DESIGNING RELIABLE HIGH-PERFORMANCE STORAGE SYSTEMS FOR HPC ENVIRONMENTS

by

Lucas Scott Hindman

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2011

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Lucas Scott Hindman

Thesis Title: Designing Reliable High-Performance Storage Systems for HPC Environments

Date of Final Oral Examination: 06 May 2011

The following individuals read and discussed the thesis submitted by student Lucas Scott Hindman, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Tim Andersen, Ph.D.	Member, Supervisory Committee
Murali Medidi, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Amit Jain, Ph.D., Chair, Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

Dedicated to my beautiful wife, Annie, whose love and encouragement made this work possible

ACKNOWLEDGMENTS

I would like to thank Amit Jain and Tim Andersen for the countless hours they spent reviewing draft after draft of this thesis as well as their patience and support in allowing me to pursue this topic of research. Thanks go to Nathan Schmidt and Casey Bullock for teaching me that designs on the whiteboard don't always work so well in production. And a special thanks to Shane Panter whose friendship and support have helped to keep me sane these last two years.

This work has been supported by the Boise State University Computer Science department as well as the Department of Defense DNASafeguard research grant.

AUTOBIOGRAPHICAL SKETCH

Luke Hindman has more than 15 years of experience working in computer technology. These years include a variety of IT roles where he learned the importance of customer service. In 2003, Luke enrolled in the Computer Science program at Boise State University. While at Boise State, Luke was heavily involved in the High Performance Computing (HPC) lab including the design, construction, and administration of the lab's 120 processor Linux Beowulf cluster. From 2003 until he graduated in 2007, Luke worked with several research groups wishing to leverage the computational power of the Beowulf cluster. These projects included atmospheric modeling, multiple genome/bio-informatics projects, and a material science project focusing on the development of a 2D/3D micro-structural model. Luke presented his work on the 2D/3D micro-structural model at the NASA Undergraduate Research Conference held at the University of Idaho, fall of 2007.

After graduation, Luke was hired as a senior system engineer by Balihoo, a multi-million dollar Internet-based marketing company, to manage their data center. This position at Balihoo required wearing multiple hats with responsibilities that included software development, system engineering, and customer support. While at Balihoo, Luke managed the complete redesign of Balihoo's production infrastructure to address application changes and scalability issues.

In 2009, Luke returned to Boise State University to complete a Master of Science in Computer Science. Luke currently works as a research assistant on the DNASafeguard project (a DOD funded research grant).

ABSTRACT

Advances in processing capability have far outpaced advances in I/O throughput and latency. Distributed file system based storage systems help to address this performance discrepancy in high performance computing (HPC) environments; however, they can be difficult to deploy and challenging to maintain. This thesis explores the design considerations as well as the pitfalls faced when deploying high performance storage systems. It includes best practices in identifying system requirements, techniques for generating I/O profiles of applications, and recommendations for disk subsystem configuration and maintenance based upon a number of recent papers addressing latent sector and unrecoverable read errors.

TABLE OF CONTENTS

ABSTRACT vii

LIST OF TABLES xiv

LIST OF FIGURES xvi

LIST OF ABBREVIATIONS xix

1 Introduction 1

1.1 Background 1

1.2 Commercial storage solutions 3

1.3 Problem statement 3

1.4 Thesis overview 5

2 Understanding Storage System Requirements 7

2.1 Overview 7

2.2 Background 9

2.3 Storage capacity and growth 9

2.4 Storage client details 11

2.5 Data details 13

2.5.1 Data classification 13

2.5.2 Storage zones 14

2.6 Applications details 17

2.7 Disaster recovery 18

2.8 Facility 19

2.8.1 Power requirements 20

2.8.2 Cooling requirements 21

2.9 Budget 22

2.10 Conclusion 23

3 Designing Reliable Disk Subsystems in the Presence of Latent Sector Errors and Infant Disk Mortality 24

3.1 The threat to disk subsystems 24

3.1.1 Infant disk mortality 24

3.1.2 Latent sector errors 27

3.1.3 Silent data corruption 29

3.2 Disk considerations 30

3.2.1 Classes of disks 31

3.3 RAID considerations 36

3.3.1 Encountering latent sector errors 37

3.3.2 Utilizing mean time to data loss (MTTDL) 38

3.4 Designing a reliable disk subsystem 41

3.4.1 Disk burn-in 42

3.4.2 Leveraging RAID 43

3.4.3 RAID scrubbing 45

3.4.4 Leveraging a hot-spare 46

3.4.5 Replacement strategies (end of life) 47

3.5 Other considerations 48

3.5.1 Quality hardware 48

3.5.2 RAID is NOT backup 49

3.6 Conclusion 50

4 Throughput and Scalability of Parallel Distributed File Systems . . 52

4.1 Introduction 52

4.2 Benchmarking techniques 53

4.2.1 Testing environment 53

4.2.2 Basic file transfer test 54

4.2.3 Block-range file transfer test 54

4.2.4 Client scalability test 55

4.3 "Parallel" distributed file systems overview 55

4.3.1 Simple file distribution 57

4.3.2 File striping 58

4.3.3 File replication 59

4.4 Parallel Virtual File System (PVFS) 61

4.4.1 Background 61

4.4.2 Architecture 61

4.4.3 Distribution techniques 62

4.4.4 Feature summary 62

4.4.5 Performance characteristics 63

4.5 Lustre 66

4.5.1 Background 66

4.5.2 Architecture 67

4.5.3 Distribution techniques 68

4.5.4 Feature summary 69

4.5.5 Performance characteristics 69

4.6 GlusterFS 74

4.6.1 Background 74

4.6.2 Architecture 75

4.6.3 Distribution techniques 75

4.6.4 Feature summary 76

4.6.5 Performance characteristics 76

4.7 Hadoop Distributed File System (HDFS) 83

4.7.1 Background 83

4.7.2 Architecture 84

4.7.3 Distribution techniques 85

4.7.4 Feature summary 86

4.7.5 Performance characteristics 86

4.8 Conclusion 90

5 Identifying Application Performance Constraints Using I/O Profiles 93

5.1 Introduction 93

5.2 Establish an I/O performance baseline 96

5.2.1 Set up the environment 96

5.2.2 Benchmark the environment 97

5.3 Generate an I/O profile for an application 104

5.3.1 Tools of the trade 104

5.3.2 Profile of an I/O bound application 105

5.4 Case Study: *seqprocessor* 109

5.5 Summary of *seqprocessor* performance improvements 116
5.6 Tuning application I/O operations 117
5.7 Profiling random I/O application 118
5.8 Profiling parallel I/O applications 120
5.9 Wrapping it up 121

6 Conclusion 123
6.1 Wrapping it up 123
6.2 Extensions of this research 124
6.2.1 Parallel file transfer 124
6.2.2 Data management: storage zones and data preservation strategies125
6.2.3 Statistical model to calculate ideal number of hot-swap disks to include in a storage system 126
6.2.4 Disaster recovery of parallel distributed file systems 126
6.2.5 High-availability configurations for parallel distributed file systems 127
6.2.6 Persistent versus non-persistent scratch space in HPC environments 127
6.2.7 Objective-C inspired dynamically generated non-persistent scratch space for HPC environments 129
6.2.8 Extended application I/O profiling 130

BIBLIOGRAPHY 132

A Storage Configuration Worksheet 137

B Data Classification 140

B.1 Large multimedia files (greater than 100MB) 140
B.2 Large text files (greater than 100MB) 141
B.3 Large compressed files (greater than 100MB) 141
B.4 Large database files (greater than 100MB) 142
B.5 Medium multimedia files (1MB - 100MB) 142
B.6 Medium text files (1MB - 100MB) 143
B.7 Medium compressed files (1MB - 100MB) 143
B.8 Medium database files (1MB - 100MB) 144
B.9 Small files (less than 1MB) 144
B.10 Large number of files (small or large) 145

C Application I/O Profiling Worksheet 146

D Atlantis Research Cluster Configuration 148

D.1 Storage node specifications 148
D.2 Network diagram 149
D.3 Chapter 4 RAID configuration and performance baseline 149
D.4 Chapter 5 RAID configuration and performance baseline 150

E Seqprocessor Application Source Code 153

E.1 Seqprocessor version 1 153
E.2 Seqprocessor version 2 155
E.3 Seqprocessor version 3 157

LIST OF TABLES

2.1 Storage zone policy definitions 15

3.1 Probability of disk failure based upon SMART data [34] 26

3.2 Comparison of desktop, nearline, and enterprise disk drive classes 32

3.3 Description of commonly used RAID levels 37

4.1 Summary of PVFS design features 62

4.2 Summary of PVFS configuration on Atlantis 63

4.3 Summary of Lustre design features 69

4.4 Summary of Lustre configuration on Atlantis 70

4.5 Summary of GlusterFS design features 76

4.6 Summary of GlusterFS configuration on Atlantis 77

4.7 Summary of HDFS design features 86

4.8 Summary of HDFS configuration on Atlantis 88

4.9 Summary of HDFS configuration on GeneSIS 88

5.1 Tools for benchmarking disk subsystems and network interconnects . . . 98

5.2 Bonnie++ throughput results for md0 on *atlantis01* 102

5.3 Bonnie++ IOPS results for md0 on *atlantis01* 102

5.4 Tools for monitoring system utilization 104

5.5 Summary of *seqprocessor* performance improvements 117

D.1 Atlantis research cluster node specifications 148
D.2 Chapter 4 throughput results for md0 on *atlantis00* 150
D.3 Chapter 4 throughput results for md0 on *atlantis01* 150
D.4 Chapter 4 throughput results for md0 on *atlantis02* 150
D.5 Chapter 4 throughput results for md0 on *atlantis03* 151
D.6 Chapter 5 throughput results for md0 on *atlantis01* 151
D.7 Chapter 5 IOPS results for md0 on *atlantis01* 152
D.8 Chapter 5 throughput results for md0 on *atlantis02* 152
D.9 Chapter 5 IOPS results for md0 on *atlantis02* 152
D.10 Chapter 5 throughput results for md1 on *atlantis02* 152
D.11 Chapter 5 IOPS results for md1 on *atlantis02* 152

LIST OF FIGURES

1.1 Anatomy of a high-performance storage system 2

2.1 Native file system client communicating directly with storage nodes over a dedicated private interconnect such as Infiniband 12

2.2 CIFS/NFS client communicating with a storage gateway over a workstation network such as gigabit Ethernet 12

2.3 Digital pictures are downloaded from a camera to storage zone A via USB 15

2.4 Video content is downloaded from a video camera to storage zone B via Firewire 16

2.5 Digital pictures are touched up and stored back in storage zone A 16

2.6 Movie is rendered from source material in zones A and B and written to zone C 16

2.7 Hi-def version is compressed and written to zone B 16

2.8 Hi-def version is written to Blu-ray disks 17

2.9 Intermediate movie files in zone C are removed from the storage system 17

3.1 Bathtub curve representing disk failure rates [55] 25

3.2 Diagram of the various layers in the storage stack 30

3.3 Probability of encountering an Unrecoverable Read Error while rebuilding an array of n+1 disk drives 39

3.4 Failure rates for a variety of RAID levels [33] 41

4.1 Example of "parallel" distributed file system architecture 56

4.2 Simple file distribution technique 57

4.3 File striping distribution technique 58

4.4 File replication distribution technique 60

4.5 PVFS basic file transfer with file striping (64KB) 64

4.6 PVFS block-range file transfer with file striping (64KB) 64

4.7 PVFS client scalability with file striping (64KB) 66

4.8 Lustre basic file transfer with simple file distribution 70

4.9 Lustre block-range file transfer with simple file distribution 71

4.10 Lustre client scalability with simple file distribution 72

4.11 Lustre client scalability with file striping (1MB) 73

4.12 GlusterFS basic file transfer using various distribution techniques 78

4.13 GlusterFS block-range file transfer using various distribution techniques 80

4.14 GlusterFS client scalability with simple distribution configuration 81

4.15 GlusterFS client scalability with 3x replication configuration 82

4.16 GlusterFS client scalability with 128KB stripe configuration 83

4.17 HDFS basic file transfer with 64MB blocks and 2x replication 88

4.18 HDFS block-range file transfer with 64MB blocks and 2x replication . . 89

5.1 Iozone test using 4KB to 16MB block sizes on files up to 4GB 99

5.2 Iozone test using 64KB to 16MB block sizes on files up to 32GB 101

5.3 NetPIPE throughput results over GigE link with the default MTU of 1500 bytes 103

5.4 I/O profile of `dd` reading a 32GB file using 1MB blocks 106

5.5 Top command output 107
5.6 I/O profile of dd writing a 32GB file using 1MB blocks 107
5.7 Top command output 108
5.8 `top` output for `seqprocessor-1` 110
5.9 `iostat` output from `seqprocessor-1` 111
5.10 I/O profile of `seqprocessor-1` with a single disk subsystem for both read and write operations 112
5.11 I/O profile of `seqprocessor-2` with a single disk subsystem for both read and write operations 114
5.12 I/O profile of `seqprocessor-2` with separate disk subsystems for read and write operations 114
5.13 `top` output for `seqprocessor-2` 115
5.14 I/O profile of `seqprocessor-3` with separate disk subsystems for read and write operations 116

D.1 Network layout of Atlantis research cluster 149

LIST OF ABBREVIATIONS

HPC – High-Performance Computing

IOPS – Input/Output Operations Per Second

SMART – Self-Monitoring, Analysis, and Reporting Technology

DFS – Distributed File System

ROMIO – A High-Performance, Portable MPI-IO Implementation

GigE – Gigabit Ethernet

10GigE – 10 Gigabit Ethernet

IPoIB – IP network protocol transported over Infiniband datalink protocol

CHAPTER 1

INTRODUCTION

1.1 Background

The Mercury project was started in October 1958, and fewer than 4 years later NASA had placed John Glenn in orbit around the earth. The level of planning and technological achievement required to make that happen was phenomenal. Now, 52 years later, we owe much of our modern technology to these efforts. During the Mercury project, multiple IBM 709 computer systems were used to assist in the data processing effort [31]. The IBM 709 was capable of up to 12 kiloflops or 12,000 floating point operations per second [25]. In comparison, the Intel i7 processor in my personal desktop system is capable of 40 gigaflops or 40,000,000,000 floating point operations per second [26].

Unfortunately, processor performance improvements have far outpaced performance improvements in I/O throughput and latency. Modern super-computing clusters have reached petaflops of processing power but rely upon traditional hard disk drives for I/O. In order to provide users and their applications with high-performance I/O, distributed file systems are employed. These distributed file systems run on storage clusters with 1000s of disks distributed across 100s of storage nodes. These storage nodes are interconnected with the processing nodes via dedicated high-performance network interconnects.

Many issues are involved in the design and construction of these high-performance storage systems. Individuals looking to deploy such a system must make design decisions based upon requirements for throughput, latency, redundancy, availability, capacity, scalability, number of processing clients, power, and cooling. The diagram in Figure 1.1 gives a high-level look at the different components that must be considered in the design of a high-performance storage system.

