40 GB hard disk drives running Linux kernel 2.6.24. The experiment was run for a file size of 2 GB with record size set to 128 KB. Figure 5.4 shows the results with overhead of using *sfuse* compared to other file system configurations.

We first ran IOZone on a default EXT3 formatted non-root partition to obtain

the base performance of the system without any stacks. Next, we ran IOZone on *basefs*, a stackable file system generated by FiST. *basefs* is an empty file system in that it simply forwards all calls to the lower-level file system. By running IOZone on *basefs*, we evaluate the overhead of including a stackable file system between VFS and disk-based file system such as EXT3. *basefs* is available along with the source code of *fstgen*. From Figure 5.4, we can see that including an empty stackable file system has a very negligible overhead.

Our third experiment was run on a *fusexmp* file system mount. *fusexmp* is a FUSE-based file system that simply mounts the root directory of the system on the given directory. It is available freely along with the FUSE source code [9]. Our experiments confirm the overhead that is expected to be present on any FUSE-based file system due to memory copies between user-space and the kernel-space. We observed almost 10% overhead for writes and slightly less than 15% for reads in this test. Our fourth setup was an empty file system similar to *basefs*, but set up in *sfuse*. This file system simply returned the request back to the kernel space without doing anything to the transferred data. We conducted this experiment to observe the real overhead of context switching and memory copy operations. In our experiments, we observed an overhead of around 8% for writes and around 5% for reads when compared to the local file system.

Our fifth experiment was done on a slightly modified version of *cryptfs* stackable file system available with the *fstgen* source code. We disabled the encryption of file names in *cryptfs* and only allowed data block encryption. We implemented the same encryption algorithm in user-space and implemented a *sfuse* file system for it. We observed almost 26% overhead for writes and around 39% overhead for reads by porting the code to the user-space. The overhead in this case is primarily because of context switching between the user-space and kernel space. The data buffer memory is copied two times in each I/O operation for each direction of the data flow, i.e., once

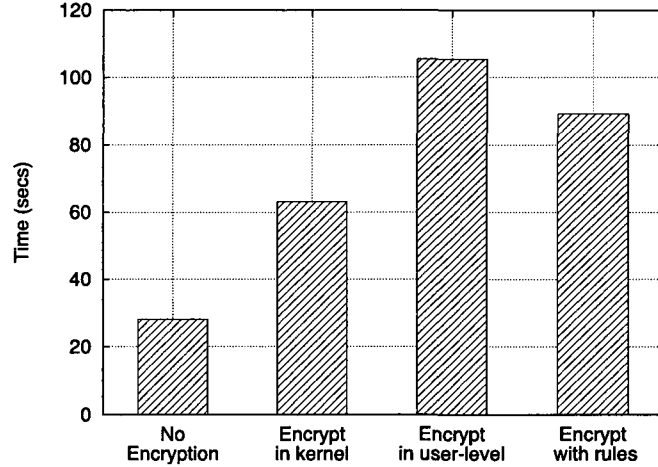


Figure 5.5: *Time to compile amutils on ATTEST with rules to encrypt only .c files.*

from the kernel to the user-space, and then, from the user-space back to the kernel space. This overhead also comes from several other aspects within the operating system like processor registers that need to be saved and restored, cache entries that need to be evicted and reloaded for the incoming processes, etc. [88, 106].

We also tested *sfuse* when integrated with the ATTEST framework. In Figure 5.5, we compiled *am-utils* on a local disk without any encryption and compared it with encryption done by using the *cryptfs*. We compared the time to compile *am-utils* with encryption in user-space and also under ATTEST framework with rules to enable encryption only on .c files. From the figure, we can see the overhead due to *sfuse* when compared to kernel-based stackable encryption, however, this can be reduced by applying ATTEST-based rules.

While the overhead in our experiments are certainly non-negligible and casts doubt over the need to port file systems into the user-space, we remind the reader the benefits such as the ability for non-administrator accounts to control their data, simpler programming and debugging in user-space with FUSE bindings available in many programming languages other than C and per-file granularity control sufficient to make this a useful solution. Further, our *sfuse* code has not been highly optimized

and could be improved significantly to lower this overhead.

5.6 Summary

In this chapter, we have presented a stackable user-based file system model which can be controlled based on user defined rules. Most of the existing file systems make compromise on adding costly functionalities because there is no way to make policy decisions at a finer granularity. In this work, we reinforce our commitment to providing users more control over policy decisions on the files by using ATTEST. Stackable file systems in user-space opens up a variety of opportunities to design file systems. However, the stackability of these file systems are expensive due to context switching. By using the ATTEST framework, we can select files and enable stackability and absorb the overheads only for files that require the stackable functions enabled.

Chapter 6

Extendable Storage Framework for Clustered Storage Systems

In earlier chapters, we have described the ATtribute-based Extendable STorage (ATTEST) architecture which allows users to set a variety of rules at the file-level granularity [98]. In this chapter, we extend the existing ATTEST framework to clustered file systems by implementing it into the Parallel Virtual File System (PVFS) [49, 21]. Clustered file systems not only allow the data to be stored over the network, but also allow it to be shared with others. In large organization, this forms the typical setup for sharing data. Such a framework on distributed file system can be used to provide a comprehensive storage system with a variety of options that all applications and all users can take advantage of.

For our work, we implemented ATTEST within PVFS to forward a file's attributes to the storage servers, which further push it down to the local file system that already supports ATTEST. Message passing interface (MPI) already has provisions to forward I/O hints provided by the application to the file system, but only some of them can really be useful for PVFS. PVFS stores all the file's data on storage node's local file system. Still, practically, none of those hints is forwarded further down the stack on the storage node, that can be of any use to the local file system. This is because there is no provision in the operating system, or the local file system for making any

use of it. In our work, we use ATTEST and PVFS together to provide applications a way to forward a file's attributes to the storage nodes where the data is stored, to be used by the node's local file system with several plugins and variety of devices. We do this by making use of hints in PVFS which are typically used for carrying MPI hints to the storage nodes.

6.1 Implementation

In PVFS, a directory's metadata, including the extended attributes, is stored on the metadata server. Therefore, for every open system call on a file, a request is sent to the metadata server to obtain the file's attributes. The metadata reply, however, only includes the file's attributes along with information about the storage nodes on which the data is placed. To support ATTEST, we require that the metadata server also return the client's extended attributes with the parent directory's extended attributes. While on a single machine running ATTEST, this can be easily done by making another system call to read the extended attributes, on networked file systems, it is not as straight forward. This is because the request will be sent across the network to the metadata server and the client will have to spend extra time waiting for the metadata server to respond. This process can be very expensive and can bring significant overhead to the system. PVFS, however, has integrated support for file system related hints that can be included in every network communication that takes place between the nodes. For our work, we modified PVFS version 2.8.1 to use these hints to attach requests to receive the extended attributes from the metadata server, and send any new or updated attributes to the metadata server.

Once we have the file's extended attributes, we transfer the attributes to the storage nodes, to be saved as extended attributes of the data files on the each storage nodes. We again use I/O hints in PVFS as a carrier for the extended attributes. On storage nodes, PVFS caches file descriptors of all open data files until a high threshold

```

user.attest.cache='1'
user.pvfs.encrypt='1'
user.pvfs.redundancy='10'

```

Figure 6.1: *File's Extended Attributes on Clustered ATTEST.*

is reached, or the node runs out of memory. If the I/O request is on a file which is being opened for the first time, we set the attributes on the file during the create system call. We also cache the attributes along with the descriptors. If the attributes is changed in any subsequent requests, we simply update the attributes on the file system and also in the cache. We support this to handle the case where the file is created without any data in it (eg. *touch*), and the user wishes to modify its attributes after creating. We however currently do not allow any online migration of data from one device to another. This behavior is similar to PVFS's own data distribution limitations, in that it does not allow a user to modify the striping mechanism once the files have been created. If the user still requires a change in the properties, for example, a change in redundancy level or need to include a new plugin, the data will have to be copied into a new file with the new attributes.

Attributes are set on each file, or a directory in the same way as explained in Section 4.3.1. The only difference in defining rules is the way to distinguish the location where the rules are to be checked. As shown in Figure 6.1, `user.attest` rules are checked on the client side while `user.pvfs.attest` rules are checked on the storage nodes.

Figure 6.2 shows the block diagram of our framework with ATTEST included on both, the client and the storage targets. Applications running on PVFS can take advantage of various plugins available through the POSIX interface on both the client, as well as on the storage node. The applications can also opt for the kind of storage device that they prefer the data to be stored on the target node.

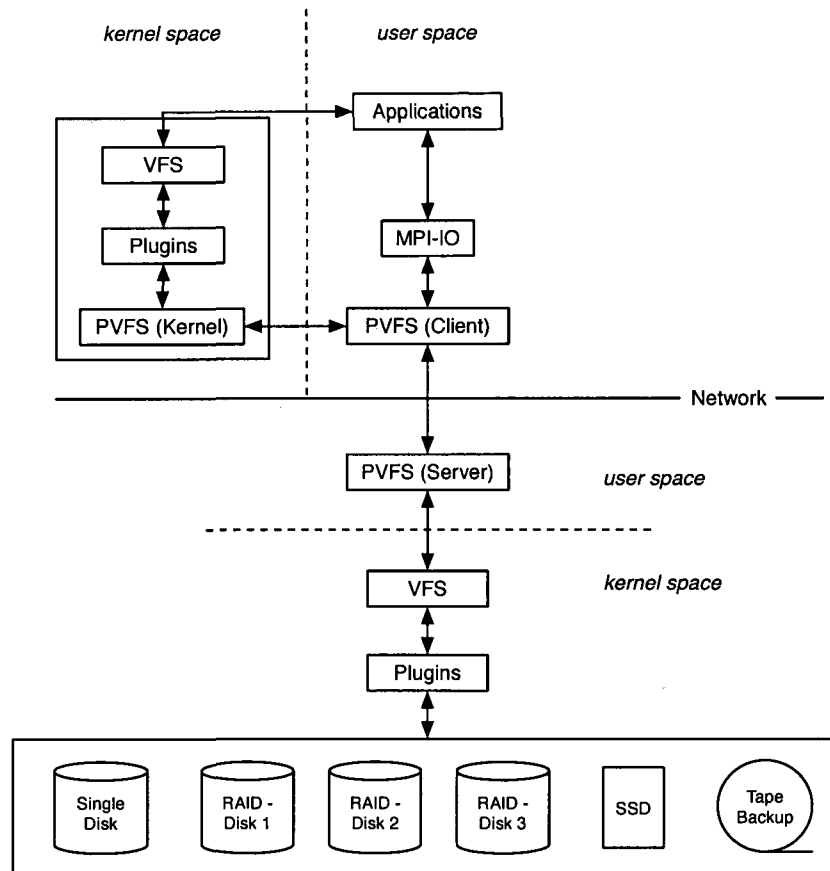


Figure 6.2: *ATTEST framework incorporated within PVFS*

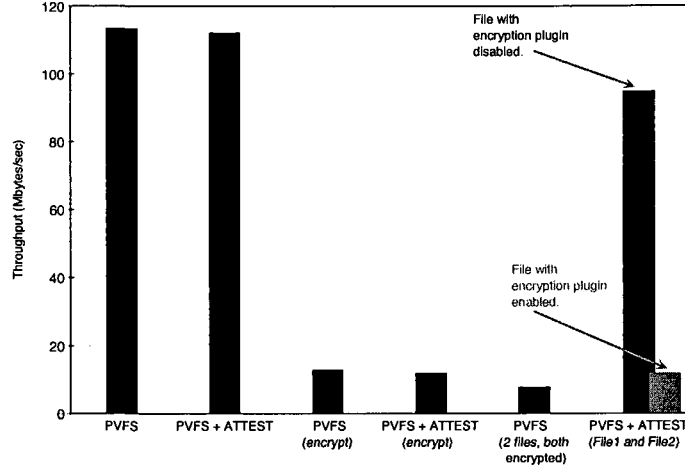


Figure 6.3: Write throughput using on IOZone on PVFS with ATTEST setup compared with the default PVFS setup

While application-aware storage could provide for some of this work, we distinguish ourselves by providing file-based policies with plugins and access control to variety of storage devices. The ATTEST framework on clustered file systems will provide the applications a way to assign particular device based on the rules set by the user storing data on the storage system. The storage stack on its part will try its best to match the requirements based on the available storage devices connected to the system.

6.2 Results

We evaluated our ATTEST implementation in PVFS by configuring the parallel file system with three metadata server and three storage nodes. Each node is a dual-core AMD Opteron 1.8MHz dual-processor machines with 2 GB RAM, five 80 GB, 7200rpm SATA disk drives and an Areca RAID controller. Three out of the five drives were set up as RAID-5, providing a 160 GB volume capacity while the other two drives were used as a regular disk volumes, also forming the system's root partition. RAID-5 was set up with a 64 KB stripe size. We used the EXT3 file system throughout our

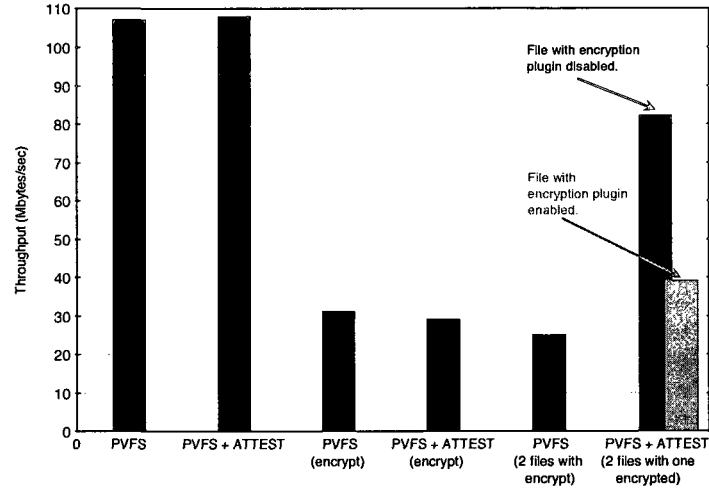


Figure 6.4: Read throughput using IOR benchmarking tool on PVFS with ATTEST setup compared with the default PVFS setup.

experiments as the underlying file system on storage nodes. All results presented here are the average of five runs.

Figure 6.3 shows the results for PVFS write throughput with ATTEST compared to the default PVFS setup under IOZone. We ran the benchmarking tool for a 4 GB file size with record size set to 1 MB. We can see from the figure that ATTEST incurs a very minor overhead when compared to the default PVFS. To test plugins, we also ran the experiment with encryption enabled, for both, PVFS and PVFS with ATTEST. We again see no major impact of including ATTEST. However, when we ran the same test with two files, we could clearly see the advantage of using ATTEST. We ran IOZone simultaneously on two nodes inside a parallel shell to observe their throughput. The last bar in Figure 6.3 shows the throughput for files, when only one file was set using attributes to go through the encrypt plugin. We can see the performance for both files improved compared to having both pass through the encrypt plugin. Read throughput performance for the IOZone provided similar results.

We also evaluated our framework using IOR, another file system benchmark de-

veloped at Lawrence Livermore National Lab (LLNL) used to measure the aggregate I/O throughput by focusing on parallel/sequential read/write operations. Figure 6.4 shows the read throughput for PVFS with and without ATTEST along with encrypt plugins. From the figure, it is clear that the behavior is similar to what we have seen previously. The overhead due to ATTEST is very minimal and we can clearly benefit from only selecting files that require any cpu-intensive plugins and leaving the other files for a much higher throughput.

6.3 Summary

In this chapter, we have presented a framework for extending storage system policy administration to the file level on a clustered file system. The system provides opportunities for efficiencies in terms of storage overhead and performance at several levels of the storage hierarchy. User-directed storage policy through the use of extended attributes can assist storage administrators in determining optimal storage usage and allocation. The framework is an extension of the ATTEST extendable storage technology that enables this file based storage policy model. As storage administration costs go up, we believe the one size fits all policy model no longer works and extendable policy models such as what we have presented are required. These file based storage policy rules can be set by applications, administrators or end users allowing flexibility in decision making.

Chapter 7

Improving Reliability in Clustered Storage Systems using ATTEST

For increased reliability, a key requirement of parallel file systems must be the ability to deal with failures. These failures may be in the form of either data corruption on a disk on a storage node, or complete loss of the node itself. Disk drives are complex electro-mechanical systems with several moving parts, making them unreliable in terms of availability and integrity of the data. They could suffer from potential data loss or complete drive failure. Similarly, a node failure can be because of any reason at any level from network interface, to processor, to disk. With multiple nodes in the cluster, and multiple disk drives present on each of them, the likelihood of failures on a single node increases. This makes designing a storage system a complex problem. The system must not only be efficient but also fault tolerant. Further, in cases of node failures, the algorithm must not only be able to handle reads, but also address write requests. A simple and frequently used approach to provide data redundancy in single node systems is to stripe the data across several disks using RAID [102] techniques such as parity protection or mirroring. However, applying RAID techniques to a clustered storage system introduces several new challenges particularly with regards to performance. Further, they can at best only be used for handling node failures. RAID techniques, by themselves, do not guarantee the integrity of the data. The

reason is that RAID does not have a notion of what is “correct” data from the user’s point of view. In terms of availability, the most common solution implemented in several clustered storage systems is the use of failover systems which allow the functionality to transparently move from a failed node to a backup node, both connected to the same storage array [46]. On the other hand, to handle data corruption and test the integrity of the data, hash functions such as cyclic redundancy checks (CRCs) or checksums are frequently used. But all these fault tolerant mechanisms are very expensive, both in terms of performance and cost. As a result, the reality is that most of the time, in large clusters, the systems are maintained without any protection.

In this chapter, we present algorithms for distributing data reliably and maintaining scalability in a clustered file system. We describe new techniques used to ensure a functioning system in the face of failures. We implemented our algorithms on two popular parallel file systems – Lustre [46] and Parallel Virtual File System (PVFS) [49] and show how with minimal overhead we were able to achieve both data redundancy and data integrity in our system. This chapter is organized as follows. Section 7.1 explains our failure model. Section 7.2 describes our algorithm to preserve consistency and data availability and Section 7.3 describes our algorithm to protect against data protection. Our performance results are presented in Section 7.4.

7.1 Failure Modes in Clustered Storage

Parallel file systems gain high throughput by striping the large chunks of data known as objects across multiple storage nodes as shown in Figure 7.1. During any I/O operation, the client simply reads these blocks in parallel from all the nodes. However, with this striping of data comes the challenge of keeping the data available. A failure of any node could compromise the entire file system. In this chapter, we work on two popular open-source parallel file system – PVFS and Lustre to demonstrate our reliability algorithms.

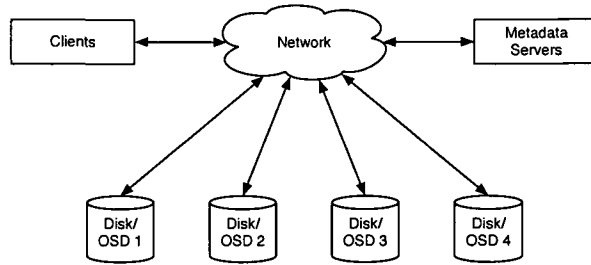


Figure 7.1: *Parallel file system architecture.*

Providing fault tolerance in a parallel file systems is a complex problem. With data spread across several nodes, special techniques are required to tackle coherency and consistency issues of the file system. PVFS and Lustre, both behave differently in a node failure scenario. Lustre keeps the system available for all I/O operation under any node failure. Instead of throwing admin alarms during failure, it complains only when a failed node is encountered during the I/O operation by returning a failure message. It spreads all writes during the failure on the remaining available nodes. Even if the failed node returns, it is not used for I/O on files that were created during its absence. This means that for those files, the system is being under-utilized because of a short failure of access to a node. While this could be acceptable in cases where not many failures occur, in large clusters where there are over 1000 nodes and where chances of failures are much higher, it is important to have the system utilization to the best.

PVFS, on the other hand, has no failure handling capabilities. In case of a failure, no I/O operation can be performed on the file system. The file system is unavailable for all operations, including the creation of new files, until the failed node returns. This is because PVFS by default stripes the file system's root directory information across all storage nodes.

7.1.1 Single-node Failure Model

Before discussing our algorithms for maintaining data availability, it is useful to explain the failure model assumed. We expect that the file system should suffer no loss of data under any single failure condition and provide continuous response to client nodes. The potential failures in the system that we consider are transient failures to a single machine which could be because of a network failure or a temporary hardware failure. We define these faults as *clean errors*. Our algorithms are completely clean-error tolerant. We ideally assume that only one machine will fail during any defined period. Because the failures may be intermittent, the protocols must handle machines temporarily disappearing from the network which may be caused due to disconnection or failure of a network cable. A disk failure can mean a variety of errors in the low level disk subsystem.

As for unclean errors, we mention here the possibility of incorrect data. Examples include network cards flipping bits inadvertently, memory getting corrupted, processors performing erroneous calculations or silent data corruption occurring on disks.

7.1.2 Data Corruption

Data corruption refers to unintended modification of data during its transmission, storage, or retrieval. These changes go undetected or unreported to the storage system. They result in incorrect data being flushed to the drive or sent to the application, depending on the flow of data operation. These kind of errors could occur for several reasons. Misdirected writes, torn writes, data path corruption, and bit-rot are some of the ways in which data corruption can occur, resulting in incorrect data being stored on the block. Even if the data was stored correctly, a bug in the firmware code could cause incorrect data to be read from the disk resulting in an error.