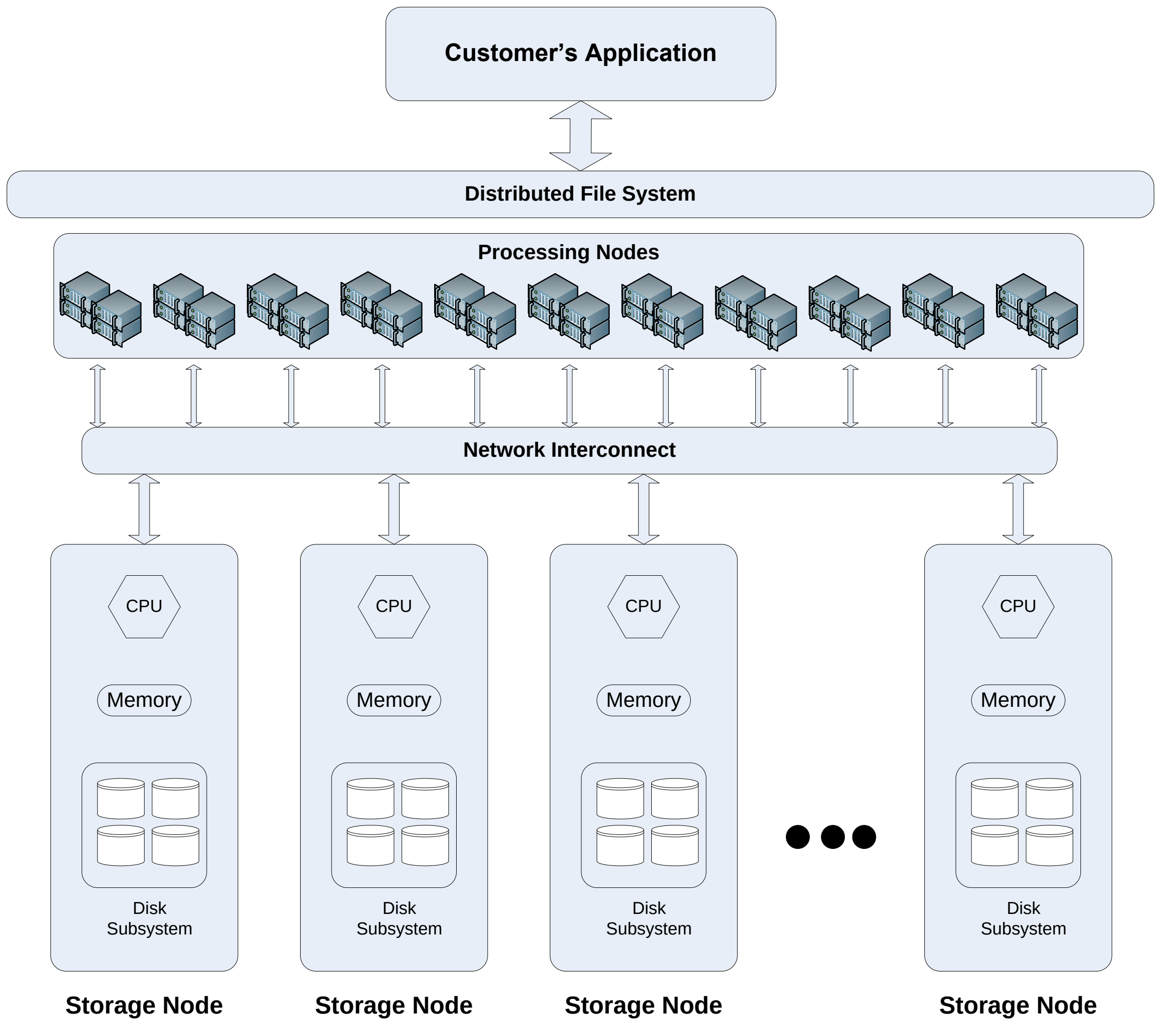


Figure 1.1: Anatomy of a high-performance storage system

1.2 Commercial storage solutions

There are a number of options to consider when looking to deploy a high-performance storage system. Will it be a home-grown system with custom-built hardware and open source software? Or will it be a commercial, turn-key solution with proprietary software? Two popular proprietary options are provided by Panasas and OneFS. The underlying questions of hardware selection, disk subsystem reliability, and distributed file system selection are addressed by engineers from the respective companies. There are also commercial, open source, options provided by Penguin Computing and Microway that allow for customized storage solutions but that are still essentially turn-key.

Regardless of who provides the storage solution, it is important to understand how it will be used to ensure that it is configured properly. These include issues of usable capacity, redundancy of data, throughput and latency, as well as how data will flow through the system and be archived. Additional criteria include whether an organization has adequate facilities with space, cooling, and power. There may also be policies or contract requirements for vendors to provide maintenance agreements with specific service levels such as having a technician on-site within four hours.

A vendor's sales engineer may be able to assist with answering these questions, but they are trying to sell *their* solution, not necessarily the best solution. Understanding the requirements of a storage system upfront can save a lot of frustration later on.

1.3 Problem statement

High-performance storage systems are complicated, requiring expert level knowledge to design and maintain them. Unfortunately, documentation on the key area of storage system design is incomplete and scattered across a number of sources. In addition, the knowledge that comes from the experience of working directly with these systems is localized within corporations and national laboratories and not generally available except in mailing lists and user forums.

This thesis addresses four areas in storage system design. Each of these areas was a pain point during the construction and maintenance of GeneSIS, a Beowulf style Linux cluster with 84TB of storage located in the HPC lab at Boise State University, requiring months of research and experimentation to understand and incorporate back into the design of GeneSIS. Each of the following questions addresses one of these areas.

1. What questions should be asked when determining storage system design requirements?
2. What techniques for designing disk subsystems best protect data against latent sector errors and infant disk mortality?
3. Which distributed file system will best meet the performance and scalability requirements of the storage system?
4. How can I determine the performance constraints and I/O characteristics of a given application?

The answers to these questions are not cut and dry and require a solid understanding of the underlying hardware and software components before educated design decisions can be made. This thesis documents the design considerations and potential pitfalls faced when deploying reliable, high-performance storage systems. This documentation includes critical design details that have been gleaned from research papers, user guides, mailing lists, SC2009 conference presentations, and the lessons learned from the design and maintenance of GeneSIS. The above questions are not specific to GeneSIS and are not entirely unique to high-performance storage system design. As a result, the information provided in this thesis will be valuable long after the current technology has been consigned to the scrap heap.

1.4 Thesis overview

There is a lot more to designing a storage system than simply purchasing a bunch of cheap, fast disks, putting them in servers, and installing some open source software. Chapter 2 discusses the questions to answer when designing a storage system. It is presented from the perspective of a storage consultant designing a storage system for a customer, but in reality the information presented applies to anyone considering the deployment of a high-performance storage system.

Storage systems are made up of hundreds or thousands of disks grouped together by RAID or some other mechanism into disk subsystems and these disk subsystems are the building blocks for a reliable, high-performance storage system. Chapter 3 takes a close look at how to design reliable disk subsystems in the presence of the well-published issues of latent sector errors and infant disk mortality.

The "high-performance" in a high-performance storage system comes from the ability to aggregate the performance and storage capabilities of multiple servers into a single unified file system. These types of file systems are a specialized type of distributed file system known as a parallel distributed file system. Chapter 4 provides a survey of several popular, general purpose, parallel distributed file systems, focusing on configuration options, performance, and scalability.

The client applications running on the storage system have as much influence upon whether the storage system is high performance as any other component in the storage system. Some applications are well suited to run on parallel distributed file systems, while others are not. On one end of the spectrum there are applications that process large data files in large (1MB or more) sequential operations. These applications are ideal for use with parallel distributed file systems. On the other end of the spectrum are applications that perform small (8KB or less) random I/O operations. In between is a world of grey where most user applications reside. Chapter 5 presents a technique for identifying application I/O characteristics and performance constraints.

CHAPTER 2

UNDERSTANDING STORAGE SYSTEM REQUIREMENTS

2.1 Overview

Before disk drives and RAID volumes, before interconnects and file systems, before thinking about tower vs rack cases, a storage engineer must carefully consider the system requirements when designing a new storage system. In the words of Sherlock Holmes, “It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.” [5] This quote, taken from “A Scandal in Bohemia,” applies remarkably well to storage engineering. Invest time in gathering the facts, then design a storage system to fit the facts. Remember, the storage engineer’s job is to help the customer solve a problem, not create a new one.

It is easy to design a poor-performing storage system. Even a storage system with average performance can be designed by someone with little or no storage experience and a credit card. Designing a high performance storage system requires knowledge of the various types of storage systems and the performance characteristics of each. It also requires quality components. But most important, it requires a thorough understanding of how the system will be used. Imagine that an engineer is consulted with to design and build a bridge across a river. He spends two years on the project, and when he is finished he has constructed a beautiful foot bridge, complete with solar-powered LED lighting system and dedicated bike lanes. When the customer returns to inspect the work, he is shocked. How is he supposed to join two six-lane freeways together with a simple foot bridge?

In the bridge example, the customer knows his needs: type of traffic, number of lanes, weight requirements, etc. These are physical. In the early design phases, the customer would see the plans that the engineer was drafting and realize, before construction began, that the foot bridge would not meet his needs. The requirements for storage systems, on the other hand, are more abstract, making it difficult for customers to know their needs. The customer typically understands the problem he is trying to solve but not what it will take to solve it. This is where the storage engineer must be a good listener and part psychic. Helping the customer probe these issues enables the storage engineer to design a storage system that will meet the customer's needs without excessive cost and complexity.

A storage configuration worksheet is provided in Appendix A to assist in the design of a high performance storage system. The customer may not be able to answer most of the questions directly; however, the storage engineer should be able to answer the questions after talking with the customer. Each topic covered in the worksheet is discussed in the following sections, including how the information requested on the worksheet directly affects storage design decisions.

2.2 Background

Why is the customer considering a high performance storage system? This is a good opportunity to learn about the particular problems the customer is attempting to solve. Chances are that there is an existing storage solution in place, either in a production or a development environment. What aspects of the existing solution are currently meeting the customer's needs? What are the actual and perceived limitations of the existing solution?

Managers, application developers, and system engineers can have drastically different concerns from a storage perspective. Managers are concerned with maintenance cost and return on investment. Managers like fixed, known costs and they care about the big picture. Application developers want to quickly store and retrieve data in the form of streams, objects, or flat files. Application developers like simple, configurable interfaces for I/O operations. Application developers resist changing code to improve performance, preferring to push for faster hardware. System engineers care about ease of management, scalability, performance, backups, data integrity, disaster recovery, and maintenance agreements. If the managers, application developers, and end-users are happy, then the system engineer is happy. $\ddot\smile$

2.3 Storage capacity and growth

One aspect to consider when designing a storage system is the amount of usable storage capacity the customer would like to have available. This amount does not directly translate to the number of disks required in the storage system because the storage engineer must take into account data redundancy configurations, both at the file system and block device level. An example is a customer who requires 20TB of usable capacity for his application. After considering the critical nature of the data, it is decided that 2x replication should be used at the file system level and RAID10 should be used at the disk level; the resulting raw capacity requirements are in fact 80TB.

Another aspect of storing data is how quickly the data will grow. How much storage capacity will be required over the next two to three years? This is a difficult question to answer, but it is important to consider as it affects many of the storage system design decisions. Planning for growth often increases the initial system cost but can significantly decrease the cost to scale the system, especially in installations where floor space / rack space comes at a premium.

For instance, a 48U rack can comfortably hold ten 4U storage nodes. If the storage system is configured with storage nodes that can hold eight 1TB hot-swap SATA disks, there is a raw storage capacity of approximately 80TB per rack. Increasing the capacity of the storage system would require a second storage rack and purchasing new storage nodes. If this equipment is housed at a co-location facility, the cost of a second rack will come at a premium. An alternative would be to configure the original system with 4U storage nodes that can support up to 16 hot-swap SATA disks, but use only five nodes instead of ten. In this configuration the raw capacity of the storage system can be doubled in the future without requiring a second rack. This example can be taken one step further. In the initial system configuration, 2TB disks could be purchased instead of the 1TB disks. Using 2TB disks would require only eight of the sixteen hot-swap slots in each storage node to reach 80TB of raw storage capacity. At the time I'm writing this paper, the cost of the upgraded case, RAID controller, and disks increases the cost of each storage node by 20 to 30% but allows for growth up to four times the raw capacity in the same footprint. This can be a bargain when it comes time to expand the capacity of the storage system.

Another benefit of designing a system for scalable growth is that it leverages the trend for decreasing hardware costs over time. An example of how this trend can be leveraged is by purchasing raw disk capacity to meet the customer's initial storage needs plus 20% extra for growth. Several months later, as the customer's storage needs increase and the price of disk storage has dropped, the storage capacity can be increased by purchasing additional (and possibly larger) storage disks. The idea for this approach is that the customer is not paying a premium for storage that is not needed yet. This strategy can be modified to account for the growth rate of the customer's data as well as the customer's policies for disk drive replacement.

An important item to consider when planning for growth is vendor support for hardware upgrades. Our research lab purchased an EMC AX150 in 2007, configured with twelve 500GB SATA disk drives. In 2010 we wanted to upgrade this unit with 1TB SATA disk drives, but EMC customer support stated that the unit would only support up to 750GB capacity disk drives. To top it off, only hard drives purchased directly from EMC would work in the unit, and those drives cost six times more than retail. This was a limitation enforced in the device firmware, and the solution recommended by EMC customer support was to purchase the latest model of chassis.

2.4 Storage client details

Both the number of storage clients and the client operating system will have a significant impact on the overall design of a high performance storage system. Linux clients provide the greatest amount of flexibility in the design of the storage system while Windows clients provide the least. The reason for this is that the majority of shared disk and parallel distributed file systems are developed specifically for Linux environments. Several of these file systems have native clients that work on MacOS and Unix, but not Windows. Connecting a Windows client requires the use of a gateway node. Gateway nodes can be used as a cost-effective method of providing clients access to the storage system, but they can easily become a performance bottleneck. For that reason, it is preferable for client systems to use native file system clients.

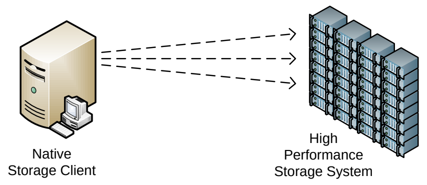


Figure 2.1: Native file system client communicating directly with storage nodes over a dedicated private interconnect such as Infiniband

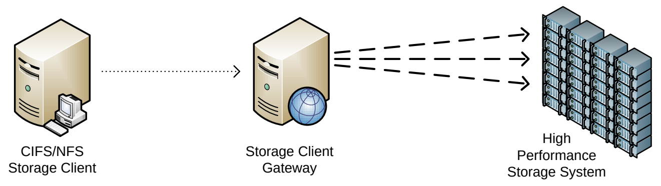


Figure 2.2: CIFS/NFS client communicating with a storage gateway over a workstation network such as gigabit Ethernet

The number of client machines helps to influence type and configuration of the storage interconnect as well as the number of storage nodes that should be present in the storage system. A large number of active clients can easily overwhelm a small number of storage nodes, while a small number of clients will not fully utilize a large number of storage nodes. Unfortunately, there is no “golden-ratio” specifying the ideal number of clients to the number of storage nodes.

As the number of clients increases, the load on the storage system will increase. Increasing the aggregate throughput of the storage system requires either an upgrade to the storage interconnect, the addition of more storage nodes, or both. Knowing that the number of clients is going to increase can mean using an Infiniband interconnect, rather than gigabit Ethernet, to increase the throughput each storage node is able to provide. The local disk subsystems on the storage nodes will also need to be configured to supply data at the increased throughput levels.

2.5 Data details

2.5.1 Data classification

A good source of information for helping with storage system design decisions is the actual data that will be stored on the system. Quite often, data is thought of as simply information stored on hard disks and retrieved by various applications. However, a good understanding of the data can reveal a lot about how the storage system should be designed.

For instance, large video files are processed sequentially, either as a stream or in chunks. Video files typically support concurrent client access, which can lead to a performance bottleneck. Distributing a video file across multiple nodes using striping can improve performance. Because the files are processed sequentially, they can benefit from read-ahead caches, which can help hide interconnect and file system latency.

On the flip side, applications that store data in a database format primarily perform non-sequential I/O operations that often do not benefit from large caches. The size of the database I/O operations are often in small block in the range 512B up to 128KB. [47] As a result, interconnect latency and file system overhead can severely limit the throughput performance.

Appendix B contains a general list of data classes and some of the characteristics of each. These classifications should not be used as firm, fixed rules, but rather as guidelines to help a storage engineer begin thinking about how the data can influence system design. In the end, it is the application that determines how the data is accessed, but looking at the type of data is a good place to start.

2.5.2 Storage zones

It is a rare storage system that stores a single type of data. The result is that there are mixtures of large and small files. Some data types are primarily read-only while others are read-write. In addition there are questions of data redundancy and backup, as well as performance requirements that may be different depending upon the type of data. Unfortunately, there is not a one-size-fits-all solution that will meet all of a customer's data storage and processing requirements.

To address these issues in data management, storage zones can be defined to group data based upon type, client access requirements, and data redundancy and backup policies. Storage zones can also have policies defined for data lifetime to prevent stale data from wasting space on the storage system. Multiple storage zones can be defined on a storage system. Storage zones are only guidelines for managing data on a storage system and are not enforced by the storage system.

This concept of storage zones describes how data moves into and out of the storage system. Figures 2.3 - 2.9 show an example of how data might flow in a storage system designed for video rendering. Table 2.1 defines the policies for each storage zone. Understanding how data moves through the storage system can help the storage engineer understand the throughput requirements of each client. From the example, the clients transferring media to the storage system do not require 10 gigabit Infiniband interconnects since the throughput will be limited by the source devices. The clients processing the digital photos in Figure 2.5 also do not require high levels of throughput. For these clients, accessing a gateway node using CIFS or NFS over gigabit Ethernet will be more than sufficient. The clients in Figures 2.6 and 2.7 will be doing work that is CPU intensive. However, if the application is multithreaded and the client systems have a lot of processing power, clients performing these operations could benefit from a high throughput interconnect such as Infiniband.