Several methods are available for detecting data modification or silent corruption. Data integrity guarantees the users that once their data is stored, it is persistent and

perpetually available in the same condition as it was when submitted. A variety of checksum algorithms are available for detecting data corruption. These algorithms convert strings of different lengths of data to short, fixed-size results. Since they are designed to avoid collisions, finding two strings with the same hash result is almost impossible. CRCs, Adler32, and Fletcher's checksum are examples of non-cryptographic hash algorithms for detecting unintentional modification of data. Other examples of hash algorithms capable of detecting intentional modification include secure hash algorithms (SHA0, SHA1, SHA2) and message digest algorithm (MD5). They use cryptographic hash functions that are usually complex and computationally expensive. Such algorithms are typically used to encrypt data or to provide digital signatures for files that are transferred over hostile networks. This study concentrates on identifying unnoticed unintentional data corruption which occurs primarily because of hardware faults present in the storage medium or bugs in the software controlling them. In our work, we use the CRC algorithm for evaluating data integrity because of its easier implementation.

Several corruption detection mechanisms exist for handling disk-related errors within the hardware itself. For example, most near-line disks provide a 64-byte data integrity segment after every block of 4KB for storing checksums [41]. Hard drives also set aside extra space for storing error correction code (ECC) for each sector that can be used to detect and correct misreads or corrupted data. However, studies have shown that not all of these errors are always detected by the system [42, 41, 75]. CRCs and checksums provide check codes that help uncover such errors. These data verification codes are calculated inline with a write request and written to disk for validation at a later time. For every change to the data, its verification code is also updated accordingly. During a read operation, the hash code is regenerated for the data read from the disk and verified with the code saved during write. If they match then the data is complete and verified; otherwise, a read error is sent to the

application.

In this chapter, we study the cost of providing integrity checks that might occur on faulty storage hardware in context of a parallel file system. Similar bit-flip errors could occur on faulty network cards or interconnects. Detection of such faults during network transmission would be better handled at PVFS's BMI abstraction layer, a study of which is beyond the scope of this thesis.

7.2 Reliable Clustered Storage

To provide complete availability of the system during a single-node failure scenario, the file system must be able to regenerate the data from the remaining nodes. This can be done using long existing RAID techniques. Parity blocks are extra bits of information calculated by performing XOR of all data blocks in the strip. Later, if any of those data blocks are missing, it can be regenerated using the calculated parity block. In our work, we have developed a RAID strategy for generating parity object on the client for both parallel file systems mentioned earlier. Our implementation is based on the RAID4 algorithm and can be easily modified to RAID5 using minor additional programming effort.

In a naive implementation, one can generate the parity object in line with the write operations handled by the client. Object-based parallel file systems like Lustre and PVFS operate on larger blocks of data compared to a single-node file system. On single-node systems, parity calculation is usually offloaded to the RAID adapter which manages all the disks attached to it. RAID adapters have direct block-level access to the disks and since all I/O operations flowing through it are in smaller block sizes, a small memory is enough to cache all the data blocks and generate parity. However, in parallel file systems, large object sizes make the process of generating parity a little different from that on RAID adapters. Instead of caching all the objects of a strip sent for write in the node's local memory, we generate parity with each object only

when they pass through the I/O module and preserve it until the last object has been submitted to the node for I/O. At that point, the parity object is flushed to the parity node for write. This method is however very costly, particularly during the re-write operation. To maintain consistency during re-write, we will have to perform two reads to read the old data and parity and two writes to commit the new data and the parity. This behavior was concurred by our experiments and will be discussed later in Section 7.4.

To avoid the costly step of doing the extra two reads and two writes during the re-write operation, we have implemented another strategy with the idea of using a thread to handle the operation independently. On the client, we introduce a thread in the file system which would identify and perform the reads and writes required to generate the parity. Since the extra reads are required only when we are updating a part of the strip, only the requests operating on partial strips are handled by this thread. The file system writes the partial strip's data on the temporary space available on the client's local disk. The new thread will then assume the task of generating the parity and updating the nodes. The partial data must be written to the local disk to ensure successful recovery of the data in case the target holding the primary copy of the data fails before the thread could complete generating the parity. Information regarding this partial data is also added to the file's metadata so that any reads occurring before before the thread completes the parity update can be directed to the local disk of the writing client. The thread decides how to generate parity for the new data based on the length of the strip being modified. For changes limited to only one node, the thread sends the new data to the storage node in charge. The node, upon receiving the data, reads the old data and generates the intermediate parity object. This parity object is then sent back to the thread on the client which then sends it to the parity target. The parity target updates the parity object by calculating the new parity for the strip. By doing this, we save one network data transfer operation. Two read and

two write transfers are reduced to just one read and two write data transfers. We also offloaded the expensive parity calculation to the storage nodes. The read of old data is now done locally on the data target node, and the parity is also generated on the parity target node. For changes spanning beyond a single node, the thread reads the data from the unchanged parts of the strip, calculates the parity on the client and updates the storage nodes and the parity node with the new data.

7.2.1 Writes Under Failure

In the context of a distributed system, writes are more interesting than the reads, particularly with parity striped writes. In this section, we discuss methods to ensure that the write operations will succeed even under a single node failure. In most distributed or clustered file systems, if a failure occurs during a write or an update operation, the request is either blocked and kept waiting for the failed node to return, or is terminated immediately. Even in the case of Lustre, it is only the new files that can be created successfully with the file’s data striped across the remaining nodes. Any write to update any existing file waits until a timeout occurs. In our work, we changed this behavior in Lustre and PVFS to continue to create files with its data striped across all listed nodes, irrespective of the node’s current status.

To handle the failure case and to ensure that the system returns to its full capacity upon return of the failed node, even for the writes which were missed during the node failure, we introduce a database within the file system called the *dirty region database* (DRD). This idea is similar to the “fact server” defined by Bright et al. in [47]. The DRD contains the information about the data which could not be written to the target during the write operation, but had its parity calculated and stored on the parity node. DRD is stored on the first and last node of the strip across which the file is spread. We store two copies of DRD to handle the case where one of the nodes containing the DRD fails. A failure of both the DRD nodes would result in

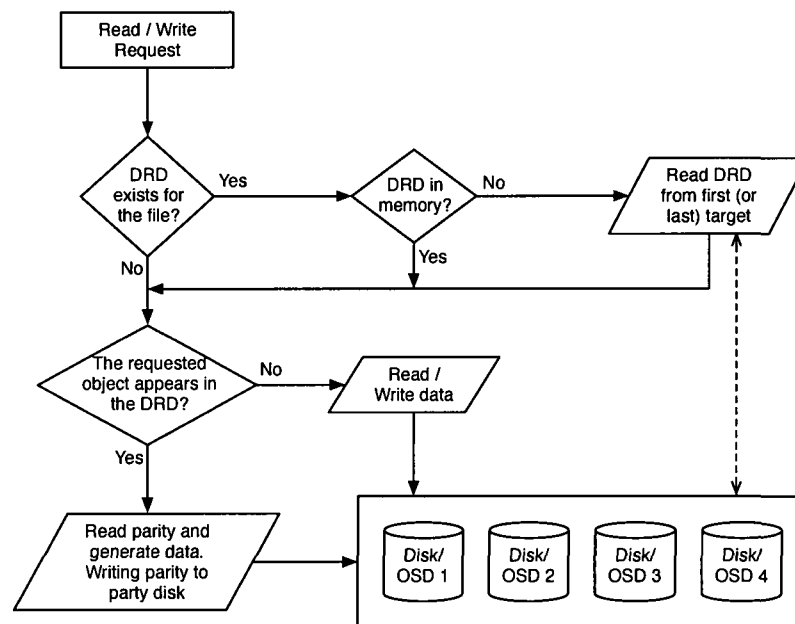


Figure 7.2: Handling of I/O operation using dirty region database (DRD).

a two-node failure scenario, the handling of which is beyond the scope of this work. The DRD also contains information regarding the incomplete strips which are being written to the local disk and will be transferred to the dedicated node by the new file system thread. We refer to these regions of the file as the *dirty regions*. For any file, information regarding the existence of the DRD is present as a flag in the file's metadata structure and is also updated on the MDS. The DRD is stored as a regular file on the storage node. Since this file is already mirrored on another node, local redundancy to protect this file is not required. Under a multi-user scenario, a lock would have to be set on the database to keep it consistent across different client accesses.

Before starting any reads or writes, the file system checks if a DRD entry exists for that file. If it does, then the client will have to read the DRD from the storage node into the client's local memory. This needs to be done only once for that operation, as the DRD entry is locked to that particular process and cannot be modified by any other file I/O thread on the client machine. The DRD will let the client know about the dirty regions present in the file and it will automatically change read requests for the file's dirty regions to be sent to the parity node to regenerate the data using parity. Even if the failed node returns to operation, the DRD will let the client know that read requests must be satisfied by regeneration because the data present on the failed node is old and inconsistent. Along with it, an update request for that region is posted on the request queue in the thread, which updates the storage nodes containing dirty data with the new data. A flow chart showing the handling of read/write operation when using the DRD is shown in Figure 7.2.

7.2.2 Reads under failure

In our algorithm, reads under failure is very similar to the handling of reads during a disk failure in a single node RAID system. While doing a read, if the client notices

that a node has failed, it will regenerate the data by reading the contents of the remaining nodes and the parity. This operation is done from within the file system and is not visible to the user or the application. The client issues the read requests to the remaining nodes in the stripe, including parity node, for data belonging to that offset within the object of that particular stripe. When all the data is available, the client can simply XOR them to regenerate the original data. However, if this object had recently been re-written and was currently being used by the thread to generate the parity, the system would simply read the data from the local disk without going to the target. The system can easily check if this data is available on the local file system by checking the DRD.

It could be contended that this scenario under large clusters could cause the DRD to grow to very large sizes, and hence could affect the performance due to long search times. However, since the thread is running constantly to clear the DRD, the DRD size does not typically grow large.

7.3 Detecting Data Corruption

Adding checksum capability to a parallel file system brings its own challenges. With data spread across several servers, constantly updating CRCs and verifying reads with the latest CRC values becomes a challenge. Further, all of these tasks must be done with minimum impact on performance of the parallel file system. File systems such as Btrfs [4] provide both data and metadata integrity checks and could be used as the underlying file system for PVFS. However, Btrfs is currently under heavy development and not yet ready to be deployed with PVFS. Such a setup also ties PVFS with other file systems and limits its portability. On the other hand, providing CRC checks within PVFS allows taking advantage of user space caching on servers. It also gives PVFS the flexibility of setting its own CRC's buffer size. This design can also be extended to handle verification only for critical files, without

affecting performance for others. Using MPI hints or POSIX extended attributes, applications could provide PVFS information as to which files require validation.