Zone Name	**Throughput**	**Data Distribution**	**Backups**
A	Med	Simple	Nightly Full
B	Med	Striped	Weekly Full with Nightly Incremental
C	High	Striped	None

Table 2.1: Storage zone policy definitions

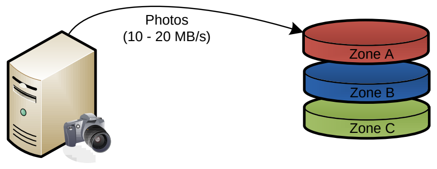


Figure 2.3: Digital pictures are downloaded from a camera to storage zone A via USB

When dealing with large volumes of data, especially when there are a variety of data types, have policies in place to ensure appropriate use of each storage zone. Some of these policies may be enforced at the system level, but in the end the storage system will depend upon its users to make appropriate decisions where data should be stored. This process requires good communication between the application developers and

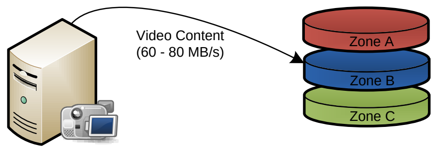


Figure 2.4: Video content is downloaded from a video camera to storage zone B via Firewire

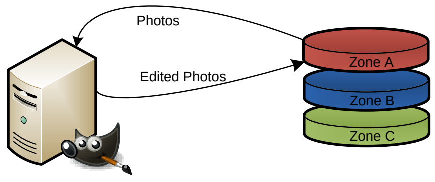


Figure 2.5: Digital pictures are touched up and stored back in storage zone A

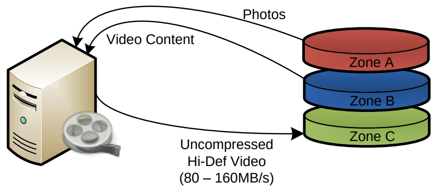


Figure 2.6: Movie is rendered from source material in zones A and B and written to zone C

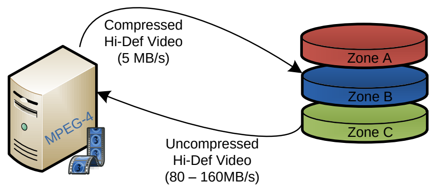


Figure 2.7: Hi-def version is compressed and written to zone B

the system engineers. Along these lines, it is important to know who will manage the storage zones, clean up stale data, perform backups, and monitor the storage system. Storage systems that are not managed effectively can quickly go from high

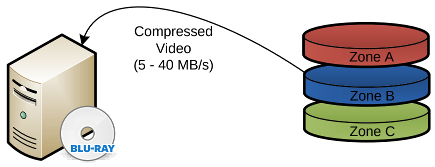


Figure 2.8: Hi-def version is written to Blu-ray disks

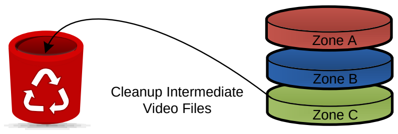


Figure 2.9: Intermediate movie files in zone C are removed from the storage system

performance systems to poor performing ones.

2.6 Applications details

The data can give part of the picture, but achieving high performance for customer applications requires a solid understanding of the flow of data and of how the applications interact with the storage system. To begin, the system engineer needs to have a list of applications that will interact with the storage system. This is where it is helpful to sit down with application developers, system engineers, and end-users. Discuss how they use the storage system, work out their process flows, and compose a list of applications. This is also a good chance to discuss performance issues.

For each application on the list, specify the data that the application accesses and whether those I/O operations are read-only or read-write. It is also beneficial to profile these applications while they are running to get an idea of the I/O characteristics. Chapter 5 provides an in-depth discussion of tools that are available to assist with this process. These profiling techniques can identify whether an application is I/O bound, memory bound, or CPU bound. They can also provide information on the current read and write throughput as well as the percent read vs percent write operations. This information is useful because it can help the storage engineer understand the throughput requirements of an application, but it can also help gauge expectations of application performance. If an application is CPU bound, moving the data to the fastest storage system in the world will not improve the performance of the application. [47]

2.7 Disaster recovery

Questions of uptime and high availability(HA) all relate to how much redundancy is built into the system. There are two different aspects to this topic. The first is data redundancy, focusing on replication and backups. The second is system availability, focusing on building levels of redundancy into the storage nodes and interconnect to ensure that the system can remain functional in the event of hardware failures.

In many cases, there is a trade-off between performance and redundancy. Most high performance parallel distributed file systems do not provide built-in functionality for HA or data replication; instead they rely on the underlying systems to implement this functionality. File systems that do provide replication typically sacrifice some write performance. Understanding the customer's need for performance vs redundancy is imperative when designing a storage system.

Is the data on the storage system critical to business operations? If so, connect the storage system to an Uninterruptible Power Supply (UPS) with sufficient run-time to allow the storage system to shut down cleanly or transition to backup power generators. Storage systems use several layers of caching to improve performance. To prevent loss of data, the write caches must be flushed to disk. Design the system so that data is not lost in the event that a single disk or even an entire storage node fails. Xin writes in Reliability Mechanisms for Very Large Storage Systems: “Based on our data, we believe that two-way mirroring should be sufficient for most large storage systems. For those that need very high reliability, we recommend either three-way mirroring or mirroring combined with RAID.” [56] A high level of reliability for business critical data can be achieved using a layered approach. First, configure the RAID subsystem in the storage nodes to ensure that a single (or multiple) disk failure will not result in data loss. Second, replicate data across multiple storage nodes, ensuring that no data is lost in the event of a complete node failure. And of course, perform regular backups of critical data to external disks or tape.

Is access to the storage system critical to business operation? If so, the system should employ file replication or shared block devices with HA fail-over. There should also be redundant storage interconnects and any gateway nodes should be configured for HA. Storage nodes can be configured with dual power supplies, redundant memory, and even an internal Fibre Channel loop with dual controllers. The key here is to balance the level and expense of redundancy against the risk of failure and the cost of downtime.

2.8 Facility

Knowing where the storage system will be installed helps for determining the density of the storage system. At a co-location facility, there is a monthly cost per storage rack as well as a fixed amount of power available in each rack. In co-location environments, maximize the amount of storage per rack while staying within the available power limits. A benefit of co-location facilities is that most provide site-wide UPS systems with automatic fail-over to backup generators in the event of power failure.

2.8.1 Power requirements

If the system is installed at the customer's site, ensure that the facilities have sufficient power and cooling. It would be unfortunate to design and build a beautiful four-rack storage system but only have a single 20amp circuit to power it. A rough estimate of the storage system power requirements can be obtained by examining the input voltage and amp requirements for each storage node. This can be found printed on a label on the back of the power supply or in the documentation included with the storage node. This number will be a max power level. To obtain a more "real-world" value, attach an amp meter to a storage node and run a series of tests to simulate peak load on CPU cores and disk drives. Assuming that all the storage nodes require the same input voltage, multiply both the max amps and the real-world amps by the number of storage nodes. The result is the max and real-world amperage requirements for the storage system at the required input voltage.

The power required for the storage nodes will dominate the overall power requirements of the storage system, but it is a good idea to check the power requirements of interconnect devices (switches, routers, etc.) as well as plan for growth of the storage system. These values for max and real-world amperage can be used to calculate VA and Watt values for UPS specification. Remember to plan for power for the cooling system as well.

```
VA = voltage * amperage
Watts = voltage * amperage * pf
```

Sizing a UPS system is not a trivial task. An important fact that many people overlook is that UPS systems have ratings for capacity in terms of Volt-Amps(VA) and Watts. Volt-Amps are used to measure the "apparent" power usages while Watts measure the actual power usage. [35]. Volt-Amp capacity measurements are used for marketing, but the nasty little secret in the UPS industry is that many UPS systems have a power factor(pf) as low as 0.66. [2] This means that a 1000VA UPS system will only be able to power a load of 660 watts. Unlike UPS manufactures, who often calculate wattage capacity assuming a power factor in the range of 0.66 to 0.90, most modern computer systems have a power factor approaching 1.0 (unity). [2] Many UPS manufactures provide capacity planning tools to match UPS systems to site-specific load and run-time requirements.

2.8.2 Cooling requirements

An estimate of the cooling requirements for the storage system can be calculated from the above power requirements. Due to the fact that essentially all of the power consumed by the storage system is converted to heat, the thermal output of the storage system is the same as the power input. [36] Heat generated by the storage system is equivalent to the max and real-world wattage values calculated above. These values can be converted to BTUs or Tons using the following formulas: [36])

```
BTU per Hour = 3.41 * Watts
Tons = 0.000283 * Watts
```

These values are estimates on the cooling requirements for the storage system itself. When specifying the cooling requirements for a particular environment, one must consider all the possible heat sources. These include IT Equipment, UPS with

Battery, Power Distribution, Lighting, and People. [36] In addition, care must be taken in planning for growth of the storage system. It is strongly recommended to consult with an HVAC engineer experienced with data-center cooling systems once the power requirements have been identified.

2.9 Budget

The customer's budget is the single most influential factor in the storage system design. Sections 2.1 through 2.8 deal with identifying what the customer needs from the storage system design. The budget determines what the customer can afford to buy. Ideally, the customer can afford what he or she needs, but to often this is not the case. In such an event, compromise becomes the order of the day. High-capacity, high-performance, high-reliability, and low cost lie in the four corners of the magic square of storage system design, unfortunately storage engineers can only choose up to three of these to include as priorities in the storage system design.

In addition to the initial funds required to purchase and configure a high performance storage system, there are costs for operation and maintenance. These include recurring costs for power and cooling, if the system is installed at the customer's location, or a facility charge if the system is housed at a co-location facility. If the components of the storage system were purchased from a vendor such as Dell or IBM, the storage nodes will most likely include a three-year maintenance agreement, covering the costs of any hardware failures. Storage systems that are custom built will still include warranty agreements on components but may require funds to ship defective parts for replacement.

When a component in a storage system fails, the time required to replace the failed component is referred to as the *window of vulnerability.* A large window of vulnerability increases the probability of data loss, so it is critical to have processes in place to quickly replace failed components. [56] To minimize the window of vulnerability, budget for spare components or purchase a maintenance agreement with four-hour or next-day on-site service.

Finally, budget time for an engineer to maintain the storage system. A storage system will require monitoring to detect potential issues as well as someone to replace components when they fail. Components will fail. “In petabyte-scale file systems, disk failures will be a daily (if not more frequently) occurrence.” [56] The amount of time to budget for an engineer will vary depending upon the size of the storage system.

2.10 Conclusion

Storage systems are diverse creatures with a multitude of design choices and configuration options. A thorough investigation of the storage system requirements will enable the design of a storage system that will meet the customer’s needs without excessive cost or complexity. Listen carefully to the problem the customer is trying to solve and ask lots of questions. In the design of the storage system, demonstrate how the storage system addresses each of the limitations of the existing storage solution as well as any specific requirements the customer has listed. Once both parties agree on the requirements and design details, it is time to begin selecting hardware.

CHAPTER 3

DESIGNING RELIABLE DISK SUBSYSTEMS IN THE PRESENCE OF LATENT SECTOR ERRORS AND INFANT DISK MORTALITY

3.1 The threat to disk subsystems

It is easy to assume that when a file is stored to disk it will be available and unchanged at any point in the future. However, this is not guaranteed. Imagine a world where disk manufacturers publish expected bit error rates of one in every 12TB read, where large numbers of disks fail in their first year of operation, and where data can be silently altered between the computer's memory and the hard disk platters. This world is in fact our reality. This chapter will examine the issues of infant disk mortality, latent sector errors, and silent data corruption, and provide recommendations for how to configure reliable disk subsystems to protect against these issues.

3.1.1 Infant disk mortality

Infant disk mortality is the tendency for hard disk drives to fail at a much higher rate in their first year of operation than the Mean Time Before Failure (MTBF) rates specified by the manufacturer suggest. Several studies using data from large HPC deployments indicate that disk drives are replaced by a factor of 2 - 10 times the rate suggested by the MTBF rating [37, 22, 57]. That fact alone is concerning, but these studies have also shown the shape of the drive failure curve to be bathtub shaped with the bulk of the failures coming in the first year of operation or at the end of the life of the drive (typically 5 years) [57].

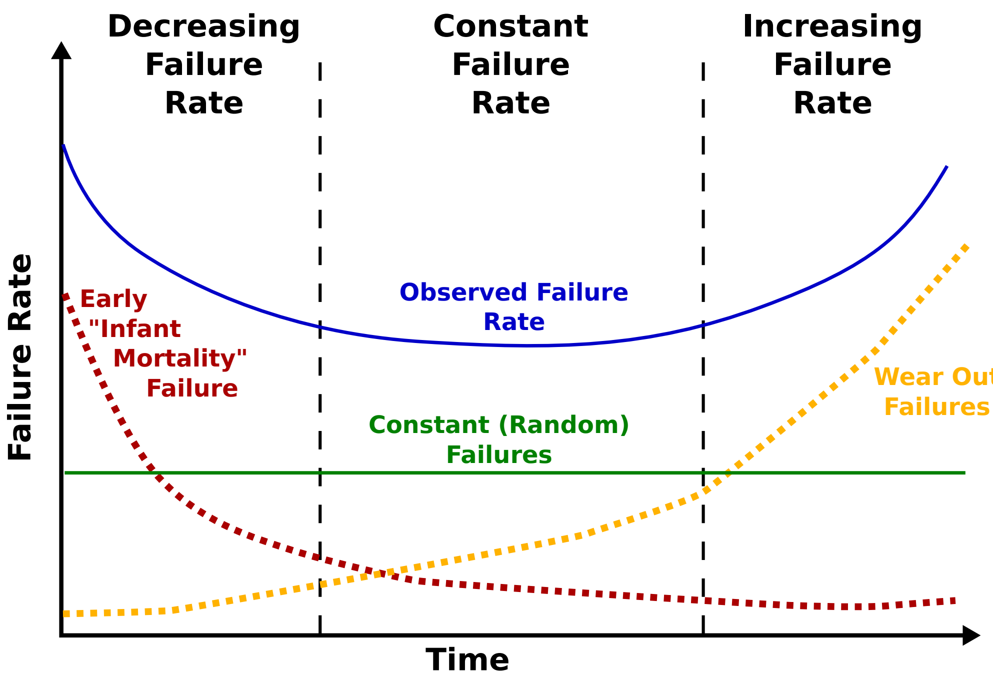


Figure 3.1: Bathtub curve representing disk failure rates [55]

Figure 3.1 provides a graphical explanation of this failure curve. The curves do not reflect any specific disk failure data, but instead show generalized failure trends described in multiple studies [57]. The Infant Mortality curve represents disks that fail early in their life while the Wear Out curve represents disks that fail toward the end-of-life. The constant failure curve represents the expected failure rate if disk failures were random and evenly distributed across the expected life of the drive. The Observed Failure curve depicts the bathtub shaped failure curve discussed previously.

Table 3.1: Probability of disk failure based upon SMART data [34]

SMART Counter	Probability of failure within 60 days	Description
Scan Errors:	39 times more likely to fail	Sometimes referred to as seek errors, these errors occur when the drive heads are not properly aligned with the track.
Reallocation Count:	14 times more likely to fail	The number of sectors that have failed and been remapped.
Offline Reallocation Count:	21 times more likely to fail	The number of failed sectors that were detected and remapped using background disk scrubbing.
Probational Count:	16 times more likely to fail	The number of sectors that experienced read errors and that rescheduled to be remapped upon the next write operation unless a successful read of the sector occurs before the remap.

Modern disk drives provide extensive monitoring capabilities through a standardized interface called SMART (Self-Monitoring, Analysis, and Reporting Technology). Several attempts have been made to accurately predict when a disk drive is about to fail by using this SMART data. A study examining a large collection of disk drive failure and usage information gathered by Google attempted to ascertain whether SMART counters can be used to predict drive failure. This work showed that drives with Scan Errors, Reallocation Counts, Offline Reallocations, and Probational (Pending) counts had a significantly higher probability of failure than drives with zero counts. One of the conclusions from this study is that SMART data cannot be used as the only indication of pending drive failure as 56% of the drive failures in their sample set show zero counts for the above SMART counters. These results showed a high infant mortality rate in the 3 month and 6 month time frame; however, these values would have been significantly higher if not for the initial system burn-in testing that disks go through before being put into production. Table 3.1 shows some interesting statistics from this study. Other items of interest are that drive activity and temperature do not have a significant impact on drive failures [34].

3.1.2 Latent sector errors

A latent sector error is a generic term that is used when a disk drive is unable to successfully read a disk sector. Latent sector errors can show themselves as Sector Errors, Read Errors, Not-Ready-Condition Errors, or Recovered Errors. They can be caused by a variety of factors including media imperfections, loose particles causing media scratches, "high-fly" writes leading to incorrect bit patterns on the media, rotational vibration, and off-track reads or writes [3]. The term bit error rate (BER) refers to the frequency that unrecoverable/uncorrectable read errors (URE) are expected to occur. Manufacturers publish expected bit error rates based upon disk drive class(see section 3.2.1 for definition of desktop, nearline and enterprise disk classes). These errors are considered part of normal disk operation as long as the errors are within the rate provided in the disk specification. The dirty little secret about latent sector errors is that they are only detected when an attempt is made to read the sector. This means that a disk may contain corrupted data without the user knowing it.

Schwarz observed that latent sector error rates are five times higher than disk failure rates [38]. As a result, latent sector errors can wreak havoc on RAID arrays. For example, imagine a 2TB array with three 1TB disks in a RAID-5 configuration. Now imagine that one of the disks fails, leaving the array operational, but in a degraded condition. A new disk is added and the rebuild process begins, regenerating the RAID striping on the new disk from the remaining two disks. Three quarters of the way through the rebuild process, one of the disks from the original array encounters an unrecoverable read error. At this point the RAID set is lost and the data can only be retrieved using time consuming and expensive data recovery techniques.