The hash code for each file can be stored either on an MDS along with the file's metadata or on a storage node along with the file's data. Storing the CRC on the MDS allows us to use the client's usually higher processing power compared to storage nodes for calculation of the hash code. Storing the CRC on the MDS also gives us a single point of contact for all data integrity checks. However, this also means that the CRCs will have to be updated atomically with every change of data. In large environments with multiple clients, doing frequent updates creates a single node bottleneck. Although the checksums are small in size even for a very large file and can be easily cached in the client's memory, in a multiple-client environment with multiple writers of the same file it is necessary to always have the latest copy of the CRC code, making the metadata server a major bottleneck.

Alternatively, placing the hash code on storage nodes along with the data allows us to take advantage of distributing the CRC computation across several nodes. It also allows us to take advantage of the node's local cache by placing the CRCs in memory after writes and avoiding CRC reads for subsequent read requests. The I/O overhead of providing integrity checks will thus be visible to the user only during writes. Any following reads will require only obtaining the CRC from the local memory and validating the data. For these reasons, in our work, we chose to place the check codes on storage nodes. We modified the DBPF Trove implementation in PVFS (version 2.8.1) to create a *cksum* file for every *bstream* file that is saved on the storage server to store the CRC codes. CRCs are calculated for every fixed chunk size of a *bstream* file, also referred to as CRC buffer size. The size of this chunk can be changed during the configuration of PVFS. All CRC codes for the data file are saved sequentially in the same order in the *cksum* file as shown in Figure 7.3.

In our implementation, we used multiple threads to achieve maximum performance

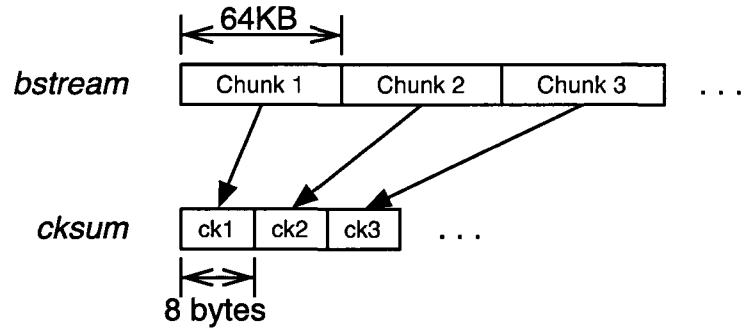


Figure 7.3: *CRCs are stored in a `cksum` file in the same order of data chunks appearing in the `bstream` file.*

for calculating the CRC code. During reads, we use the threads to read-ahead the *cksum* block if it is not already available in memory. This process is done in 4 KB block sizes even if only one CRC code is required. The remaining codes are stored in PVFS's CRC cache instead of being discarded. They remain in the cache until a CRC watermark, another configurable parameter, is reached, after which it is discarded based on a simple LRU algorithm. Once the read is complete, the thread simply checks for correctness of the data using the hash code. For writes, we use the threads to calculate the CRC code in parallel to the write operation. The CRC is updated on disk after the write is complete but before the handle is returned to the client. Any failure between results in a failed write and an error is reported to the application.

Variations in request size and offset create a huge challenge in handling I/O on servers. Nodes can receive requests that may or may not be aligned to the CRC chunk offsets. The offset of the request plays an important role in how expensive the CRC computation will be. The reason for the variation in computation of the hash code for each each of them is explained in detail below.

7.3.1 Aligned requests

Since the CRCs are always calculated for the complete CRC buffer length, having request offsets aligned to the buffer offsets translates into doing a single I/O of full

CRC buffer length on *bstream* and a single I/O on *cksum*. Requests on a *cksum* file will not be necessary if the hash code is already available in the node’s cache. Data read from the disk can be verified with the hash code read from *cksum*, while data written will have its corresponding CRC updated on disk and also preserved in cache.

7.3.2 Unaligned requests

Because of an offset to the CRC data buffer, unaligned requests need to be handled in a special way. In order to check the integrity of the data during reads, the complete CRC buffer has to be read from disk and verified, although only a part of it is returned to the client. Unaligned writes require a more complex operation. The partial blocks of the request will have to be read and verified before being updated with new data. The CRC will then have to be calculated for the updated block and flushed to the disk. If we do not verify the blocks with partial offsets and simply overwrite them, we may fail to detect any corruption that might already exist in the region with old data. This could place an extra overhead on calculating CRCs. The idea of appending the CRC code for updated data to the end of the CRC file is used by the Google file system [64], but it requires keeping track of each updated region and cleaning the CRC file for inactive chunks during the server’s idle time.

7.4 Results – Single Node Failures

In this section, we discuss the results of our implementation for both file systems under single client and multiple client scenarios. For our implementation, we modified the Lustre file system version 1.4.9 and PVFS version 2.8.1. Since Lustre provides a POSIX interface, we tested the throughput of the system using the popular file system benchmarking tool IOZone [14] by varying file sizes. We tested the throughput of PVFS with our algorithm using another benchmarking tool IOR developed at Lawrence Livermore National Laboratory (LLNL) to measure the aggregate I/O

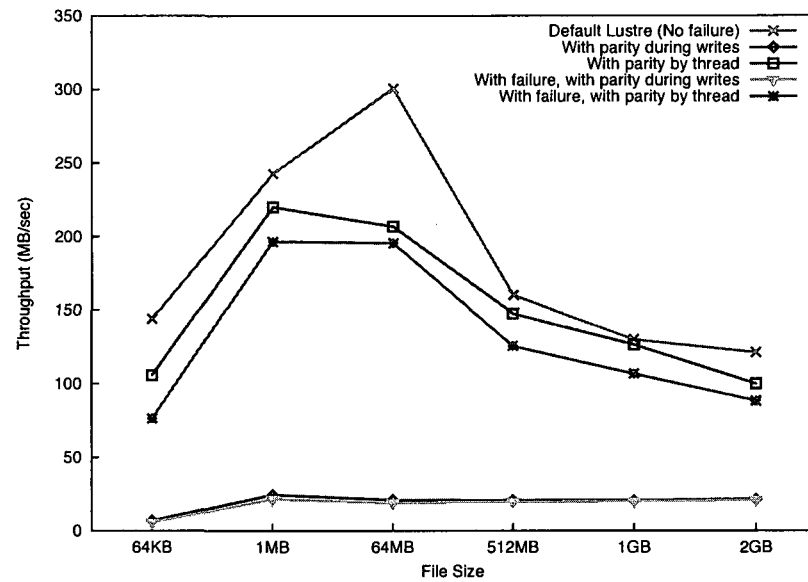


Figure 7.4: Writes on Lustre file system

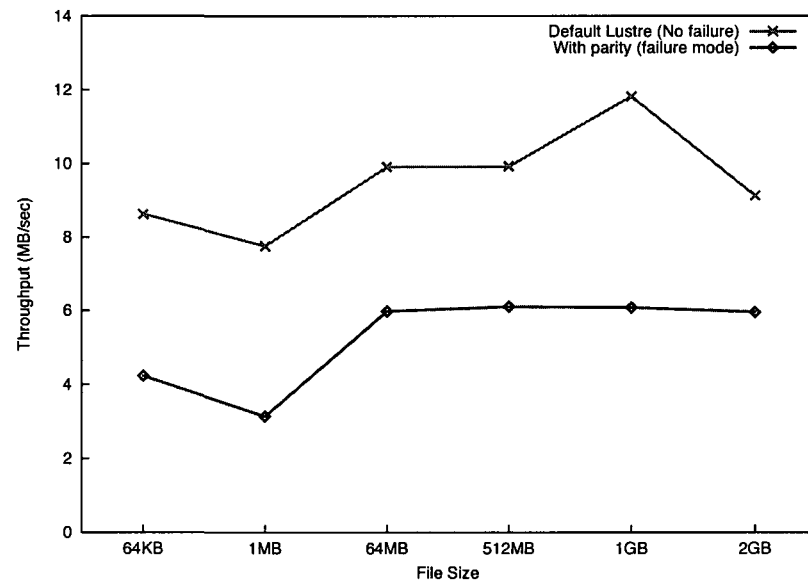


Figure 7.5: Reads on Lustre file system

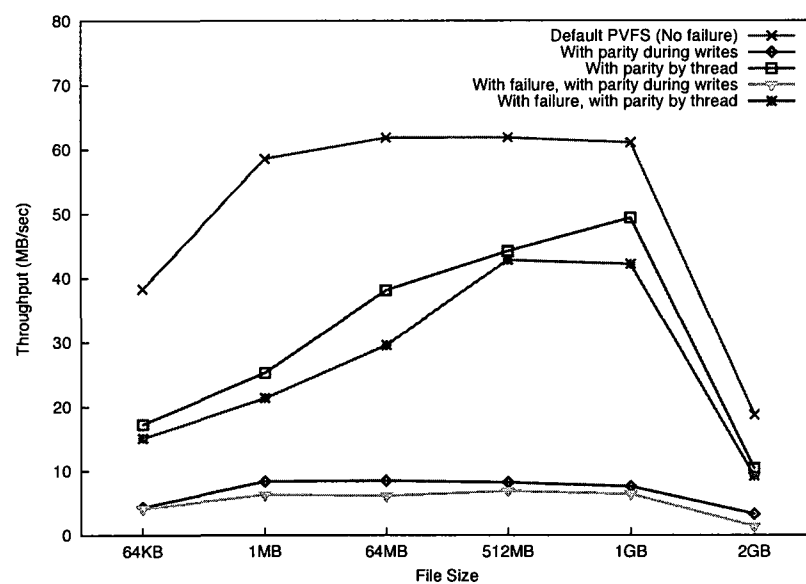


Figure 7.6: Writes on PVFS file system

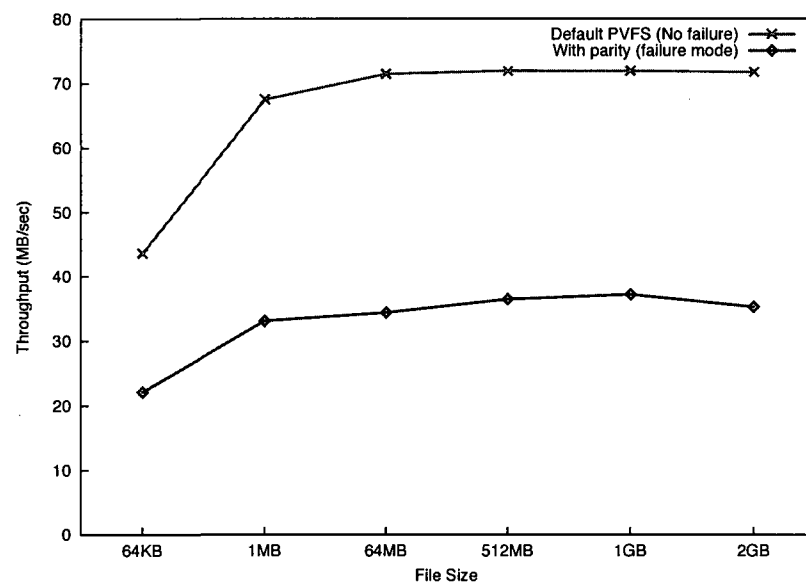


Figure 7.7: Reads on PVFS file system

throughput [13]. IOR is a benchmarking tool for parallel file systems that allows I/O through a variety of interfaces including POSIX and MPI-IO. In our experiments, we conducted the tests using the MPI-IO library with unaligned requests for transfer sizes of 64KB.