Microsoft Research conducted a study focused on the bit error rates advertised by disk manufacturers. They performed a series of tests where they would generate a 10GB file and calculate the checksum. Then they would read the file and compare the checksum of the file to the original checksum to test for read errors. The results were written to disk, then the test was repeated. This was run for several months with a total of 1.3PB of data transferred. Another round of tests was performed using 100GB test files and continually reading the file to test for bit-rot. These tests moved more than 2PB of data and read 1.4PB. They observed a total of four definite uncorrectable bit errors and one possible uncorrectable bit error across all of their tests. However, in their testing they saw far more failures in drive controllers and operating system bugs than in read errors. Their conclusion is that bit error rate is not a dominant source of system failure [22]. However, their testing was conducted across four test systems with a combined total of only seventeen hard disk drives. This is a statistically insignificant number of disks. Other studies by Bairavasundaram and Paris demonstrate that bit error rates and latent sector errors can have a significant impact on storage system reliability [4, 3, 32].

This study of data corruption used statistics captured from 1.53 million disk drives over a period of 41 months found 400,000 instances of checksum mismatches on the disk drives [4]. A checksum error occurs when the bits stored in a disk sector are read but the calculated checksum value does not match the checksum value stored on the disk when that sector was written. An interesting finding from this study is that nearline class disks develop checksum errors at a rate that is an order of magnitude higher than enterprise class disks (see section 3.2.1 for definition of desktop, nearline and enterprise disk classes). This study also provides a section on "Lessons Learned" including recommendations for aggressive disk scrubbing, using staggered stripes for RAID volumes, and replacing enterprise class disks at the first sign of corruption [4].

In the literature, several ideas have been put forward as techniques to help address the issues of latent sector errors. These include a variety of intra-disk parity schemes [13], using staggered striping for RAID volumes [4], and a variety of disk, file/object, and RAID scrubbing techniques [3, 4, 32, 38]. Unfortunately, many of these ideas are not generally available for use in production environments. However, Mean Time To Data Loss (MTTDL) models that account for latent sector errors, RAID scrubbing has been shown to increase reliability by as much as 900% [32].

3.1.3 Silent data corruption

Silent data corruption can occur in processor caches, main memory, the RAID controller, drive cables, in the drive as data is being written, or in the drive as the data is being read. Desktop and workstation class systems with standard DDR3 memory and SATA disk drives are far more susceptible to silent data corruption than enterprise class systems (enterprise class servers have error correcting memory, high end RAID controllers with built-in error correcting procedures, SCSI, SAS, and FC protocols that natively support error correction, and enterprise class disk drives with an extra eight bytes per sector to use for storing checksum data directly on the disk.)

Figure 3.2 shows the layers that data must pass through when stored to or retrieved from disk. The upper layers exist in the application and operating system space, while the lower levels represent the physical storage hardware. Data corruption can occur at any of these layers. Even with enterprise class hardware, errors introduced at a high level in the storage stack will be silently stored to disk.

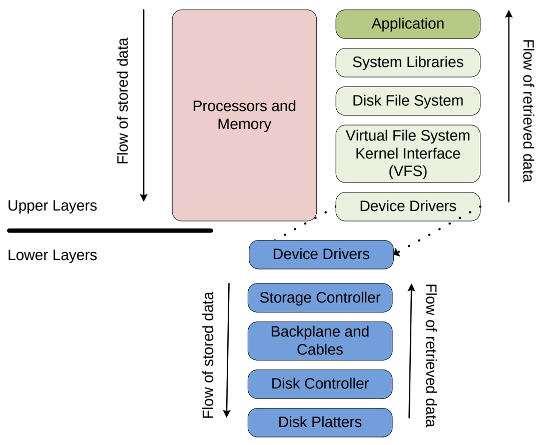


Figure 3.2: Diagram of the various layers in the storage stack

To address the issue of silent data corruption, Oracle and Emulex have joined forces to provide end-to-end (application to disk) data integrity [11]. The idea is that an object is created at the application level using a standardized object storage format. In addition to the application data, the object also contains error correction data that can be used to verify the integrity of the object. As the object is passed from the application through the various storage layers, the storage hardware is able to verify that the data remains correct. This object will be written directly to the storage system as an object. Oracle is contributing their block I/O data integrity infrastructure code to the Linux kernel [11].

3.2 Disk considerations

Disks drives are the building blocks of a disk subsystem. Understanding the characteristics of the various types of rotational storage media will go a long way for designing a reliable disk subsystem.

3.2.1 Classes of disks

There are a wide variety of disk drives available on the market with an equally wide variety of performance, capacity, and error correction features. These disks have been loosely categorized into classes based upon a particular feature set. Originally there were two basic classes: desktop and enterprise. Desktop drives used the ATA interface protocol while enterprise class disks used the SCSI protocol. In recent years the distinction between desktop and enterprise class disks has blurred. The development of aggressive power management and data recovery features as well as the fact that disk drive classifications are not consistent across manufacturers makes choosing the appropriate disks for a storage system a challenge.

To maintain consistency in this thesis, the following disk classifications are used: *desktop*, *nearline*, and *enterprise*. *Desktop* class disks are intended for home computer or office workstation systems that are not run continuously and have a low duty cycle. *Nearline* class disks are designed for use in data center RAID environments where there may be large amounts of data that must be available 24x7, but in actuality are not accessed very often. *Enterprise* class disks are designed for use in mission critical data center systems where they must be available 24x7, are accessed continuously, and must sustain high throughput levels and low latency with a high level of reliability. These definitions were chosen because they are consistent with the usage of the nearline and enterprise disk classifications used in the papers and articles cited in this thesis. Table 3.2 is derived from several white papers published by Seagate to show the differences between the different disk classes [43, 42, 40, 41, 39].

Table 3.2: Comparison of desktop, nearline, and enterprise disk drive classes

	Desktop	*Nearline*	*Enterprise*
Capacity	up to 2TB	500GB-2TB	300GB-600GB
Cost	low	med	high
Cache	32MB; 64MB	16MB; 32MB; 64MB	16MB
Performance	7200 RPM	7200 RPM	10K RPM and 15K RPM
Interface	SATA 6Gb/s	6Gb/s SAS; SATA 3Gb/s	6 Gb/s SAS; 4Gb/s FC
Mean Time Between Failure (MTBF)	750,000 hours	1,200,000 hours	1,600,000 hours
Annualized Failure Rate (AFR)	0.34%	0.73%	0.55%
Bit Error Rate (BER)	1 in 10^{14}	1 in 10^{15}	1 in 10^{16}
Duty Cycle	8x5	24x7	24x7
Power On Hours	2400	8760	8760

Desktop class

Desktop class drives have a great price-to-capacity ratio; however, they do not have many of the reliability features found in the nearline and enterprise class equipment. There are also a couple features of desktop drives that make them undesirable to use in a RAID environment. The first is power management. Oftentimes desktop class drives have power conservation features that allow the drive to spin down or go to sleep when not in use. For a laptop or desktop workstation this is great; however, if the drive is part of a RAID array, in the best case the RAID array will be slow responding while waiting for the disk to speed up. In the worst case, the RAID controller will assume the drive has failed and drop it from the array. Depending upon the number of drives and the type of RAID subsystem, it is possible, even likely, that multiple drives will enter power-save mode and be dropped from the RAID set. The RAID array will then be degraded and must be recovered, possibly resulting in data loss.

The second feature of desktop drives that makes them unsuitable for RAID environments is that they have some extremely powerful sector recovery features built into the on disk controller. At first glance this might not seem like a bad thing, but this deep recovery cycle can be time consuming [27].

“When an error is found on a desktop edition hard drive, the drive will enter into a deep recovery cycle to attempt to repair the error, recover the data from the problematic area, and then reallocate a dedicated area to replace the problematic area. This process can take up to two minutes depending on the severity of the issue. Most RAID controllers allow a very short amount of time for a hard drive to recover from an error. If a hard drive takes too long to complete this process, the drive will be dropped from the RAID array. Most RAID controllers allow from seven to fifteen seconds for error recovery before dropping a hard drive from an array. Western Digital does not recommend installing desktop edition hard drives in an enterprise environment (on a RAID controller).” –Western digital FAQ [46]

Nearline and enterprise class drives implement a feature which limits the amount of time spent attempting to recover a bad sector. Once this time elapses, a signal is sent to the RAID controller notifying it of the issue to allow it to obtain the sector from a different disk. Different disk manufacturers have different names for this feature, but in the end it all boils down to the same thing. Time-Limited Error Recovery (Western Digital), Error Recovery Control (Seagate), Command completion Time Limit (Samsung, Hitachi).

Nearline class

There is not a consistent name for this class of hard drives across all manufacturers. A few examples of drives that fall into the nearline class include business class disks, low-cost server disks, enterprise class SATA, and nearline SAS. The performance and reliability features also vary widely between manufacturers and disk models. In some cases, the only difference between a manufacturer's desktop and nearline class disk drives is the firmware on the drive controller.

In several of the papers cited in this thesis, the nearline disks have a bit error rate of 1 in 10^{14}; however, in Table 3.2 nearline disks are shown with a bit error rate of 1 in 10^{15}. This discrepancy is due to the fact that the data in Table 3.2 is from 2011 and the disk drives in the cited studies are considerably older. In addition, the data in Table 3.2 is provided by Seagate; other disk manufacturers may have a higher bit error rate for their nearline class disk drives.

Nearline class disk drives are designed to meet the need of low cost, high capacity storage for use in the data center. They are designed to be powered on and available 24x7, but only accessed infrequently with a duty cycle of 20-30%. This class of disk is designed for storing large quantities of reference data that must remain online, but that is not continuously accessed. Nearline class drives are not well suited to database-style workloads requiring a continuous duty cycle and a high number of I/O operations per second (IOPs), due to both the mechanical design of the disks as well as the limited processing capabilities of the onboard controller.

Nearline class disk drives are designed for use in RAID applications and are extremely well suited for large parallel distributed storage systems used in HPC environments. These environments often deal with 10s or 100s of TBs of data that require high levels of throughput, but not necessarily high numbers of IOPs, and the $/GB price point of nearline class disk drives is very attractive.

Enterprise class

There are a number of key differences between desktop/nearline class disk drives and enterprise class disk drives. Enterprise class hard drives have a more rugged construction than desktop or nearline class drives that allows them to operate reliability in 24x7 data center environments with a continuous duty cycle. Desktop and nearline class disks have a fixed sector size of 512 bytes while enterprise class disks support variable sector sizes with the default being 520 to 528 bytes. These extra eight to sixteen bytes are leveraged for end-to-end data integrity to detect silent data corruption [27]. They also include specialized circuitry that detects rotational vibration caused by system fans and other disk drives and compensates by adjusting the head position on-the-fly to prevent misaligned reads and writes [27].

Enterprise class disks have dual processors and advanced error detection and error correction capabilities built into the disk drives. The extra processing capabilities of enterprise class disk drives enable them to implement advanced techniques for ensuring data integrity. One of these techniques is *disk scrubbing*. During times of low disk activity, the disk controller can issue commands to the disk drive to verify the integrity of the disk sectors using the extra eight to sixteen bytes of data stored along with each sector [27]. Data scrubbing at the disk or RAID level has been shown in multiple studies to have a dramatic impact on the reliability of a disk subsystem [38, 4, 32]. Section 3.4.3 discusses data scrubbing in greater detail with an example of usage in a production environment.

In addition, disk manufacturers implement a number of proprietary techniques to further increase the reliability of enterprise class disk drives. These efforts allow enterprise class disk drives to operate at twice the RPM of desktop and nearline class drives but still maintain a bit error rate that is two orders of magnitude lower than desktop class disks. The result is a trade-off of price and capacity for performance and reliability.

3.3 RAID considerations

RAID is a powerful tool that can be leveraged to improve both the reliability and the performance of a disk subsystem. Xin demonstrates that using the MTBF rates published by disk manufacturers, a 2PB storage system composed of 500GB nearline disks can expect to have one disk failure each day [56]. Add to this fact that many real-world studies conclude that actual disk failure rates are up to ten times higher than the manufacturer's rates [32, 37, 22] and the need for RAID becomes apparent. Table 3.3 is a summary of the commonly used RAID levels in production environments.

Though the implementation of each RAID level is different, the underlying protection mechanisms boil down to two things: replication and parity. RAID subsystems utilizing replication-based protection mechanisms have significantly lower failure rates than ones that leverage parity-based protection mechanisms. This can be seen in Figure 3.4 where the failure rates for RAID-1 and RAID-10 are significantly lower than the failure rates for RAID-5 and RAID-6.

Another way to think about this is with an analogy of soldiers protecting maidens. In a RAID-1 scheme, each soldier is charged with protecting a single maiden with his life. In a RAID-10 scheme, there are M pairs of soldiers and maidens with each soldier charged with protecting a single maiden (similar to RAID-1). In this scheme, any or all of the soldiers can die as long as the maidens remain unharmed; however, if a single maiden dies, it does not matter how many soldiers remain, the battle is lost. With RAID-5, a single soldier is charged with protecting N maidens with his life. And finally a RAID-6 scheme charges two soldiers with the responsibility of protecting N maidens. It is easy to understand that if a maiden has her own personal bodyguard, she is a lot safer when trouble comes knocking than the maidens who share protectors.

Table 3.3: Description of commonly used RAID levels

RAID Level	Min Disks	Protection Level	Description
Level-0	2	none	Data is striped in chunks across disks
Level-1	2	single failure	Data is mirrored between the two disks
Level-5	3	single failure	Data is striped across n-1 disks with the nth disk containing an XOR parity. The parity stripe is offset so that the parity information is not stored exclusively on a single disk.
Level-6	4	double failure	Similar to RAID-5 but with dual parity stripes
Level-10	4	single/double failure	Data is striped across n/2 mirrored disk pairs. Up to n/2 disks can fail as long as no two disks are from the same mirrored pair.

3.3.1 Encountering latent sector errors

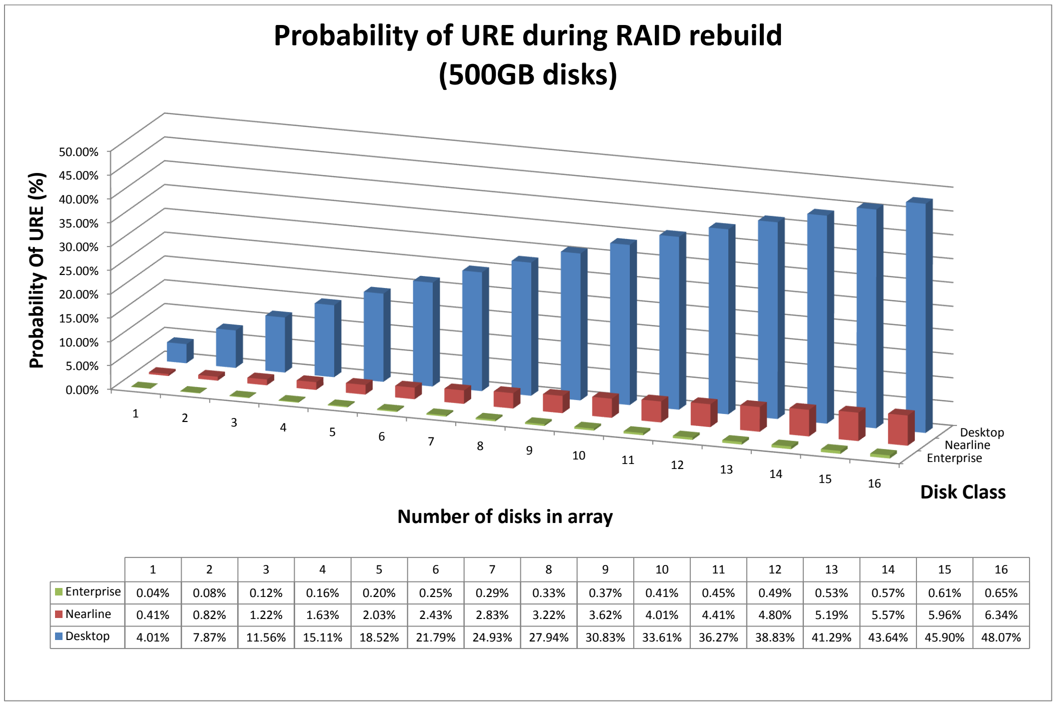
Care should be taken in the selection of the RAID level used as well as the number of disks in each RAID set. Paris conducted a study that looked specifically at RAID-1, RAID-5, and RAID-6 arrays in the presence of latent sector errors. The results indicate that unrecoverable read errors can reduce the mean time to data loss (MTTDL) of each of the three array types by up to 99% [32].