7.4.1 Single Client

For single client tests, we conducted our experiments on 7 dual-core AMD Opteron dual-processor machines with 2 GB RAM and a 80 GB SATA disk drive connected over a gigabit Ethernet port. Five nodes were used as storage targets, one as the metadata server and one as the client. The original file system was tested with 5 storage node and for testing our algorithms using RAID mechanisms, we had 4 storage nodes and 1 parity node. For both file systems, we used the same object size of 64 KB, thus setting the total strip size to 320 KB for the default file system installation and 256 KB for the RAID setup.

7.4.1.1 Lustre File System

From Figure 7.4 we can see that, as expected, calculating the parity inline with the write operation by buffering the full strip is very expensive. This is due to the reads involved to generate the parity. We see that throughput for large file sizes at best reaches up to 18% of Lustre without parity. For smaller file sizes, the performance was even worse. However, if we use a separate thread to perform the partial I/O as proposed in this work, the performance improves significantly. The parity performance with the thread technique ranges from 68-97% of normal Lustre. The thread which handles the writing of partial stripes and updating the parity, flushes out data continuously. Since this is done using an independent thread, the impact of the parity update is not visible on the write throughput. On average, the degradation in performance due to using parity is about 15%. The overall load on

the system due to the thread was minimal because the parity was mostly generated on the storage node. We see a drop in performance for very large files because the size is most likely overwhelming the memory on the storage nodes.

For testing under a failure scenario for read and for write, we modeled the failure condition of a target by modifying the time-out interval and forcing the client to view a random target as a failed node. Figure 7.4 also shows the performance of writes under a single node failure scenario compared to the normal Lustre file system performance. It can be seen that under failure, the writes do not differ much from no failure condition. It can also be observed that the use of the thread does improve reconstruction time significantly. The little loss of performance is due to writing of the DRD which is done only under failure cases. For testing read throughput, we conducted our experiments with direct I/O enabled to avoid Lustre reading the data from the Linux kernel's VFS cache on the client. In terms of the read under failure condition, we observe regenerating data using parity could incur a performance penalty in the range of 40-65% of reads when there is no failure, as can be seen in Figure 7.5. Considering that we were able to maintain the availability of the system despite failure of one node, a 40-65% performance hit during reconstruction is not significant. We can improve this number further by implementing advance reconstruction algorithms and read-ahead of the parity data and is part of our future work.

7.4.1.2 Parallel Virtual File System (PVFS)

Figure 7.6 shows the write performance for PVFS with our algorithm's implementation for both with and without failures. The graphs seems to follow the same pattern as with the Lustre file system. We observe that under normal working condition, with threads, we achieved redundancy with about a 20% hit for 1GB files. For smaller file sizes, the performance was noticeably poorer because of extra overhead

in communication. Without using threads, we can clearly see the poor performance, reaching at best of 18% of regular PVFS, again only for larger file sizes. By using threads, we obtained performance as high as 81% of regular PVFS for 1GB file sizes. Under failure, we observed an almost constant degradation of 60% for all file sizes. Again, as was the case with Lustre, this is an acceptable performance loss as the system was still available throughout the downtime of the failed node. Similarly, we see a drop in performance for very large file due to cache effects.

Figure 7.7 shows the read performance under failure condition. Unlike Lustre, for PVFS, we see very little variation in read performance under failure. The results show a leveled degradation of around 50% for all file sizes.

7.4.2 Multiple Clients

We conducted another test to monitor the performance of the system with 8 clients doing simultaneous I/Os on the same set of target nodes. All machines used for the test were dual core AMD Opteron dual-processor with 2GB RAM. Eight servers were used as clients, one as the metadata server and five as storage nodes.

Figure 7.8 shows the write performance of our algorithm as compared to the original Lustre performance when put under stress by 8 clients simultaneously on different files. We ran the benchmarking tool IOZone on each client for different files. From Figure 7.8, we can observe that with multiple clients, write performance is not as good as the single client results. This is because each client is operating on separate files, thus putting more load on each storage server. Without threads, we observed very poor performance reaching at best 17% for 1GB file writes. With threads however, we managed to get a performance of about 70% for small file sizes and a little over 50% for larger file writes. This is in contrast to the 3% best performance hit seen on single client experiments. This is because more I/O is handled by the storage nodes. A similar trend can be seen under PVFS in Figure 7.9. We observe a performance

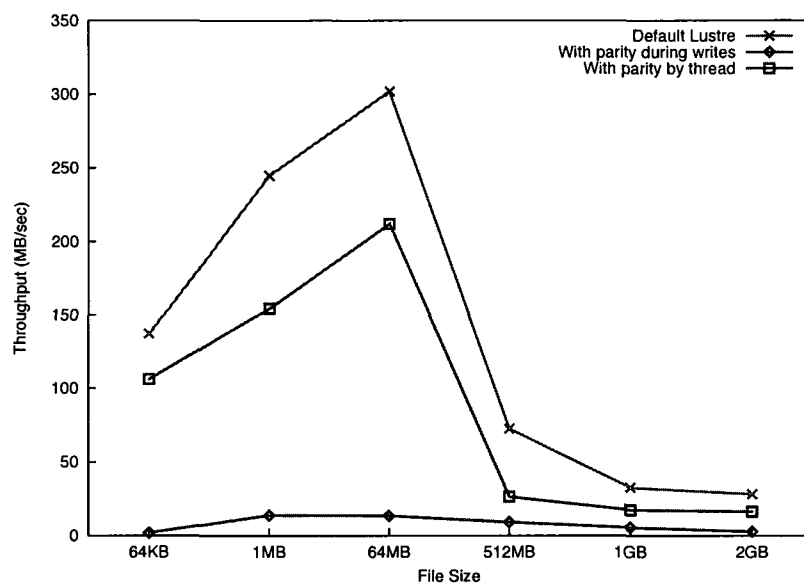


Figure 7.8: Write performance with 8 clients on Lustre file system

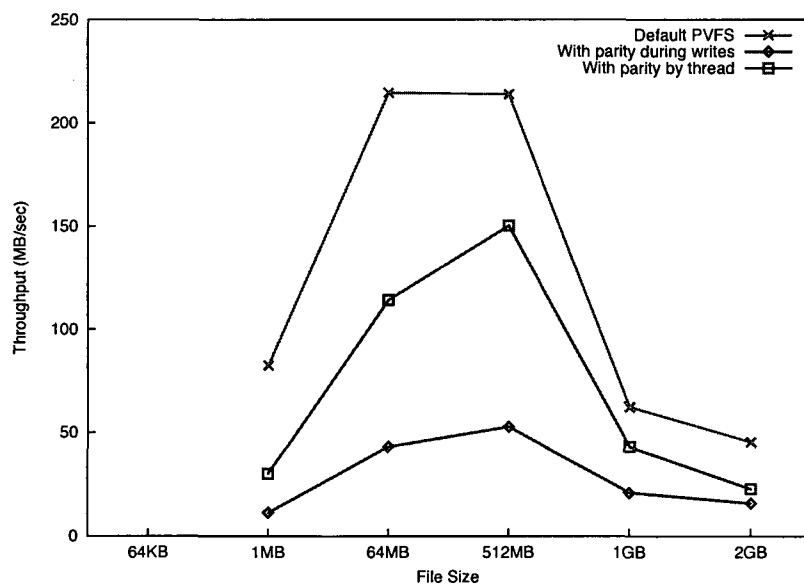


Figure 7.9: Write performance with 8 clients on PVFS file system

loss of about 30% compared to regular PVFS setup when using threads for larger file sizes. Without threads, we could only get 35% performance of the default PVFS install. Using RAID5 instead of RAID4 as we have done may improve these numbers because RAID5 will help distribute the parity calculation load equally across the storage servers.

7.5 Results – Silent Data Corruption

We ran our experiments on a Linux cluster with 8 PVFS I/O nodes serving both as storage servers and as metadata servers. Each node has two dual-core AMD Opteron 2.8 GHz processors with 4 GB RAM. Each PVFS server was configured to use an XFS file system on RAID5 with four 80 GB disks for local storage. We set the CRC buffer size to 64 KB. To test the performance of our system, we used the IOR benchmark developed at LLNL to measure the aggregate I/O throughput [13]. IOR is a benchmarking tool for parallel file systems that allows I/O through a variety of interfaces including POSIX and MPI-IO. The tests were run for single client and multiple client scenarios doing I/O on a 8 GB file using the MPI-IO library. The results shown here are an average of 10 runs with the error bars indicating the maximum and minimum measurements for each setup. The CRC cache was available in all cases.

We tested the performance of our implementation in two different scenarios and compared it with results from the production release of PVFS. Figure 7.10 shows the write throughput for a single client writing to an 8 GB file without data sync for different transfer sizes. We observed that for aligned requests, with an overhead of 36% for 4 KB transfer sizes and an overhead of 40% for 4 MB transfer sizes, we were able to provide integrity checks on the data stored. Contrary to our assumption of higher overhead for unaligned requests, we observed only a little extra overhead of about 6% over aligned requests for 4 MB transfer sizes.

Figures 7.11 and 7.12 show the read and write bandwidth for 8 clients with the

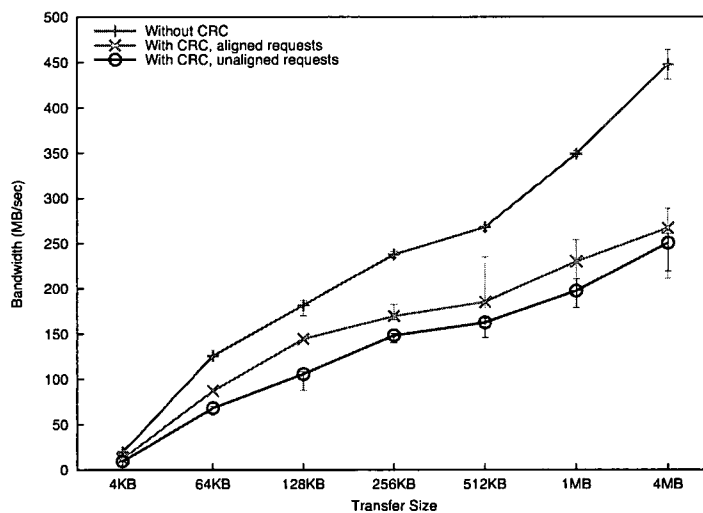


Figure 7.10: IOR aggregate Write bandwidth with 8 GB file using MPI-IO. (with sync, single client)