Figure 3.3 shows the probability of encountering an unrecoverable read error (latent sector error) while rebuilding an array of n+1 disks. Probabilities are shown for both 500GB (Figure 3.3a) and 2TB (Figure 3.3b) disk drives. These probabilities are calculated directly from the manufacturer's published bit error rates for desktop, nearline, and enterprise class disk drives using the probability equations published by Adaptec Storage Advisors [1]. Observe that the probability of encountering an unrecoverable read error (URE) increases as the size of the storage array increases, by increasing either the capacity of the disks or the number of disks. Also observe that the probability of an URE decreases by an order of magnitude with each increase in disk class (Desktop $->$ Nearline $->$ Enterprise). The adage, you get what you pay for has never been more true than in storage system design.

To bring this point home, imagine an array that is composed of eight 2TB desktop class disk drives configured for RAID-5. When a disk fails in the array, every sector from the seven remaining disks must be read successfully to rebuild the RAID set. From Figure 3.3b, there is a 68% chance that a latent sector error will cause the rebuild to fail and result in the loss of all data on the array. Even using nearline class disks there is still a 10% chance that a latent sector error will be encountered. For this reason, many storage experts feel that RAID-5 can no longer provide a sufficient level of reliability for modern disk subsystems.

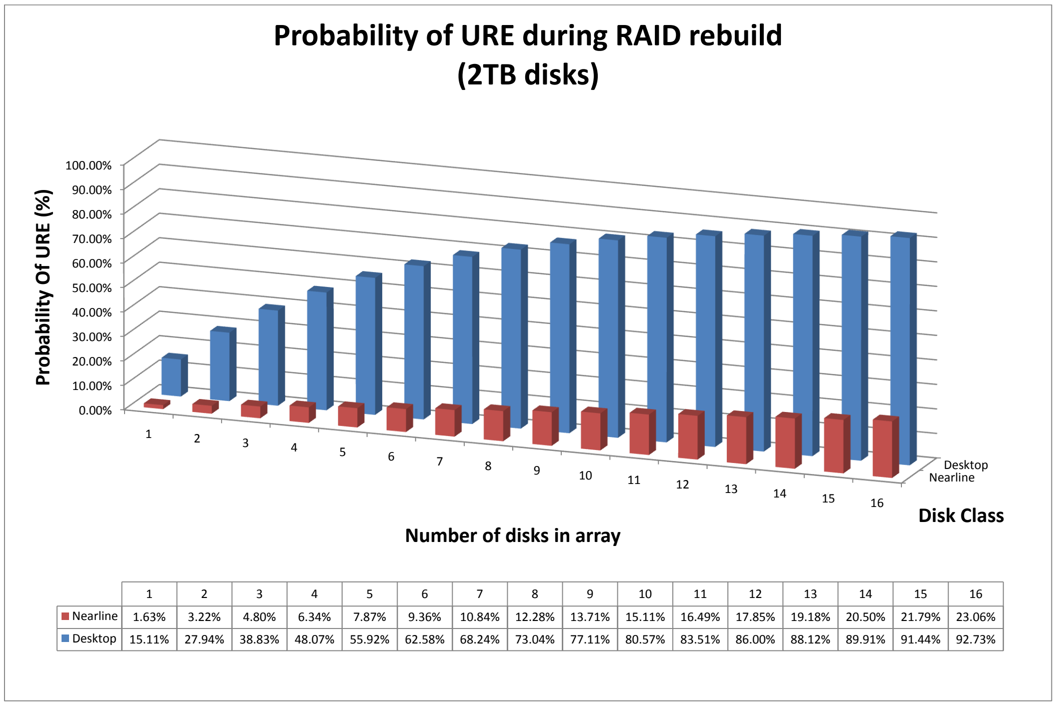
3.3.2 Utilizing mean time to data loss (MTTDL)

Figure 3.3 only reveals part of the picture. To adequately compare the relative merits of each of the different RAID levels requires a measure called mean time to data loss



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
🟩 Enterprise	0.04%	0.08%	0.12%	0.16%	0.20%	0.25%	0.29%	0.33%	0.37%	0.41%	0.45%	0.49%	0.53%	0.57%	0.61%	0.65%
🟥 Nearline	0.41%	0.82%	1.22%	1.63%	2.03%	2.43%	2.83%	3.22%	3.62%	4.01%	4.41%	4.80%	5.19%	5.57%	5.96%	6.34%
🟦 Desktop	4.01%	7.87%	11.56%	15.11%	18.52%	21.79%	24.93%	27.94%	30.83%	33.61%	36.27%	38.83%	41.29%	43.64%	45.90%	48.07%

(a) 500GB Disks



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
🟥 Nearline	1.63%	3.22%	4.80%	6.34%	7.87%	9.36%	10.84%	12.28%	13.71%	15.11%	16.49%	17.85%	19.18%	20.50%	21.79%	23.06%
🟦 Desktop	15.11%	27.94%	38.83%	48.07%	55.92%	62.58%	68.24%	73.04%	77.11%	80.57%	83.51%	86.00%	88.12%	89.91%	91.44%	92.73%

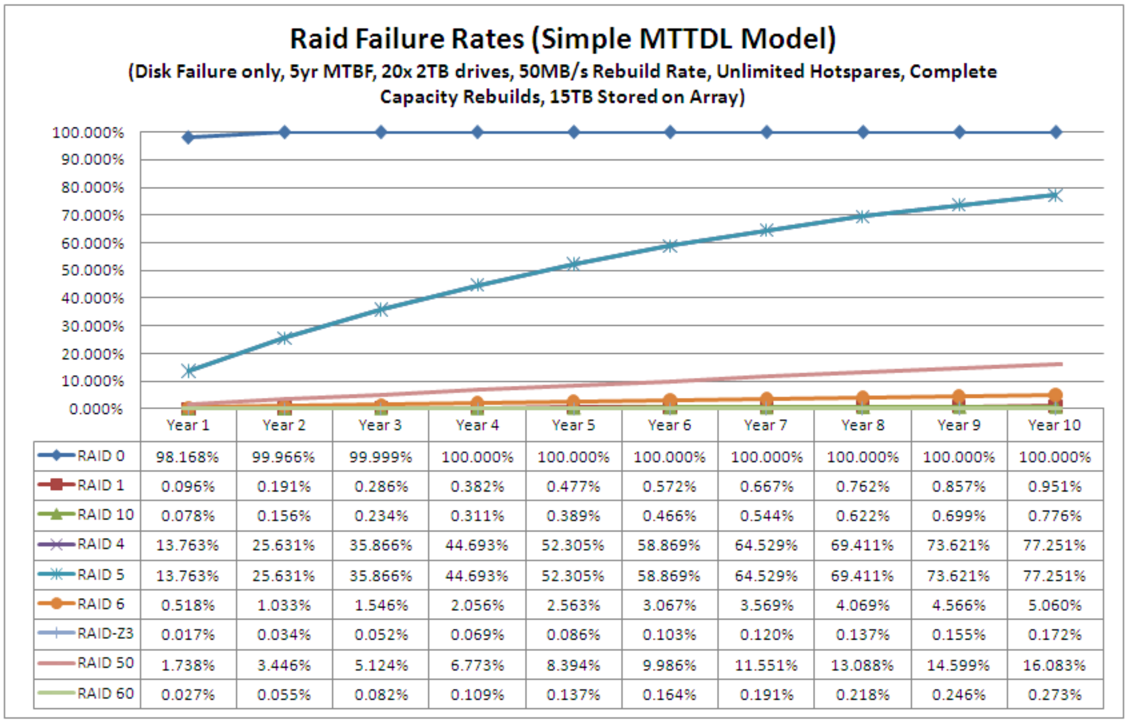
(b) 2TB Disks

Figure 3.3: Probability of encountering an Unrecoverable Read Error while rebuilding an array of n+1 disk drives

(MTTDL). MTTDL is a measure used to evaluate the reliability of a disk subsystem. Unfortunately, the values returned by most MTTDL models are next to useless. What value is it if a disk array has a MTTDL of 100,000,000 years versus 1,000,000 years? The trouble with most MTTDL models is that they do not take into account latent sector errors and silent data corruption. In addition, it is assumed that failures are evenly distributed and independent, not accounting for infant disk mortality, manufacturing issues, or firmware glitches. Many studies have examined the issues with MTTDL and proposed a variety of solutions including the use of Markov chains, Poisson distribution, and even Monte Carlo based approaches [16, 3, 13, 57].

In spite of this debate over the accuracy of MTTDL, it can still be utilized as a measure for evaluating the relative reliability of different RAID protection schemes. Figure 3.4 shows the failure rates for a variety of RAID levels. These values are calculated based upon a simple MTTDL model that utilizes the MTBF values published by disk manufactures. As a result, the values themselves are nearly worthless since they do not account for latent sector errors and infant disk mortality. However there is some value if we look at them from the perspective that they are best case values. What we see is that replication-based protection schemes have significantly lower failure rates than parity-based schemes.

What Figure 3.4 does not show is that parity-based schemes are far more susceptible to latent sector errors than replication-based schemes [33]. This is the result of the combination of a limited number of tolerated failures with the probability of encountering corrupt data during a rebuild. In a replication-based protection scheme such as RAID-1 or RAID-10, the maximum number of disks that will need to be successfully read for a single disk failure is one. From Figure 3.3b the probability of encountering a latent sector error is 1.63%. Not only is the risk of encountering a latent sector lower, but the time the array remains in a degraded state is lower since less data must be read to rebuild the array. With a parity-based protection scheme such as RAID-5 or RAID-6 with n disks, n-1 disks must be successfully read to regenerate the array. This results in both a higher probability of encountering a latent sector error and a longer period of time that the array remains in a vulnerable state.



	Year 1	Year 2	Year 3	Year 4	Year 5	Year 6	Year 7	Year 8	Year 9	Year 10
RAID 0	98.168%	99.966%	99.999%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
RAID 1	0.096%	0.191%	0.286%	0.382%	0.477%	0.572%	0.667%	0.762%	0.857%	0.951%
RAID 10	0.078%	0.156%	0.234%	0.311%	0.389%	0.466%	0.544%	0.622%	0.699%	0.776%
RAID 4	13.763%	25.631%	35.866%	44.693%	52.305%	58.869%	64.529%	69.411%	73.621%	77.251%
RAID 5	13.763%	25.631%	35.866%	44.693%	52.305%	58.869%	64.529%	69.411%	73.621%	77.251%
RAID 6	0.518%	1.033%	1.546%	2.056%	2.563%	3.067%	3.569%	4.069%	4.566%	5.060%
RAID-Z3	0.017%	0.034%	0.052%	0.069%	0.086%	0.103%	0.120%	0.137%	0.155%	0.172%
RAID 50	1.738%	3.446%	5.124%	6.773%	8.394%	9.986%	11.551%	13.088%	14.599%	16.083%
RAID 60	0.027%	0.055%	0.082%	0.109%	0.137%	0.164%	0.191%	0.218%	0.246%	0.273%

Figure 3.4: Failure rates for a variety of RAID levels [33]

3.4 Designing a reliable disk subsystem

The following sections describe techniques that can be leveraged to construct reliable disk subsystems.

3.4.1 Disk burn-in

From the Google study on infant disk mortality, it was suggested that the rate of infant disk mortality they were seeing in their systems would be significantly higher if not for their initial burn-in of disks before placing them in the production environment [34]. Taking a cue from Google's playbook, the following guidelines can be used to help limit the effects of infant disk mortality on a storage system.

The idea with a disk burn-in is that rather than discovering latent sector errors or infant disk mortality of a new hard disk in a production server, use a tool to read and write a series of patterns across the entire surface of the disk to identify issues up front. The nice feature of a test like this is that if a latent sector occurs, the controller on the disk will remap the sector the next time it is written to. After the test completes, the SMART data will show the number of sectors that have been remapped. The following process demonstrates a basic sanity test (burn-in) for a new hard drive.

- Verify that the SMART counters for pending and remapped sectors is zero
- Use the `badblocks` utility to write test patterns over the surface of the disks
- Verify that the SMART counters for pending and remapped sectors are still zero

The `badblocks` utility can be configured to perform multiple passes across the disk surface until there are no bad blocks detected or until the max number of passes is completed. In the event that bad blocks (latent sector errors) are found on the disk surface there are a couple of different schools of thought. One is that the disk is bad and should be returned to the manufacturer. This is supported by the disk failure statistics shown in Table 3.1. Unfortunately, what a user considers a bad disk does not always match up with what the manufacturer considers a bad disk. The definition of a drive failure is not well defined [34]. Schroeder cites a disk manufacturer that reported that 43% of returned drives passed the its in-house quality testing [37].

The other school of thought suggests that a nearline disk drive can still continue to function properly even with a few remapped sectors as long as the counts do not continue to increase; however, enterprise class drives should be replaced at the first sign of latent sectors. This school of thought is supported by statistics gathered by Bairavasundaram in a study of 1.53 million nearline and enterprise class disk drives [4].

3.4.2 Leveraging RAID

There are many different ways to configure RAID in a disk subsystem. Issues to consider are reliability, performance, and usable capacity. Other issues to consider are rebuild time, number of disks per array, and stripe size. There is not one correct solution to a given storage problem, but one often overlooked aspect is complexity to administer. Designing an overly complicated solution is an easy trap to fall into given the wide variety of options available. Another factor that MTTDL models cannot address is human error, and overly complicated disk subsystems can increase the chances that a mistake will be made.

An internal white paper published by Oracle introduced a new methodology called SAME [30]. SAME stands for Stripe and Mirror Everything. SAME has four basic rules:

1. Stripe all files across all disks using a one megabyte stripe width

2. Mirror data for high availability

3. Place frequently accessed data on the outside half of the disk drives

4. Subset data by partition, not by disk

SAME is a simple and elegant approach that applies not only to designing reliable disk subsystems but also to designing high performance storage systems as a whole. At the block level, SAME indicates the use of RAID-1 or RAID-10. This approach is substantiated by the drastically reduced failure rates for replication-based protection schemes shown in Figure 3.4. However, this approach can also be extended to the storage system level where data is striped across multiple storage nodes, each of which contain disk subsystems protected using a replication-based scheme.

Placing frequently accessed data on the outside half of the disk platters will maximize the throughput a disk can sustain. This is simply a side effect of platter-based rotational media. Hard disk drives implement a technique called zoned bit recording. The tracks on a disk are grouped into zones and each zone has an increasing number of bits per track moving from the center of the disk out. Since the platters in a disk drive operate at a constant velocity, more bits pass under the heads per revolution at the outer tracks than at the inner tracks. This results in a higher bit rate for the outer tracks and hence a higher throughput.

Utilizing partitions to localize data to certain areas of a disk can dramatically decrease seek time and improve the performance of random disk access. This is critical for databases, metadata servers, and applications that do not perform sequential I/O operations (this is intuitive if one takes the time to think about it.) The seek time for a disk drive is a measure of the time required to move the drive head to the appropriate track, then wait for up to one full revolution of the disk to start reading data. During normal disk operation, any single I/O operation may have to move the disk heads all the way from the inner tracks to the outer tracks. Partitioning off a smaller portion of the disk from which to perform I/O operations has the effect of localizing the data on the disk and will decrease seek time.

The goal of this configuration is to make the best use of disk drives that is possible. In high performance database systems, multiple drives are utilized for the improved throughput instead of the capacity. This also provides great information on performance tuning block sizes for disk access. To optimize single disk sequential access, we only need to make sure that the seek time is a small fraction of the transfer time (i.e. transfer time $>$ 5 $\times$ position time). Localizing data on the outer half of a disk drive will ensure that random operations will achieve over 90% of the best possible throughput. To optimize random disk access, limit the area of the disk the head will have to traverse. By limiting frequently used data to the outer half of the disk, seek latency can be decreased while still maximizing the number of megabytes accessible [30].

3.4.3 RAID scrubbing

Section 3.3 demonstrates how latent sector errors can degrade the reliability of a disk subsystem. Several studies have shown that monthly data scrubbing can have a dramatic impact on the reliability of disk subsystems, improving reliability by 300% to as much as 900% when compared to annual scrubbing [32]. Disk scrubbing is a feature of enterprise class disk drives where the extra ECC information stored with each sector is compared against the data stored in the sector to verify it is correct. RAID scrubbing is implemented by the RAID controller and is used to verify data in a RAID set in a similar manner. The RAID controller reads every stripe in the RAID set and compares it against the stored parity information. If an unrecoverable read error occurs, the disk drive remaps the sector to a spare sector and the RAID controller regenerates the corrupted data from the parity information and rewrites the data to the remapped sector.

Since RAID scrubbing is implemented in the RAID controller, the procedure to initiate the scrubbing is manufacturer specific. In Linux software RAID, scrubbing can be initiated with the following command:

```
echo check >> /sys/block/mdX/md/sync_action
```

The status of the scrub can be checked with the following command:

```
watch -n .1 cat /proc/mdstat
```

The RAID scrub can be performed while the system is live, though it should be done off hours. A simple script can be executed from a cron job to check all the RAID sets on a weekly basis.