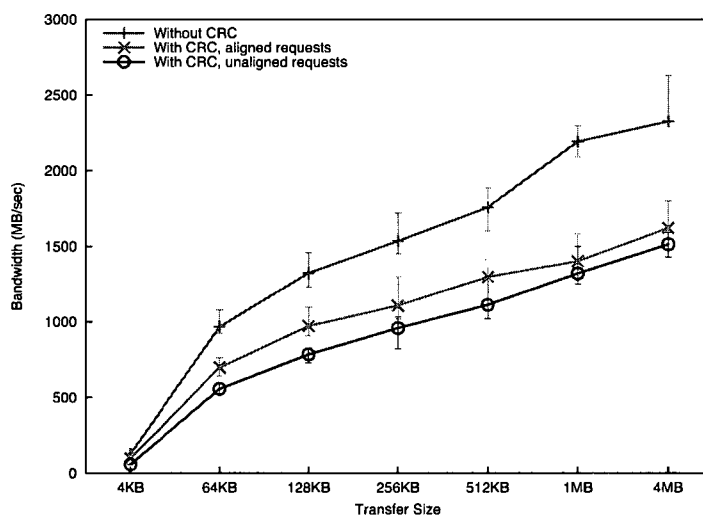


Figure 7.11: IOR aggregate Read bandwidth with 8 GB file using MPI-IO (without sync, 8 clients)

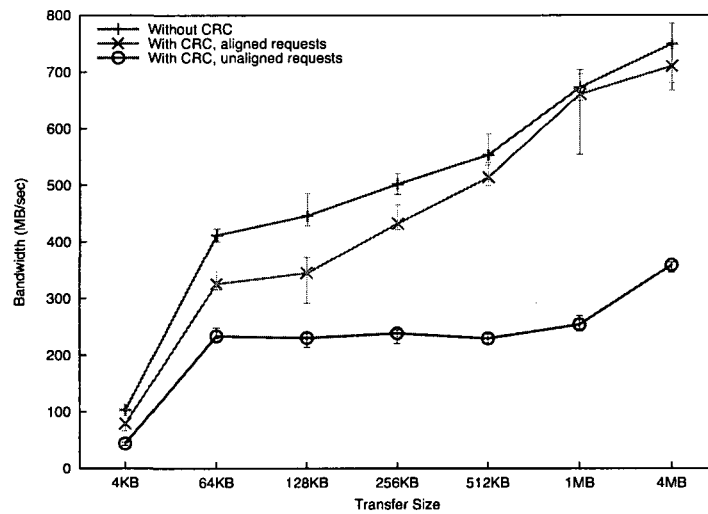


Figure 7.12: IOR aggregate Write bandwidth with 8 GB file using MPI-IO. (without sync, 8 clients)

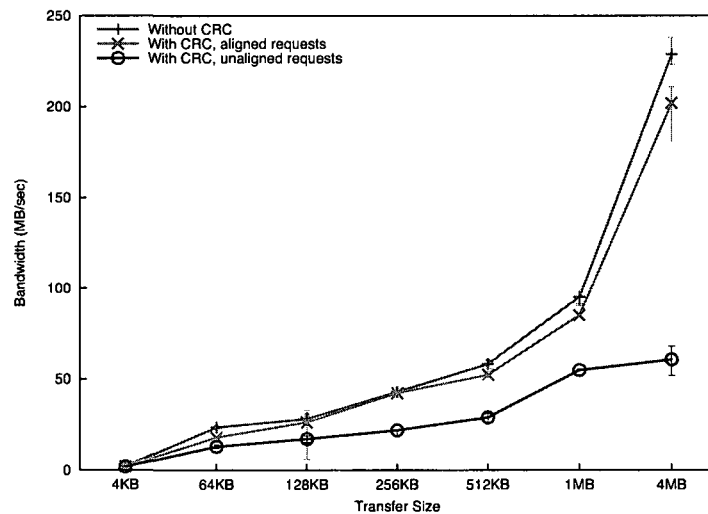


Figure 7.13: IOR aggregate Write bandwidth with 8 GB file using MPI-IO. (with sync, 8 clients)

data synchronization turned off and I/O being done asynchronously. We observed a read overhead of 22% for 4 KB transfer sizes and a 30% overhead for 4 MB transfer sizes. For writes, the overhead varied between 24% for 4 KB transfer sizes to 5% for 4 MB transfer sizes. Most of this overhead was due to the computation time of the CRCs. We observed that for reads, the threads would almost always complete the read-ahead of CRCs before the data was read from the *bstream* and for writes, the threads would wait for CRC calculation. Having a separate file for storing CRC codes thus had little to almost no impact on performance. In our experiments, reads took a larger hit compared to writes because for reads we compute the CRC after the data is read, while for writes we do the computation in parallel. For unaligned write requests, we observed a larger overhead of 57% for 4 KB transfer sizes and 52% for 4 MB transfer sizes. This overhead is because of reading the old data with partial requests from the disk and calculating the CRC to verify the existing data before updating it with the new data. This overhead was not visible in the case of a single client because fewer requests were being handled by the I/O nodes.

Figure 7.13 shows the performance when I/O on *bstream* was done synchronously by 8 clients. Writes followed almost the same pattern as in the case of requests without sync. Aligned write overhead varied from 5% for 4 KB transfer sizes to 12% for 4 MB transfer sizes. Although writes for aligned requests gave throughput almost as good as the default PVFS, unaligned requests took a large hit. Again, this difference is caused by the read-verify-write steps required for unaligned write requests. For unaligned writes, the overhead changed from 9% for 4 KB to 73% for 4 MB transfer sizes. For reads (results not shown here), we observed very little variation in the throughput between aligned and unaligned requests. In our experiments, we observed an overhead of 43% for aligned requests of 4 MB transfer sizes and 45% for unaligned requests of 4 MB transfer sizes.

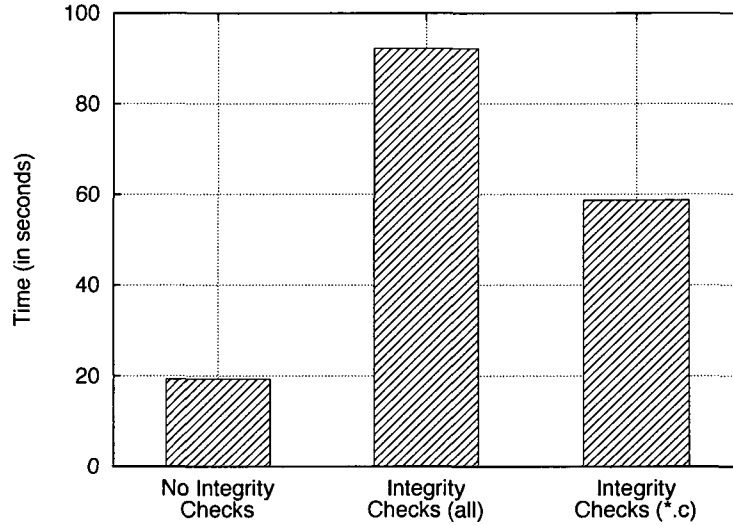


Figure 7.14: *Data Integrity enabled for all .c files.*

7.6 Results – Reliable Storage using ATTEST

One of the main aim of ATTEST framework was to allow expensive reliability algorithms to be enabled selectively on important files, while leaving the regenerable, or unimportant files with default access. To see the real advantage of using ATTEST framework to selectively enable reliability algorithms, we noted the time to compile *am-utils* utilities suite on a PVFS mount under three different scenarios. Figure 7.14 shows the comparison of the time taken to compile the source code when no integrity checks were enabled, when integrity checks were enabled on all files and when the integrity checks were done only on .c files present in the source code. From the figure, we observe a significant gain by using rules on ATTEST in contrast to having the integrity checks enabled on all the files. By using rules, we were able to reduce the compile time by almost 36%.

7.7 Summary

In this chapter, we have presented a series of algorithms for a fault-tolerant distributed storage system. The algorithms preserve data and file system integrity and consistency in the presence of concurrent reads and writes. We have shown how these protocols are designed to be tolerant of single-fault transient failures. The protocols are designed with fault-tolerance as a principal design commitment rather than as an afterthought. The main contribution is the notion of storing data locally to a client as well as to the cluster storage nodes in order to reduce the impact of parity generation on writes. We have shown that this improves the performance significantly.

We also gave an overview of the cost of providing data integrity in a parallel file system. We showed that integrity errors exist and can lead to unexpected results with data loss if not identified immediately. We provided a prototype implementation for integrity checks in PVFS and demonstrated that, with an overhead of around 5% for writes and 22% for reads, we can ensure that the data is free from any errors.

New emerging technologies such as flash translation layers (FTLs) that allow arrays of NAND flashes to be addressed in logical sectors are finding their way into the HPC domain because of their ability to do random data access at several order of magnitude faster than traditional drives. Despite having wear-leveling algorithms and efficient bad-blocks management, however, they are not free from data errors. Flash-based storage has a limited number of write cycles before it wears out and stops storing correct data. This situation emphasizes the need for having a mechanism for data integrity checks for them in place. Our work also shows that the biggest issue of providing such verification checks lies with the cost of computing check codes and not in storing them on disks. We plan to modify the way we calculate the CRCs by establishing a read/write request pipeline, thus using each individual component of the system more efficiently. We are also studying ways in which we could make use of highly parallel general-purpose Graphical Processing Units (GPGPUs) to compute

integrity checks. These devices suffer from memory transfer overheads. However, recent studies have shown that using a pipelined architecture to keep the GPU buffers busy, several magnitudes of performance improvement could be achieved [55].

Chapter 8

Conclusions

In this thesis, we have presented the motivation for extending storage system policy administration to the file level thereby providing opportunities for efficiencies in terms of storage overhead and performance at several levels of the storage hierarchy. User-directed storage policy through the use of extended attributes can assist storage administrators in determining efficient storage usage and server allocation. We have outlined an attribute-based extendable storage framework that allows users to set file-specific, or directory specific rules which can enable, or disable functionalities present within the storage system. These rules are pushed to all layers of the storage stack, including device manager layer, thus also enabling transfer of throughput or redundancy requirements of the device from the user to the storage system. We provide a user-space stackable file system design which allows non-privileged users to develop their own file system in user-space. These file systems are stacked on top of existing file systems, thus eliminating the need to depend on system administrator to enable support for a particular feature.

We also presented algorithms to protect loss of data due to failures, or detect data corruption due to hardware faults. These algorithms although computationally expensive, improve the reliability of storage systems. Since all files stored on a storage server do not require such high reliability, using ATTEST, we limit the functionality

overhead to only important files.

As storage size and administration costs go up, we believe that one size fits all model no longer provides the most efficient solution to storage management. User-directed policy models such as what is presented in this thesis is required. File-level granularity on storage policy rules and allowing user-space stackable plugins provide the users, as well as administrators with flexibility in decision making. ATTEST framework has the potential to reduce the burden on system administrators at sites where such extendable clustered storage systems are deployed.