3.4.4 Leveraging a hot-spare

One key factor in most MTTDL models is the amount of time the RAID set remains in a degraded state. This includes the time required to detect the failure, replace the disk, and rebuild the data set using the replacement disk. In a study questioning the usability of MTTDL values, Paris concludes that MTTDL can provide fairly good estimates of reliability as long as the individual disk repair rate remains well above one thousand times the disk failure rate [16]. Basically, MTTDL is accurate as long as failed disks are quickly detected and replaced.

Hot-spares should be leveraged to minimize the amount of time required to detect and replace failed disks. A hot-spare is a disk that is not a part of a RAID set but that is powered on and connected to the RAID controller. In the event that a disk failure occurs, the RAID controller will automatically add the hot-spare to the degraded RAID set and immediately begin regenerating the data. If there are multiple RAID sets on a controller, it is a good idea to configure the hot-spare as a global hot-spare instead of assigning it to a specific RAID set. This will make the most efficient use of the hot-spares in the system.

In designing reliable disk subsystems, there should be at least one global hot-spare configured per RAID controller. The specific number of hot-spares to configure will depend upon the class of disks utilized, the number of disks in each RAID set, the RAID level, and the probability of a disk failure. *It is important to include hot-spares in the disk subsystem design right from the start because it can be difficult, if not impossible, to add them after the fact.*

3.4.5 Replacement strategies (end of life)

Several studies have shown that disk failure rates form a bathtub-shaped plot, similar to Figure 3.1, with the highest failure rates occurring during the first year of operation (infant disk mortality) and after the fifth year (end of life) [37, 57]. An important part of a reliable disk subsystem is having a disk replacement policy in place. Xin developed a Markov model to simulate the effect of disk failure and replacement on the MTTDL values for a storage system [57]. His conclusion is that as the disks in a storage system reach their end of life, replace them in small batches instead of large batches. He recommends to identify all the disks that are scheduled to be replaced in the next year, then replace the disks in $1/12^{th}$ increments each month. His study examined the selection of end of life disks either by age or randomly from the pool of end-of-life disks and found there was not a significant effect on the MTTDL values for the storage system either way. His recommendation is to select disks randomly from the pool of end-of-life disks since it requires less book-keeping work [57].

Why not wait for disks to fail or until the SMART counters reveal a pending disk failure before replacing them? This is a valid question and the answer is that it depends upon the maintenance policies that an IT management has in place. First, there is the issue of warrantees and maintenance agreements. These typically are for three to five year periods following the purchase of the device, and many IT shops require maintenance agreements for all critical systems. Second, it is not possible to reliably predict when a disk drive is going to fail. Google's study of disk drives failures indicated that 56% of their failed disk drives not generate and SMART data indicating a failure was eminent [34]. Proactively replacing end-of-life disks allows the support staff to stagger the introduction of new disks into a storage system to minimize the effects of infant disk mortality. And finally, a five year old disk is a five year old disk. New replacement drives will likely have greater capacity, higher performance, and possibly even improved reliability.

3.5 Other considerations

In this section are a few other items to keep in mind when designing disk subsystems and storage systems in general.

3.5.1 Quality hardware

In a study of latent sector errors conducted by Microsoft Research, one conclusion of the study was that controller errors and OS bugs resulted in more downtime and loss of data than latent sector errors [22]. Even when using Linux software

RAID, carefully consider the selection of the RAID controller hardware. Buggy hardware and/or drivers will have a much more significant impact on the reliability of a storage system than latent sector errors and infant disk mortality. Install server class components from well-known manufacturers with a strong history of providing Linux driver support.

If the plan is to use the RAID controller itself to assemble the RAID sets, ensure that controllers are true hardware RAID controllers. Several quality hardware RAID controller manufacturers include Adaptec, QLogic, LSI, 3ware, and Areca. Avoid using "fake-raid" controllers where part of the RAID controller is implemented in hardware and the other part is in the driver. These include many of the Promise and HighPoint models as well as most onboard RAID controllers except in the case of enterprise class servers. If the RAID controller costs less than the motherboard, be vigilant. Finally, check with the controller manufacturer to determine how to periodically scrub the arrays.

3.5.2 RAID is NOT backup

RAID is not a replacement for a solid backup solution. RAID can help to keep data available and online in the event of disk drive failures, but it will not protect against firmware bugs, driver bugs, or even application bugs that can have the potential to corrupt data. But most importantly, RAID does not protect against human error or a malicious user. It is incredibly easy to delete critical data "accidentally". An easy way to determine if data should be backed up is to tell the data's owner that there was a system error and the data is gone. Their reaction will indicate whether the data should be backed up or not!

3.6 Conclusion

In most organizations, data is the life blood. People can be replaced. Workstations, servers, even entire data centers can be replaced if necessary, but in most cases the organization's data cannot be. This places a substantial burden upon the storage engineers responsible for designing systems to maintain this data. Latent sector errors and infant disk mortality can have a deleterious effect on disk subsystem reliability.

However, there is hope. There are practical techniques that, when applied to disk subsystem design, can dramatically improve disk subsystem reliability.

- Use only nearline or enterprise class disk drives with a bit error rate of 1 in 10^{15} or lower
- Perform an initial burn-in of new disk drives before placing them in a production environment
- Utilize a replication based protection scheme such as RAID-1 or RAID-10 to decrease the probability of encountering an unrecoverable read error during a rebuild
- Implement RAID scrubbing on a weekly or monthly basis to proactively detect and repair latent sector errors
- Utilize a hot-spare to minimize the amount of time a RAID set remains in a degraded state after a disk failure
- Have a disk replacement policy in place to transition end-of-life disk drives out of the storage system without introducing higher risk from infant disk mortality

Following these recommendations will dramatically improve disk subsystem reliability; however, they are not a substitute for a solid backup solution.

CHAPTER 4

THROUGHPUT AND SCALABILITY OF PARALLEL DISTRIBUTED FILE SYSTEMS

4.1 Introduction

Storage has been a recurring issue in the HPC lab at Boise State University. Many of our users' applications require high-performance storage and lots of it. Unfortunately, a storage solution that performs extremely well for one application often performs badly for another. This chapter investigates the throughput and scalability characteristics of four popular open source parallel distributed file systems: PVFS, Lustre, GlusterFS, and HDFS. This information will enable individuals interested in deploying high-performance storage systems to make informed decisions regarding the selection of which parallel distributed file system to use. It will also provide software developers with a good overview of how parallel distributed file systems work so that they can leverage the I/O capabilities of these file systems in their applications.

The Atlantis research cluster and Beowulf HPC cluster were used to generate the performance data presented in this chapter. The parallel distributed file systems were installed on Atlantis, and Beowulf's processing nodes were used as clients. Appendix D provides a network diagram showing how Atlantis was linked to the Beowulf cluster, hardware specifications for each of the storage nodes in Atlantis, and baseline performance results for the disk subsystems on *atlantis01 - atlantis03*.

This chapter is organized as follows. Section 4.2 describes the tests used to evaluate the performance of each parallel file system. These tests reflect how users and applications interact with the file systems in our HPC lab. Section 4.3 provides an overview of how parallel distributed file systems work while Sections 4.4, 4.5, 4.6, and 4.7 provide information about the file distribution techniques and performance characteristics of PVFS, Lustre, GlusterFS, and HDFS, respectively.

4.2 Benchmarking techniques

The throughput and scalability tests presented in the following sections simulate applications that perform sequential I/O operations. The reason for this emphasis on sequential I/O is that this research is the result of weeks spent diagnosing application I/O performance issues in the HPC lab at Boise State University where the majority of the applications that require high-performance storage in our lab perform sequential I/O operations. It would be equally interesting to examine performance and scalability of these file systems for applications that perform non-sequential (random) I/O operations, but that is beyond the scope of this chapter.

4.2.1 Testing environment

The testing results in Section 4.3 were generated using the Atlantis research cluster. Appendix D provides a description of Atlantis including a network diagram and baseline throughput results for the disk subsystem configurations used for the testing in Sections 4.4, 4.5, and 4.6. For the PVFS, Lustre, and GlusterFS file system testing, *atlantis00* was used as the client for the basic file transfer tests as well as the block transfer tests. The Beowulf cluster was used for the client scalability tests while *atlantis01*, *atlantis02*, and *atlantis03* provided the storage and metadata services for each of the parallel distributed file systems that were tested.

4.2.2 Basic file transfer test

The basic file transfer test measures the throughput in MB/s for a single client attempting to transfer data to/from the storage system using the `cp` command. This is performed using two different sets of data. The first, dataset1, is a single 5.6GB gzip file containing gene data from the NCBI FASTA repository. The second, dataset2, is a folder containing 160 bacteria genome datasets from the NCBI Genomes repository, each containing hundreds of files with a total size of 5.4GB. To minimize the effects of client file caching, these datasets are more than two times the amount of memory in the client systems.

4.2.3 Block-range file transfer test

The block transfer test measures the throughput in MB/s for a series of read and write operations between a single client and the storage system using a range of block sizes from 4KB up to 8MB. To measure the write performance of the storage system, the `dd` command is first used to read from `/dev/zero` and write to a file located on the storage system. The block size is set to the current test size, and block count is calculated dynamically based upon the block size and the total system memory so that the total transfer size is twice the amount of system memory. The read performance is then measured by transferring the previously written file from the storage system to `/dev/null`. The `dd` command is used for this operation as well, with the corresponding block size.

4.2.4 Client scalability test

The client scalability test gauges how the storage system performs under significant client load when there are several times more clients than there are storage nodes. This test examines aggregate throughput of the storage system as the number of clients increases. The tool used to perform the scalability testing is a custom written benchmark tool called `fs-perf`. This tool spawns multiple client processes on our Beowulf HPC cluster, each of which performs synchronized read and write operations against the storage system. These read and write operations are performed using the ROMIO implementation of the MPI-IO interface instead of the kernel VFS system calls.

To measure the aggregate throughput of the storage system, each client process tracks the amount of time required to perform its current I/O operation. Since all the processes are started simultaneously and each is transferring 4GB of data (two times the amount of memory on the client nodes), a lower bound for the aggregate throughput can be determined by dividing 4GB by the max (longest) client transfer time, and multiplying that by the number of clients.

4.3 "Parallel" distributed file systems overview

Traditional distributed file systems provide a single unified namespace for accessing data that may be spread across thousands of disk drives attached to hundreds of storage nodes. This can provide large-scale capacity and even data redundancy if replication is enabled, but this system is unable to provide I/O throughput at a level that will satisfy the needs of modern HPC environments. Parallel distributed file systems were developed to address this issue of high performance I/O throughput.

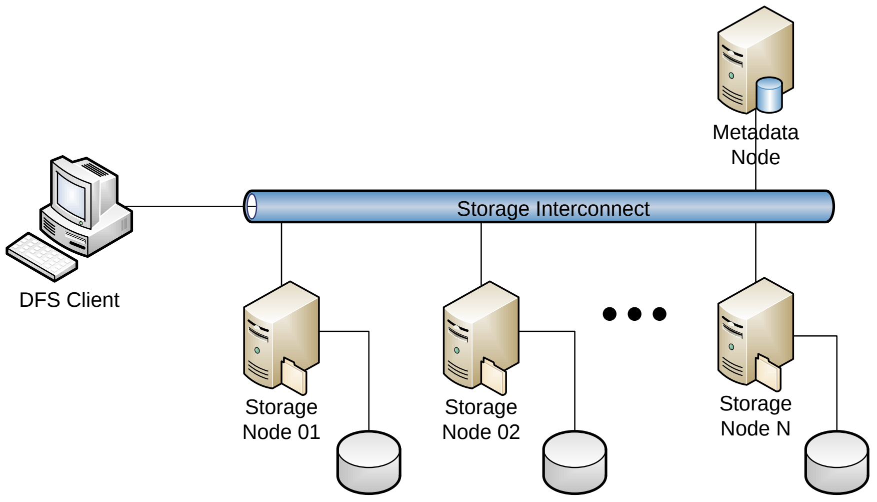


Figure 4.1: Example of “parallel” distributed file system architecture

The “parallel” in parallel distributed file system refers to having multiple storage servers with the file data distributed across them to increase both the aggregate bandwidth as well as the storage capacity. When a file is accessed, the client communicates with the metadata server to obtain a map of storage nodes that contain each piece of the file. The client then communicates directly with the storage nodes to access the file data.

Figure 4.1 shows the components that make up a parallel distributed file system. The metadata server maintains the information required to locate and identify a file, and it is responsible for the create, access, and modify time-stamps as well as user and group ACLs. The metadata server also is responsible for responding to client file operations and determining how files are distributed across (assigned to) the storage servers. The storage servers store the file data itself.

Not all parallel distributed file systems strictly follow the architecture shown in Figure 4.1. Some allow multiple metadata servers while others do not require any, choosing instead to store metadata with the associated data directly on the storage servers and giving the storage clients a hash function to locate files within the storage system. In addition, it is possible to run the metadata, storage, and client components all on the same physical storage node within a storage system.

A parallel distributed file system is designed to distributed client I/O requests across two or more storage nodes to minimize performance bottlenecks (hot spots) within the storage system. The way this distribution of client access is accomplished is file system dependent, but it typically uses one of the following techniques: simple file distribution, file striping, or file replication.

4.3.1 Simple file distribution

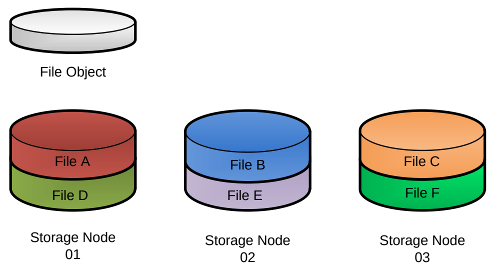


Figure 4.2: Simple file distribution technique

The simple file distribution technique distributes the creation of new files across the nodes in a storage system. There are many methods for performing this distribution from a simple round-robin approach to the use of a hash function. The goal of the distribution method can be to balance disk usage across storage nodes or it can be to balance the number of files per node. The assignment of files to storage nodes can either be managed by a dedicated metadata server, or, in the case of a hash function, the client can directly calculate where to store or retrieve a file. Figure 4.2 shows a sample distribution of six files across three storage nodes.

The advantage of simple file distribution is that each file is stored entirely on a single storage node. This reduces the overhead required to access each file and is ideal when dealing with many small files. This is also a benefit when attempting to recover from the failure of a storage node. Since the files exist entirely on a single storage node, all the files in the storage system, except for the files on the failed node, remain accessible.

The downside of simple file distribution is that each file is stored entirely on a single storage node. This can create "hot spots" within the storage system where multiple clients are attempting to access the same file. The result is that storage nodes with popular files can be overloaded with client requests while other nodes within the same storage system remain idle.

4.3.2 File striping

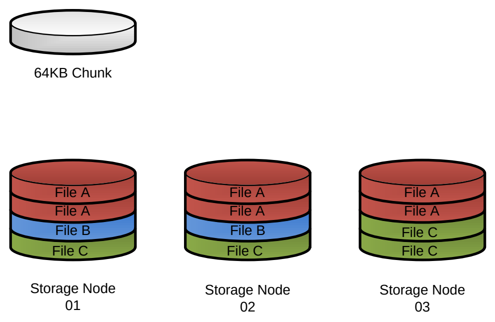


Figure 4.3: File striping distribution technique

The file striping technique breaks a file into chunks of a specific size commonly referred to as the *chunk size*, *block size*, or *stripe size*. These chunks are then distributed across the storage nodes. In the event that a large number of clients attempt to access the same file, the requests are distributed across the entire storage cluster instead of all the clients accessing a single storage node. Figure 4.3 is a diagram of how this file striping might look across three storage nodes.

The advantages of using striping are increased bandwidth and support for very large file sizes [12]. When multiple clients are attempting to access a file stored entirely on a single storage node, there can be a substantial I/O bottleneck. Striping the file across multiple storage nodes increases the total aggregate bandwidth linearly with the number of storage nodes. Striping is also beneficial if there is a file too large to be practically contained on a single storage node. An example is an application that generates a 32TB output file.

Unfortunately there are also disadvantages to striping, specifically the communication overhead and increased risk. The communication overhead occurs because a client has to establish connections with multiple storage servers to retrieve the file chunks. Common operations such as stat and file locks require multiple network operations instead of a single operation. The increased risk comes from having files spread across all of the storage nodes. If one storage node fails in a storage system with n storage nodes, *1/n* of every file in the storage system is lost, meaning that all file on the storage system are corrupted. In contrast, the failure of a single storage node in a file system configured to use simple file distribution means that only *1/n* of the files are lost. The other *(n-1)/n* of the files are unaffected.