As future work, we hope to improve the way the communication occurs between the application and the storage device. The current approach to storage has always been a one-way communication with applications providing the storage system with the data to store. We hope that this work forms the foundation for a two-way communication and help design storage-aware applications. In such a system, an application could query as to which plugins are available, and what features an application can take advantage of. The applications could request the file system for all available options that can improve the quality-of-service of the storage system. ATTEST can be designed to dynamically predict properties of files based on its attributes so that more intelligent decisions could be made on placement of data blocks [59]. Properties such as file size and file growth could be used to help system administrators gain more control on user's files.

We also hope our work helps design and development of future active storage. Active disks [109, 81, 31, 60] are used in environments where the processing power of the drive controller can be used to solve parallel problems by operating on a subset of data. ATTEST framework can be used to communicate the type of function already present on the disk to be executed on the data.

We are also looking into the use of object storage to interpret ATTEST attributes. OSD attributes present in object-based storage device commands present in T10

standard could be modified to implement ATTEST framework.

Apart from the ATTEST framework, the user-space stackable file system design can be extended to the device manager layer that will send data blocks available at the disk level to the user-space daemon in a manner similar to that done at the file system level. This can be used to write user-spaced disk block manipulation functions such as data deduplication or snapshotting. A block mapping function in the user-space can also allow users to write their own disk layout algorithms [129, 100] from the user space without the need to understand the operating systems internals or working of device drivers.

Bibliography

- [1] ADIO: Abstract-device interface for I/O. <http://www.mcs.anl.gov/~thakur/adio>.
- [2] Application-aware storage promises intelligence and automation. http://searchstorage.techtarget.com/generic/0,295582,sid5_gci1267371,00.html.
- [3] The Berkeley Automounter Suite of Utilities. <http://www.am-utils.org>.
- [4] Btrfs: Copy-on-write file system for linux. http://btrfs.wiki.kernel.org/index.php/Main_Page.
- [5] CurlFtpFS - a FTP filesystem based on cURL and FUSE. <http://curlftpfs.sourceforge.net/>.
- [6] DataDomain's CEO takes on Nearline Storage. http://searchdatabackup.techtarget.com/news/article/0,289142,sid187_gci1299845,00.html.
- [7] EXT4 File System. http://ext4.wiki.kernel.org/index.php/Main_Page.
- [8] FLASH - ASC Center for Astrophysical Thermonuclear Flashes. <http://flash.uchicago.edu/website/home>.
- [9] FUSE: Filesystem in user space. <http://fuse.sourceforge.net>.
- [10] Generic I/O control operations. http://www.gnu.org/s/libc/manual/html_node/IOCTLs.html.
- [11] HIPAA - Health Insurance Portability and Accountability Act. <http://www.hipaa.org/>.
- [12] HTTP filesystem. <http://httpfs.sourceforge.net/>.
- [13] IOR HPC benchmark. <http://sourceforge.net/projects/ior-sio/>.
- [14] IOZone. <http://www.iozone.org>.
- [15] Lustre file system. <http://www.sun.com/software/products/lustre/>.

- [16] The message passing interface (mpi) standard. <http://www.mcs.anl.gov/research/projects/mpi/>.
- [17] MPICH2 - High Performance MPI. <http://www.mcs.anl.gov/research/projects/mpich2>.
- [18] NASA Advanced Supercomputing (NAS) Division. <http://www.nas.nasa.gov>.
- [19] OpenAtom. <http://charm.cs.uiuc.edu/OpenAtom>.
- [20] Parallel computing (wikipedia). http://en.wikipedia.org/wiki/Parallel_computing.
- [21] Parallel Virtual File System (PVFS). <http://www.pvfs.org>.
- [22] puffs - pass-to-userspace framework file system. <http://www.netbsd.org/docs/puffs/>.
- [23] ROMIO: A High-Performance, Portable MPI-IO Implementation. <http://www-unix.mcs.anl.gov/romio>.
- [24] Sarbanes-Oxley Act 2002. <http://thecaq.aicpa.org/Resources/Sarbanes+Oxley/>.
- [25] SSH filesystem. <http://fuse.sourceforge.net/sshfs.html>.
- [26] Webserver traces - soccer world cup 1998. <http://ita.ee.lbl.gov>.
- [27] Common Internet File System protocol (CIFS/1.0). <http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00>, 1996.
- [28] LVM. <http://sources.redhat.com/lvm2>, 2001.
- [29] HDF Group - HDF5. <http://www.hdfgroup.org/HDF5>, 2007.
- [30] Michael Abd-El-Malek, Chuck Cranor William V. Courtright II, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In **Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)**, San Francisco, CA, December 2005.
- [31] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In **Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**, San Jose, CA, October 1998.

- [32] Irfan Ahmad. Easy and efficient disk I/O workload characterization in VMware ESX Server. In **IEEE International Symposium on Workload Characterization**, pages 149–158, Boston, MA, September 2007.
- [33] Sadaf R. Alam, Richard F. Barrett, Jeffery A. Kuehn, Philip C. Roth, and Jeffrey S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In **IEEE International Symposium on Workload Characterization**, pages 225–236, San Jose, CA, October 2006.
- [34] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. In **Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)**, page 298, Taipei, Taiwan, April 2000.
- [35] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. **ACM Transactions on Computer Systems**, 20(1):25–48, February 2002.
- [36] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. **ACM SIGOPS Operating Systems Review**, 29(5):109–126, 1995.
- [37] Todd A. Anderson and James Griffioen. An application-aware data storage model. In **Proceedings of the 1999 USENIX Annual Technical Conference**, pages 309–322, Monterey, CA, June 1999.
- [38] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A file system to trace them all. In **Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)**, pages 129–143, San Francisco, CA, March 2004.
- [39] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35(R.1), HP Laboratories, Palo Alto, CA, September 1999.
- [40] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In **Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems**, pages 10–22, Atlanta, GA, May 1999.
- [41] Lakshmi N. Bairavasundaram, Garth Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In **Proceedings of 6th USENIX Conference on File and Storage Technologies**, pages 223–238, San Jose, CA, February 2008.
- [42] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In **ACM**

- SIGMETRICS Performance Evaluation Review**, volume 35, pages 289–300, New York, USA, June 2007.
- [43] S. Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In **IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems**, pages 16–29, April 1995.
 - [44] Steve Best. Jfs overview: How the Journaled File System cuts restart times to the quick. <http://www.ibm.com/developerworks/linux/library/l-jfs.html>, January 2000.
 - [45] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. **Proceedings of ACM SIGMETRICS Performance Evaluation Review**, 28(1):34–43, June 2000.
 - [46] Peter J. Braam and Rumi Zahir. Lustre - a scalable high performance file system. Technical report, Cluster File Systems, Inc., Mountain View, CA, July 2001.
 - [47] Jonathan D. Bright and John A. Chandy. Data integrity in a distributed storage system. In **Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)**, Las Vegas, NV, June 2003.
 - [48] Jonathan D. Bright and John A. Chandy. A scalable architecture for clustered network attached storage. In **Proceedings of 11th NASA Goddard / 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)**, pages 196–206, San Diego, CA, April 2003.
 - [49] Philip H. Carns, Walter B. Ligon, Rob Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In **4th Annual Linux Showcase and Conference**, pages 317–327, Atlanta, GA, October 2000.
 - [50] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In **Proceedings of the Workshop on Design Issues in Anonymity and Unobservability**, pages 311–320, Berkeley, CA, July 2000.
 - [51] Richard G. Clegg. A practical guide to measuring the Hurst parameter. In **ArXiv Mathematics**, pages 3–14, October 2006.
 - [52] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. In **ACM Transactions on Computer Systems**, volume 14, pages 225–264, New York, USA, 1996. ACM.
 - [53] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc

- Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. **IBM Systems Journal**, 34(2):222–248, January 1995.
- [54] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In **Proceedings of SuperComputing**, page 59, San Diego, CA, 1995.
 - [55] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Accelerating Reed-Solomon coding in RAID systems with GPUs. In **Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)**, pages 1–6, Miami, FL, April 2008.
 - [56] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In **Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP)**, pages 202–215, Alberta, Canada, October 2001.
 - [57] John R. Douceur and Roger P. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In **Proceedings of 15th International Symposium on Distributed Computing**, pages 48–62, Lisbon, Portugal, October 2001.
 - [58] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In **Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)**, page 75, Elmau, Germany, May 2001.
 - [59] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer. Attribute-based prediction of file properties. Technical Report TR-14-03, Harvard Computer Science Technical Group, June 2004.
 - [60] Evan J. Felix, Kevin Fox, Kevin Regimbal, and Jarek Nieplocha. Active storage processing in a parallel file system. In **Proceedings of 6th LCI International Conference on Linux Clusters: The HPC Revolution**, Chapel Hill, NC, April 2005.
 - [61] Jeremy Fitzhardinge. Userfs. <http://www.goop.org/~jeremy/userfs/>.
 - [62] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. **Astrophysical Journal Supplement**, 131:273–334, November 2000.
 - [63] John Garrison and A. L. Narasimha Reddy. Umbrella file system: Storage management across heterogeneous devices. Technical Report TAMU-ECE-2007-06, Texas A&M University, 2007.

- [64] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In **ACM Symposium on Operation System Principles**, pages 29–43, Lake George, NY, October 2003.
- [65] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In **Proceedings of Architectural Support for Programming Languages and Operating Systems**, pages 92–103, October 1998.
- [66] Jim Gray. A census of tandem system availability between 1985 and 1990. **IEEE Transactions on Reliability**, 39(4):409–418, October 1990.
- [67] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. In **Proceedings of the USENIX Workshop on File Systems**, pages 71–78, May 1992.
- [68] Michael Hasenstein. The logical volume manager (LVM). Technical report, SUSE Inc., 2001.
- [69] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. In **ACM Transactions on Computer Systems**, volume 12, pages 58–89, February 1994.
- [70] John L. Hennessy and David A. Patterson. **Computer Architecture: A Quantitative Approach**. Morgan Kaufmann Publishers, Inc., 3rd edition, May 2002.
- [71] HFS+. Apple’s HFS Plus volume format. <http://developer.apple.com/mac/library/technotes/tn/tn1150.html>, March 2004.
- [72] Mike Holton and Raj Das. XFS: A next generation journaled 64-bit file system with guaranteed rate I/O. Technical report, Silicon Graphics Inc., 1996.
- [73] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In **Proceedings of the 6th International Data Engineering Conference**, pages 456–465, Los Angeles, CA, February 1990.
- [74] JFS. Journaled file system technology for linux. <http://jfs.sourceforge.net/>.
- [75] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? In **Proceedings of the 6th USENIX Conference on File and Storage Technologies**, San Jose, CA, February 2008.