4.3.3 File replication

The file replication distribution technique replicates entire files across two or more storage nodes. Figure 4.4 shows a diagram of the file replication technique with a

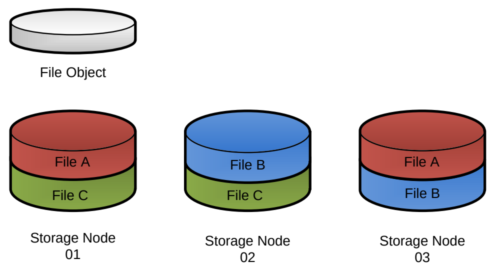


Figure 4.4: File replication distribution technique

replication factor of two. The creation of replicated files can be strictly enforced, requiring all replicas to be committed before returning a success write message to the client. Or it can be lazy, meaning that the write request is completed once the file is stored on a single storage node. The file system then schedules an internal task to fulfill the requirements of the replication policy.

The advantages of data replication are improved read performance and reliability. There is improved read performance because multiple clients can read from different replicas simultaneously. There is improved reliability because even in the event of a catastrophic loss of a storage node, the storage system will be able to continue operation with no loss of data and relatively low impact on the end-user.

The disadvantages of data replication are poor write performance and reduced storage capacity. The severity of the write performance penalty depends upon how the distributed file system enforces replication. If the file system manages the replication behind the scenes using a lazy replication approach, the performance loss will be minimal. If the client must wait until the file is replicated or if the client itself must write the file to multiple storage nodes, the write performance penalty will be more severe. The issue of reduced storage capacity is a result of storing multiple copies of each file. A replication factor of three results in the loss of 2/3 of the total capacity of the storage system. To demonstrate this issue, a 90TB storage system configured with a replication factor of three will only have 30TB of usable storage capacity. Try explaining that one to your boss!

4.4 Parallel Virtual File System (PVFS)

4.4.1 Background

Parallel Virtual File System (PVFS) is a parallel distributed file system designed for high-performance computing environments. The first version was developed in the mid 90s at Clemson University. As PVFS became more widely used in production environments it was leveraged in ways not anticipated by the original designers. The second version of PVFS introduced a modular architecture that is better able to meet the performance needs of its growing user base. PVFS is under active development by a number of universities and research institutions including Clemson University, Argonne National Laboratory, and the Ohio Supercomputer Center [48].

4.4.2 Architecture

The architecture of a PVFS based storage system utilizes a Metadata Server (MDS) and I/O Server (IOS) components. PVFS uses a single Linux process called pvfs2-server that implements both the MDS and IOS roles. This process is light weight and runs in user space on the storage nodes. PVFS uses local file systems such as ext3, ext4, and XFS for the backend datastores. Client access to the file system can be handled entirely in user space via the PVFS system interface API or using MPI-IO. Unix I/O access is provided by a VFS kernel module and a pvfs2-client process running on the storage client [9]. PVFS supports running multiple MDS nodes in the storage system to improve metadata performance when working with large numbers of files.

4.4.3 Distribution techniques

PVFS uses the technique of data striping to distribute data across the IOSs in the storage system. PVFS provides a modular approach for defining the striping of data across the IOSs. These modules are called distributions. PVFS includes four of these distributions that users can select from, but it also provides the capability for users to write their own. The included distributions are *Simple Stripe*, *Basic*, *Two Dimensional Stripe*, and *Variable Stripe*. By default, Simple Stripe is used with a stripe size (or chunk size) of 64KB [49]. The PVFS tuning guide provides limited information regarding the use of these distributions. With the Simple Stripe distribution, it is possible to set custom stripe sizes on specific folders within the PVFS file system. The commands to accomplish this can also be found in the PVFS tuning guide.

4.4.4 Feature summary

Table 4.1 summarizes the file system access mechanisms, file distribution techniques, and supported network interconnects for PVFS.

Table 4.1: Summary of PVFS design features

Software Interfaces	Unix I/O via Linux VFS kernel modules, PVFS system interfaces, ROMIO [9]
File Distribution Techniques	Simple Stripe, Basic, Two Dimensional Stripe, and Variable Length Stripe [49]
Network Interconnects	Infiniband, GigE, 10GigE, IPoIB, and Myrinet [48]

4.4.5 Performance characteristics

PVFS test environment configuration

The default configuration of PVFS recommends that pvfs2-server be configured to run both the MDS and IOS roles on each storage node, and that is the configuration used for the following throughput and scalability testing. The benefit of this configuration is that it allows all three of the storage nodes in Atlantis to be used to store data. As a result, this increases the total aggregate throughput to be on the order of 300MB/sec for the client scalability testing compared to only 200MB/sec when utilizing only two storage nodes for data. Table 4.2 shows the configuration of PVFS on the Atlantis research cluster.

Table 4.2: Summary of PVFS configuration on Atlantis

atlantis01	I/O Server and Metadata Server
atlantis02	I/O Server and Metadata Server
atlantis03	I/O Server and Metadata Server
atlantis00	PVFS Client
Beowulf Cluster	PVFS Client

Basic file transfer

Figure 4.5 shows the performance of the `cp` command transferring the test datasets to and from the PVFS file system with a read throughput of 5MB/sec to 6MB/sec and a write throughput of 17MB/sec to 26MB/sec. For this test, the datasets were read from and written to a dedicated disk subsystem on the client (*atlantis00*). This disk subsystem is capable of sustaining 155MB/sec write throughput and more than 200MB/sec read throughput (see table D.2 in Appendix D); therefore, it is clear that

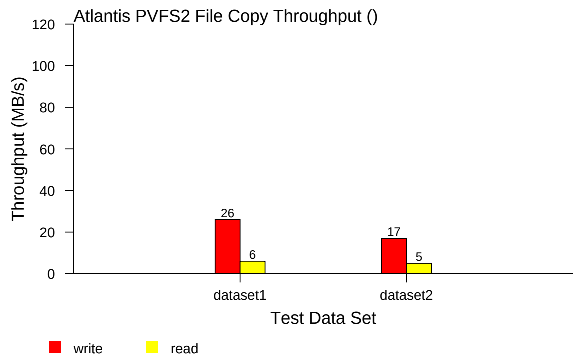


Figure 4.5: PVFS basic file transfer with file striping (64KB)

the performance is not limited by the client disk subsystem. The block-range file transfer test result show why this is the case.

Block-range file transfer

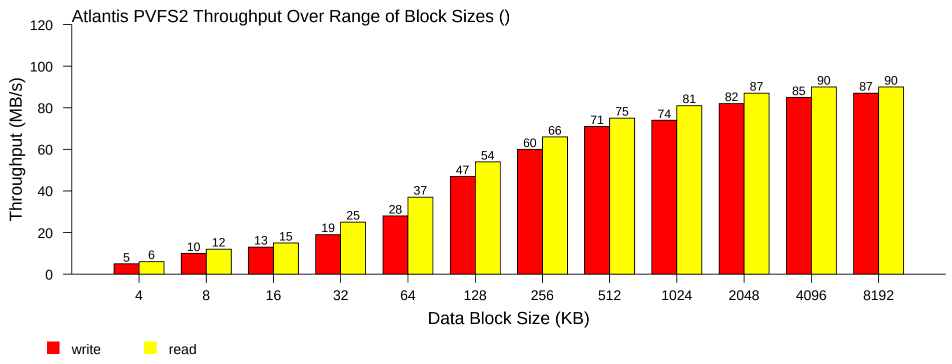


Figure 4.6: PVFS block-range file transfer with file striping (64KB)

The throughput results for PVFS shown in Figure 4.6 are probably the most interesting of the three file systems examined in this paper. This is because PVFS does not appear to perform any read-ahead or write-back caching, and as a result the throughput of the file system varies directly with the block size used by the application. Applications that perform I/O operations using 4KB blocks, which is the default buffer size allocated by calls to `fopen()`, have terrible performance when running on PVFS. Overriding this buffer size is accomplished with a call to `setvbuf()` for each file that is opened. Based upon the results in Figure 4.6, the ideal buffer size is in the area of 2MB.

Evidence of the profound impact that block size has PVFS is in the poor throughput results of the basic file transfer test in the previous section. An investigation into this issue revealed that the `cp` command makes a call to `fstat()` to determine the block size to use when copying a file. For files stored on a PVFS file system `fstat()` returns a 4KB block size. This is an important issue to consider when choosing PVFS over another parallel distributed file system. Chapter 5 explores the issue of buffer size in more detail.

Client scalability

Figure 4.7 shows how the PVFS storage system scales. Notice the near linear scaling of aggregate throughput performance as the number of clients reaches the number of storage nodes. With four clients, the storage system reaches the anticipated aggregate throughput of 300MB/sec, and the overall throughput for the read common, read individual, and write individual tests remains in that ballpark as the number of clients increases to more than five times the number of storage nodes. Another important point is that these tests were performed using the MPI-IO implementation provided by ROMIO to access the PVFS storage system. The basic file transfer and the block-range tests both used the PVFS Linux VFS kernel module.

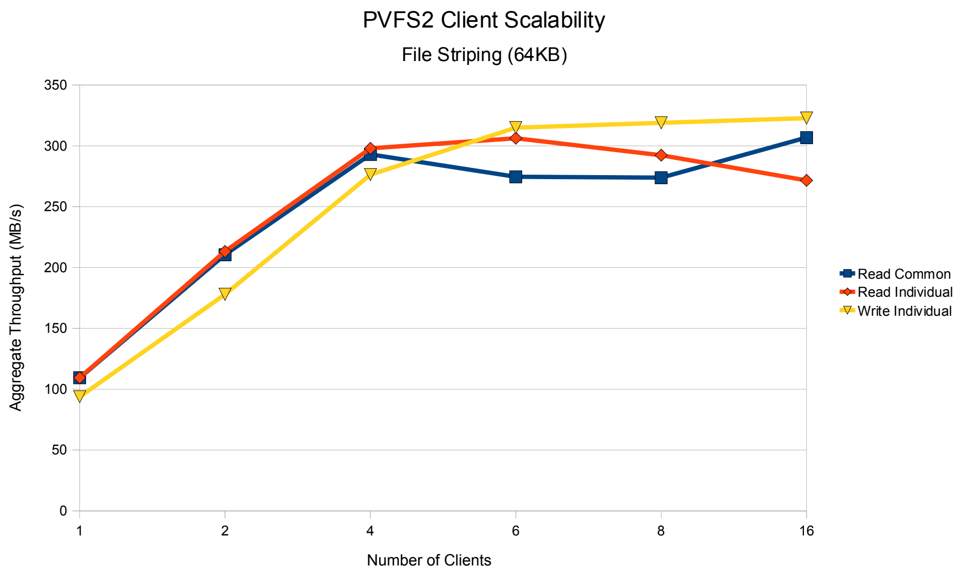


Figure 4.7: PVFS client scalability with file striping (64KB)

4.5 Lustre

4.5.1 Background

Lustre is a parallel distributed file system designed for HPC environments. It started as a research project led by Peter Braam at Carnegie Mellon University in 1999. Peter Braam went on to found ClusterFS, Inc, which was purchased by Sun in 2007. Sun was then purchased by Oracle in 2009. Lustre is currently being actively developed and supported by Oracle and the community at large and can be found on many of the largest supercomputer systems in the world, including the currently number one ranked Jaguar supercomputer at Oak Ridge National Laboratory [51]. The Lustre file system deployed on Jaguar is called Spider. The website for Jaguar says the following:

"... It is the largest-scale Lustre file system in the world with 26,000 clients, and it is the fastest in the world with a demonstrated bandwidth of 240 GB/s. Spider currently provides 5 PB of disk space [14]."

4.5.2 Architecture

The developers of Lustre took a significantly different approach in the development of a parallel distributed file system than the developers of PVFS. In PVFS the metadata and data storage services run in user space while the Lustre implementations of these same services are in kernel space. There are several advantages and disadvantages to this approach. The advantages are that Lustre is able to tune many aspects of the kernel specifically for high-performance data storage. These include customizations of everything from the TCP stack to adding Lustre-specific hooks into the ext3 and ext4 file systems. The disadvantages are that storage nodes that run Lustre require custom-patched kernels, as the changes required for Lustre cannot be exclusively handled using kernel modules. Oracle provides prebuilt kernels for most major Linux distributions, but these have the potential to lead to compatibility issues with future software updates.

A Lustre storage system consists of a Metadata Server (MDS) and multiple Object Storage Servers (OSS). The MDS mounts a block device called a Metadata Target (MDT) that is used to store the metadata for the files and directory structure of the storage system. Each OSS mounts one or more block devices called Object Storage Targets (OST). A single storage node can provide both the MDS and OSS roles within a storage system. Unlike PVFS, which allows multiple metadata servers to be active simultaneously, Lustre allows only a single active MDS within a storage system. It does, however, allow a second MDS to be configured for Active/Passive high-availability failover.

4.5.3 Distribution techniques

Lustre is similar to PVFS in that it allows both simple distribution and striping of files across the storage nodes in the storage system; however, Lustre provides extensive control of how and where data is stored. First, PVFS requires all of the disks within a storage node to be combined into a single block device that is then used as a backend datastore by the pvfs2-server service. The modularity of Lustre enables multiple disk subsystems with different disk and RAID configurations to be mounted as OSTs within a single storage node and accessed by a single OSS service. Second, file distribution (striping or simple) occurs at the OST level within Lustre, not at the storage node level. And finally, Lustre enables the creation of OST pools, subsets of OSTs within the storage system that data is distributed across [12] . An example of how this can be utilized is in a storage system with a mix of disk subsystems composed of 15k SAS disks and 7.2k SATA disks. The 15k disks could be combined into OST pools that are separate from the 7.2k disks. Then a folder could be set up so that files written to that folder were striped across the 15k disks only, ensuring high performance for applications accessing data contained within that folder.

Lustre provides a great deal of flexibility in the configuration of striping parameters. Striping rules can be defined that apply to the entire file system, specific folders, or individual files. These rules define whether striping should be enabled and, if so, how many OSTs to stripe the file data across. It allows custom stripe sizes from 64KB up to 4GB in size, though in practice only stripes in the range of 512KB to 4MB are recommended [12].

With striping disabled, Lustre will attempt to balance the creation of new files across the OSTs. This is critical to distributing client access across the storage system but also to balance disk usage. One issue with Lustre is that once an OST becomes full, the entire Lustre file system is no longer able to store additional data, even if there are other OSTs with available space [12]. Section 19.1 of the Lustre Operations Manual describes how to deal with full OSTs. Several options include taking nearly full OSTs offline, rebalancing data using OST pools, or enabling striping for directories containing large files.

4.5.4 Feature summary

Table 4.3 summarizes the file system access mechanisms, file distribution techniques, and supported network interconnects for Lustre.

Table 4.3: Summary of Lustre design features

Software Interfaces	Standard VFS via custom Linux kernel (or kernel module) with Direct I/O Support, LLAPI for programmatic control of file striping, and ROMIO [12, 50]
File Distribution Techniques	Simple File Distribution and File Striping [12]
Network Interconnects	Infiniband, GigE, 10GigE, IPoIB, Quadrics Elan, Myrinet, Cray [12]

4.5.5 Performance characteristics

Lustre test environment configuration

The installation of Lustre on Atlantis is slightly different than that of PVFS and GlusterFS due to the requirement of a dedicated disk subsystem for the metadata server. As a result, the Lustre performance tests will only use two storage nodes for data instead of three. This limits the expected aggregate throughput to 200MB/sec. This should not affect the performance results of the basic file transfer and block-range transfer tests, but it will affect the aggregate throughput levels of the client scalability tests. Table 4.4 shows the configuration of Lustre on the Atlantis research cluster.

Table 4.4: Summary of Lustre configuration on Atlantis

atlantis01	Metadata Server (MDS)
atlantis02	Object Storage Server (OSS)
atlantis03	Object Storage Server (OSS)
atlantis00	Lustre Client
Beowulf Cluster	Lustre Client

Basic file transfer

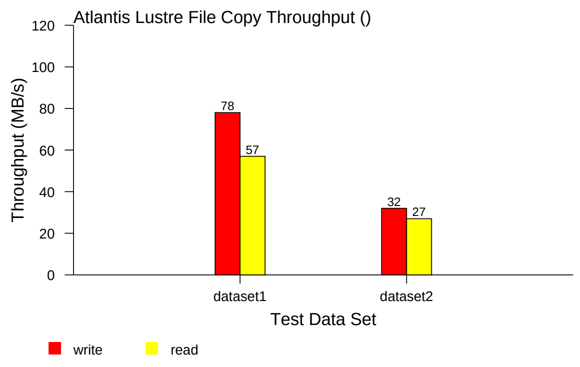


Figure 4.8: Lustre basic file transfer with simple file distribution

Comparing the results in Figure 4.8 with those from PVFS in Figure 4.5, there is a marked improvement in throughput for both dataset1 and dataset2, especially in the read performance. The reason for this is two fold. First, according to the Lustre Operators Manual[12], Lustre uses a network chunk size of 1MB. This helps hide much of the latency and overhead of communicating over a GigE interconnect by reducing the number of packets that must be sent. Second, when the `cp` makes the call to `fstat()` to determine the block size of the file system, Lustre returns a significantly larger block size (2MB) than PVFS.