- [76] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing (NAS) Division, Moffett Field, CA, October 1999.
- [77] Terry Jones, Alice Koniges, and R. Kim Yates. Performance of the IBM General Parallel File System. In **Proceedings of the Parallel and Distributed Processing Symposium**, pages 673–681, Cancun, Mexico, May 2000.
- [78] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIFF: Redundant array of independent filesystems. In **IEEE Conference on Mass Storage Systems and Technologies**, pages 199–214, San Diego, CA, September 2007.
- [79] Nikolai Joukov, Abhishek Rai, and Erez Zadok. Increasing distributed storage survivability with a stackable RAID-like file system. In **Proceedings of the 2005 IEEE/ACM Workshop on Cluster Security, in conjunction with the Fifth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)**, pages 82–89, Cardiff, UK, May 2005.
- [80] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on C++. In **Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications**, volume 28, pages 91–108, Washington, D.C., September 1993.
- [81] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISks). **SIGMOD Record**, 24(7):42–52, September 1998.
- [82] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In **Proceedings of the Summer USENIX Technical Conference**, pages 238–247, Atlanta, GA, 1986.
- [83] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In **Proceedings of the 1994 Conference on Supercomputing**, pages 640–649, Washington, D.C., November 1994.
- [84] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Wesley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In **Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**, pages 190–201, Cambridge, MA, November 2000.
- [85] Julian M. Kunkel and Thomas Ludwig. Performance evaluation of the PVFS2 architecture. In **15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing**, pages 509–516, Naples, Italy, February 2007.

- [86] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In **Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**, pages 84–92, Cambridge, MA, October 1996.
- [87] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan Miller. Measurement and analysis of large-scale network file system workloads. In **USENIX 2008**, pages 213–226, Boston, MA, June 2008.
- [88] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In **Workshop on Experimental Computer Science**, San Diego, CA, 2007.
- [89] Darrell D. E. Long, Bruce R. Montague, and Luis-Felipe Cabrera. Swift/RAID: A distributed RAID system. **Computing Systems**, 7(3):333–359, Summer 1994.
- [90] Bill Maron, Thomas Chen, Bret Olszewski, Steve Kunkel, and Alex Mericas. Workload characterization for the design of future servers. In **IEEE International Symposium on Workload Characterization**, pages 129–136, Austin, TX, October 2005.
- [91] Avantika Mathur, Mingming Cao, and Andreas Dilger. Ext4: The next generation of the ext3 file system. **USENIX ;login:**, 31:25–30, June 2007.
- [92] Ermelindo Mauriello. TCFS: Transparent cryptographic file system, August 1997.
- [93] Ethan Miller and Randy H. Katz. Input/output behavior of supercomputing applications. In **Proceedings of SuperComputing**, pages 567–576, Albuquerque, NM, November 1991.
- [94] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok. Avfs: An on-access anti-virus file system. In **13th USENIX Security Symposium**, San Diego, CA, August 2004.
- [95] James H. Morris, Mahadev Satyanarayanan, Michel H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A distributed personal computing environment. **Communications of the ACM**, 29(3):184–201, March 1986.
- [96] Kiran-Kumar Muniswami-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In **Proceedings of the 2006 USENIX Annual Technical Conference**, pages 43–56, Boston, MA, June 2006.

- [97] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In **Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)**, pages 31–44, Boston, MA, December 2002.
- [98] Sumit Narayan and John A. Chandy. ATTEST: ATtributes-based Extendable STorage. **Journal of Systems and Software**, 83(4):548–556, April 2010.
- [99] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. In **IEEE Transactions on Parallel and Distributed Systems**, volume 7, pages 1075–1089, October 1996.
- [100] James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the file system with gray-box techniques. In **Proceedings of the Annual USENIX Technical Conference**, pages 311–323, San Antonio, TX, June 2003.
- [101] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglass, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. **IEEE Computer**, 21(2):23–26, February 1988.
- [102] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In **Proceedings of the ACM SIGMOD International Conference on Management of Data**, pages 109–116, Chicago, IL, June 1988.
- [103] Manoj Pillai and Mario Lauria. A high performance redundancy scheme for cluster file systems. **International Journal of High Performance Computing and Networking**, 2(2):90–98, March 2004.
- [104] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In **Proceedings of the 20th ACM Symposium on Operating Systems Principles**, pages 206–220, Brighton, UK, October 2005.
- [105] David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok. UnionFS: User- and community- oriented development of a unification file system. In **Proceedings of 2006 Ottawa Linux Symposium**, pages 349–362, Ottawa, Canada, June 2006.
- [106] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In **Proceedings of the 25th ACM Symposium on Applied Computing**, Sierre, Switzerland, March 2010.
- [107] A. L. Narasimha Reddy and Prithviraj Banerjee. An evaluation of multiple-disk I/O systems. In **IEEE Transactions on Computers**, volume 38, pages 1680–1690, December 1989.

- [108] A. L. Narasimha Reddy and Prithviraj Banerjee. A study of I/O behavior of PERFECT benchmarks on a multiprocessor. In **Proceedings of the 17th Annual International Symposium on Computer Architecture**, pages 312–317, Seattle, WA, May 1990.
- [109] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In **Proceedings of the 24th International Conference on Very Large Databases (VLDS)**, New York, NY, USA, August 1998.
- [110] David S. H. Rosenthal. Evolving the vnode interface. In **Proceedings of the USENIX Technical Conference**, pages 107–118, Anaheim, CA, Summer 1990.
- [111] David S. H. Rosenthal. Requirements for a "stacking" vnode/VFS interface. Technical Report SD-01-02-N014, UNIX International, 1992.
- [112] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In **Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)**, pages 15–30, Boston, MA, December 2002.
- [113] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In **Proceedings of the Summer USENIX Technical Conference**, pages 119–130, Portland, OR, June 1985.
- [114] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. **IEEE Transactions on Computers**, 39(4):447–459, April 1990.
- [115] Stefan Savage and John Wilkes. AFRAID: A frequently redundant array of independent disks. In **Proceedings of the Annual Conference on USENIX Annual Technical Conference**, pages 27–39, San Diego, CA, January 1996. USENIX Association.
- [116] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In **Proceedings of the Conference on File and Storage Technologies**, pages 231–244, Monterey, CA, January 2002.
- [117] Bradley W. Settlemyer and Walter B. Ligon III. A technique for lock-less mirroring in parallel file systems. In **Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid**, pages 801–806, Lyon, France, May 2008.
- [118] Josef Sipek, Yiannis Pericleous, and Erez Zadok. Kernel support for stackable file systems. In **Proceedings of 2007 Ottawa Linux Symposium**, Ottawa, Canada, June 2007.

- [119] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: Techniques and applications. In **Proceedings of International Workshop on Storage Security and Survivability (StorageSS)**, Fairfax, VA, November 2005.
- [120] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. I/O requirements of scientific applications: An evolutionary view. In **Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing**, pages 49–59, Syracuse, NY, August 1996.
- [121] Christopher A. Stein, John H. Howard, and Margo Seltzer. Unifying file system protection. In **Proceedings of the 2001 USENIX Annual Technical Conference**, Boston, MA, June 2001.
- [122] Sun Microsystems. ZFS: The last word in file systems, 2004.
- [123] Garret Swart, Andrew Birrell, Andy Hisgen, and Timothy Mann. Availability in the Echo file system. Technical Report SRC-RR-112, Digital: Systems Research Center, Palo Alto, CA, September 1993.
- [124] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In **Proceedings of the USENIX Annual Technical Conference**, page 1, San Diego, CA, January 1996.
- [125] Hong Tang, A. Gulbeden, Jinyu Zhou, W. Strathearn, Tao Yang, and Lingkun Chu. The Panasas ActiveScale Storage Cluster - Delivering Scalable High Bandwidth Storage. In **Proceedings of SuperComputing**, page 53, Pittsburgh, PA, November 2004.
- [126] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In **Frontiers of Massively Parallel Computation**, pages 182–189, February 1999.
- [127] Jon William Toigo. ‘Love Marks’ in storage. http://searchstorage.techtarget.com/news/column/0,294698,sid5_gci1235229,00.html, 2006.
- [128] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing tunable consistency for a parallel file store. In **Proceedings of 4th USENIX Conference on File and Storage Technologies (FAST)**, pages 17–30, San Francisco, CA, December 2005.
- [129] Jun Wang and Yiming Hu. PROFS: Performance-oriented data reorganization for log-structured file system on multi-zone disks. In **Proceedings of IEEE/ACM 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MAS-COTS)**, pages 285–293, Cincinnati, OH, August 2001.

- [130] Sage A. Weil. **Ceph: A Scalable, and High-Performance Distributed Storage**. PhD thesis, University of California, Santa Cruz, CA, December 2007.
- [131] Sage A. Weil, Scott A. Brandt, Ethan Miller, and Darrell D. E. Long. Ceph: A scalable, high-performance distributed file system. In **Proceedings of Symposium on Operating System Design and Implementation**, page 22, Seattle, WA, November 2006.
- [132] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In **ACM Transactions on Computer Systems**, volume 14, pages 108–136, New York, USA, 1996. ACM.
- [133] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing (NAS) Division, Moffett Field, CA, January 2003.
- [134] Qin Xin, Ethan Miller, S. J. Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In **Proceedings of 11th NASA Goddard / 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)**, pages 146–156, San Diego, CA, April 2003.
- [135] Haifeng Yu and Amin Vahdat. Consistent and automatic replica regeneration. **ACM Transactions on Storage**, 1(1):3–37, December 2004.
- [136] Weikuan Yu, Jeffrey S. Vetter, and H. Sarp Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In **IEEE International Parallel and Distributed Processing Symposium (IPDPS)**, pages 1–11, Miami, FL, April 2008.
- [137] Erez Zadok and Ion Badulescu. A stackable file system interface for Linux. In **LinuxExpo 99**, pages 141–151, May 1999.
- [138] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.
- [139] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In **Proceedings of the Annual USENIX Technical Conference**, pages 57–70, Monterey, CA, June 1999.
- [140] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In **Proceedings of 2000 USENIX Annual Technical Conference**, pages 55–70, San Diego, CA, June 2000.
- [141] Erez Zadok, Jeffrey Osborn, Ariye Shater, Charles P. Wright, Kiran-Kumar Muniswami-Reddy, and Jason Nieh. Reducing storage management costs via

informed user-based policies. In **Proceedings of 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)**, pages 193–197, College Park, MD, April 2004.

- [142] Erez Zadok, Vasily Tarasov, and Priya Sehgal. The case for specialized file systems, or, fighting file system obesity. **login: The USENIX Magazine**, 35(1):38–40, February 2010.

Vita

Sumit Narayan earned his M.S. in Electrical Engineering from the University of Connecticut in 2004 and his B.E. in Electrical and Electronics Engineering from the University of Madras in 2002. After his M.S., he joined MindTree Ltd., Bangalore, India in 2004 as Engineer for R. & D. Services (Storage) division. In 2005, he left MindTree Ltd. to join Data Storage Institute in Singapore as Research Officer where his duties involved design and development of object-based file systems specifically targeted for media servers. Later, in 2006, he enrolled at the University of Connecticut, Storrs for his Ph.D. degree. He will be joining EMC Corporation in Southborough, MA as Senior Software Engineer. His research interests include files and storage systems, distributed storage and parallel I/O systems.