Block-range file transfer

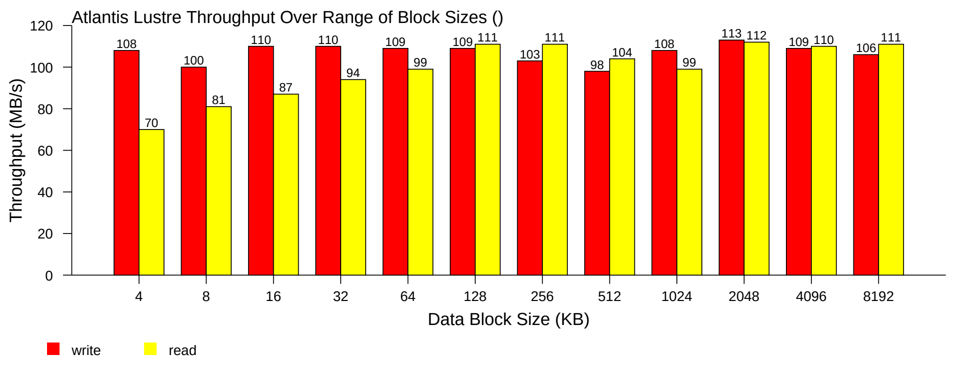


Figure 4.9: Lustre block-range file transfer with simple file distribution

Figure 4.9 provides the throughput results for file transfers using a range of block sizes. Notice that there is no appreciable difference in write throughput over the entire range of block sizes, and while the read performance does improve with increasing block sizes, Luster is able to reach full throughput with a block size of only 128KB. This is in sharp contrast to the results for PVFS shown in Figure 4.6.

From the Lustre Operations Manual:

"LNET offers extremely high performance. It is common to see end-to-end throughput over GigE networks in excess of 110MB/sec, InfiniBand double data rate (DDR) links reach bandwidths up to 1.5GB/sec, and 10TGigE interfaces provide end-to-end bandwidth over 1GB/sec [12]."

Many applications are tuned to perform file I/O in 4KB or 8KB blocks. This is not an issue when performing I/O operations on a local disk subsystem because the

SATA, SAS, and Fibre Channel protocols all have relatively low latency. However, when applications developed for these local file systems are moved to a network file system, there is a tremendous penalty because of the small block size. The developers of Lustre attempt to address this issue by performing read requests in large blocks, and caching write requests to allow for higher throughput at these smaller block sizes.

Client scalability

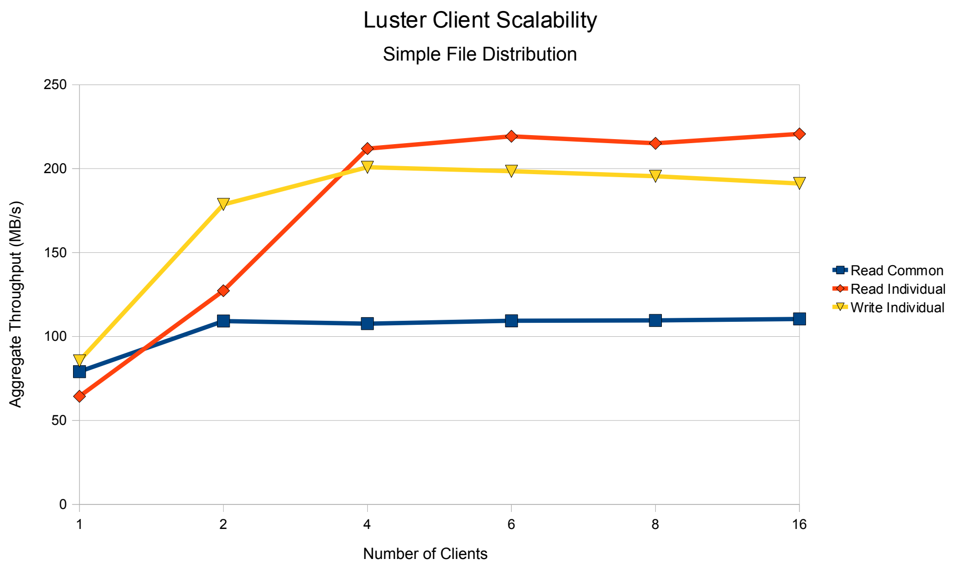


Figure 4.10: Lustre client scalability with simple file distribution

The final performance characteristic to examine is throughput performance as the number of clients increases to several times the number of OSSs. Figure 4.10 shows the aggregate throughput as the number of clients increases. Since the Lustre installation on Atlantis contains only two Object Storage Servers, each with a single gigabit Ethernet link, the maximum theoretical throughput is 250 MB/s. This value is not reachable in practice, but the read individual and write individual performance

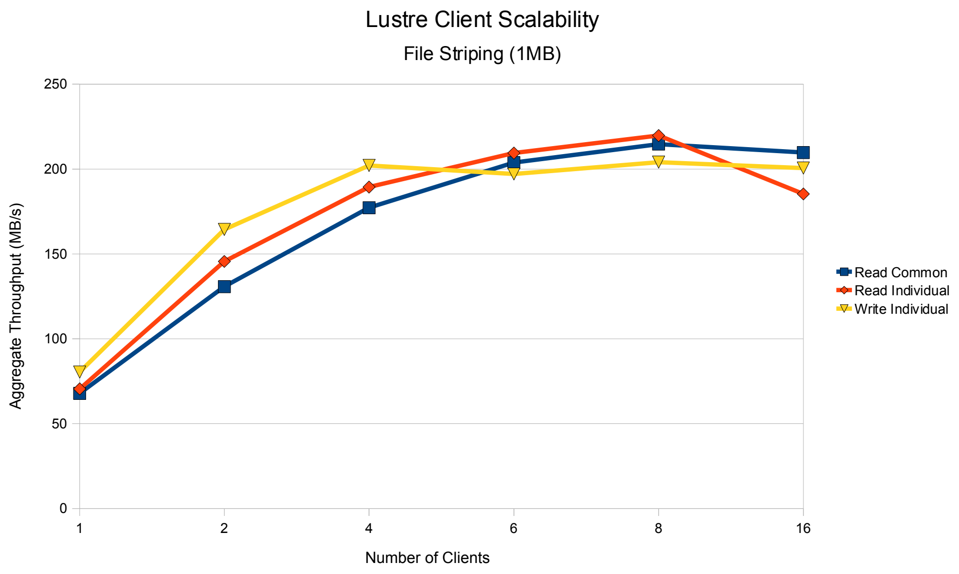


Figure 4.11: Lustre client scalability with file striping (1MB)

tests reach 80-85% of this value, which is remarkable. With only four clients, the storage nodes are able to be fully saturated. Another important item to note is that as the number of clients increases, there is not a significant drop in throughput as a result of the increased number of connections. This is important for being able to scale to a large number of clients.

The results in Figure 4.10 were generated with Lustre configured for simple file distribution. The results in Figure 4.11, on the other hand, were generated with Lustre configured for file striping with a 1MB chunk size. Notice that the Read Individual and Write Individual results are nearly identical between the two. The reason for this is that when Lustre is configured for simple file distribution, the metadata server still attempts to distribute load across the OSSs by balancing the creation of new files across all the storage nodes. Since the `fs-perf` tool generates uniform output files there is an even distribution of traffic as the number of clients increases. However, if the clients were generating files of various sizes or attempting to read from the same file, this would result in contention.

Figure 4.10 provides an example of file contention in the throughput results for the Read Common test. The use of simple file distribution instead of file striping means that all the clients are attempting to access the same file located on a single OST. This large number of clients quickly saturates the network of the OSS containing the OST with the popular file. Figure 4.11 demonstrates that this bottleneck can be avoided by leveraging file striping.

4.6 GlusterFS

4.6.1 Background

GlusterFS is an open source distributed file system designed to meet the storage requirements of a wide range of environments, not only HPC. Unlike PVFS and Lustre, which had their beginnings in the educational/research sector, GlusterFS is a corporate contribution. The team that eventually developed GlusterFS was originally assembled at California Digital Corporation in 2003 to build a supercomputer for Lawrence Livermore National Laboratory. At the time, this computer system (Thunder) was ranked the 2nd fastest in the world [18].

Leveraging this experience, the team founded Z-Research and began building custom cluster solutions. These cluster solutions had diverse storage requirements and the storage options at the time were unable to meet these requirements. The Z-Research team began development on GlusterFS, designing it from the ground up. Some of the design goals included operation in user space, elimination of the metadata server, modular design, and non-proprietary file formats. Gluster is a play on the words GNU and cluster. Z-Research changed their name to Gluster to honor their roots [18]. GlusterFS is actively developed and supported by Gluster, as well as an active user community [18].

4.6.2 Architecture

GlusterFS has a modular design that allows for a high level of customization. GlusterFS uses the concept of storage bricks and translators that can be layered in a variety of configurations to meet a particular set of storage requirements. Translators can customize everything from communication protocols and data distribution techniques to performance features such as readahead and writebehind caching.

GlusterFS is well suited for use as a network file server with 5TB of storage or as a HPC storage solution with 2PB of storage. A unique feature of GlusterFS that makes it different from PVFS and Lustre is that it does not have a metadata server. GlusterFS is able to accomplish this by pushing more intelligence to the client and by utilizing the extended attributes of the underlying file systems on the storage nodes.

4.6.3 Distribution techniques

A GlusterFS storage system can be configured to simply distribute files across storage nodes based upon a hash that is performed by the client. Since all clients use the same hash function, they can directly calculate which node holds a given file. For applications with a lot of small files, this can be a huge performance benefit since there will not be a bottleneck at the metadata server. This simple distribution technique can be implemented using the “cluster/distribute” translator [20].

To reconfigure the storage system for replication or striping, simply replace the "cluster/distribute" translator with the "cluster/replicate" or "cluster/stripe" translator respectively [20].

GlusterFS provides translators that can customize the performance of the system by controlling readahead and writebehind cache functionality. The translators for data distribution, readahead, and writebehind are all client side translators.

4.6.4 Feature summary

Table 4.5 summarizes the file system access mechanisms, file distribution techniques, and supported network interconnects for GlusterFS.

Table 4.5: Summary of GlusterFS design features

Software Interfaces	Native FUSE interface, NFS, CIFS, DAV, and Booster
File Distribution Techniques	Simple File Distribution, File Striping, and File Replication
Network Interconnects	Infiniband, GigE and 10GigE [19]

4.6.5 Performance characteristics

GlusterFS test environment configuration

The configuration of GlusterFS on Atlantis is similar to that of PVFS. Since GlusterFS does not require a separate metadata server, all three of the storage nodes on Atlantis can be used for data. This gives an expected aggregate throughput of 300MB/s for the client-scalability tests. Table 4.6 shows the configuration of GlusterFS on the Atlantis research cluster.

Table 4.6: Summary of GlusterFS configuration on Atlantis

atlantis01	Storage Brick
atlantis02	Storage Brick
atlantis03	Storage Brick
atlantis00	GlusterFS Client
Beowulf Cluster	GlusterFS Client

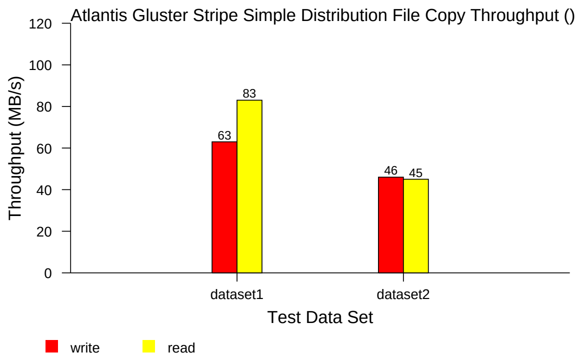
One of the aspects of GlusterFS that makes it so appealing is its modular design and its ability to be adapted to meet a wide variety of storage requirements. When looking at the performance characteristics of GlusterFS, it is important to examine several of the common file distribution configurations. The following performance results are provided for a simple distribution configuration using the "cluster/distribute" translator, a replication configuration using the "cluster/replicate" translator, and finally a striping configuration using the "cluster/stripe" translator.

Basic file transfer

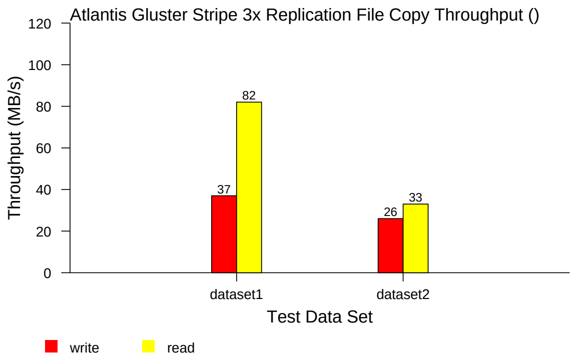
The performance results in Figure 4.12a show the best overall performance of all the tested distributed file systems for a single client reading from and writing to a storage system. These results were good for reading and writing both large files (dataset1) and large numbers of small files (dataset2).

Figure 4.12b shows the throughput results for a replicated configuration of GlusterFS. The file system was configured for 3x replication and the results demonstrate the write performance penalty for performing replication. The replication implementation in GlusterFS requires the client to enforce the replication requirements by performing all write and update requests on each server in the replication pool.

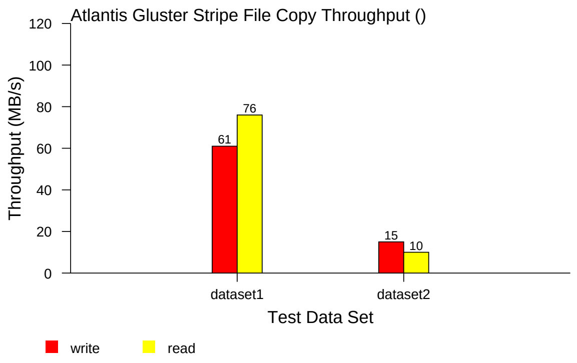
Figure 4.12c demonstrates how the overhead of file striping can impact performance. For transferring large files (dataset1), the performance impact is not



(a) Simple file distribution



(b) File replication (3x)



(c) File striping (128KB)

Figure 4.12: GlusterFS basic file transfer using various distribution techniques

significant when compared to the simple file distribution performance in Figure 4.12a. However, when working with thousands of small files (dataset2), the performance impact is substantial.

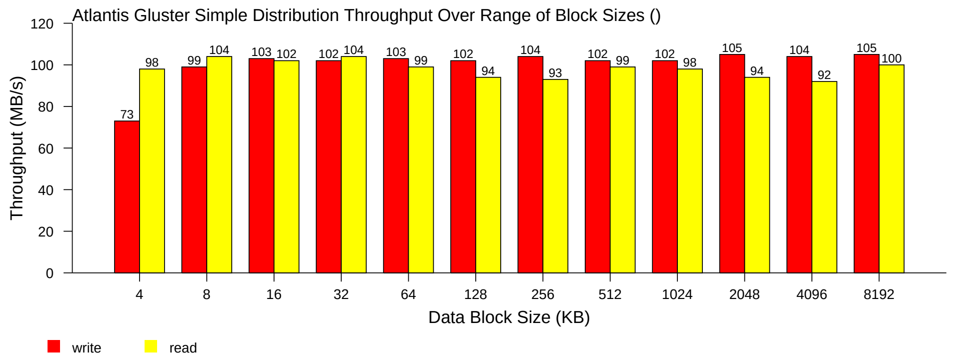
Block-range file transfer

The nice, consistent throughput performances, regardless of blocksize, shown in Figures 4.13a, 4.13b, and 4.13c are the result of the readahead and writebehind translators. These translators allow the communication between the client and the servers to use larger chunk sizes, which is desirable in high latency protocols such as gigabit Ethernet. The writebehind buffer size is 4MB and the readahead buffer is four pages.

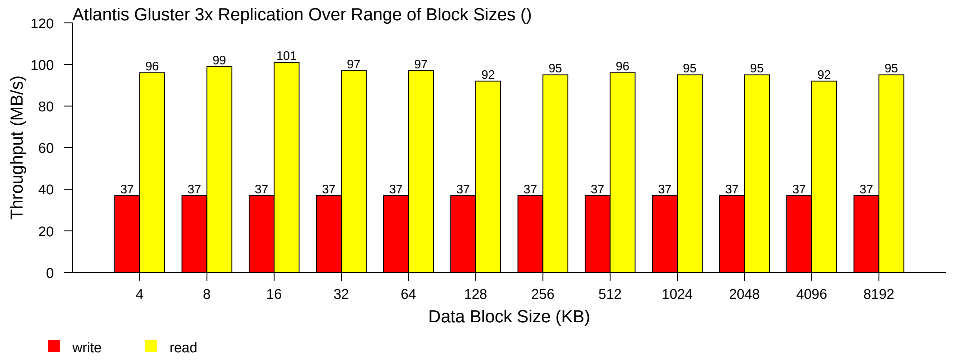
The poor write performance in Figure 4.13b is a side effect of file replication. The replication factor for this GlusterFS configuration is 3x, which explains the fact that the write performance is only about one third that of the simple distributed and the striped configurations.

Client scalability

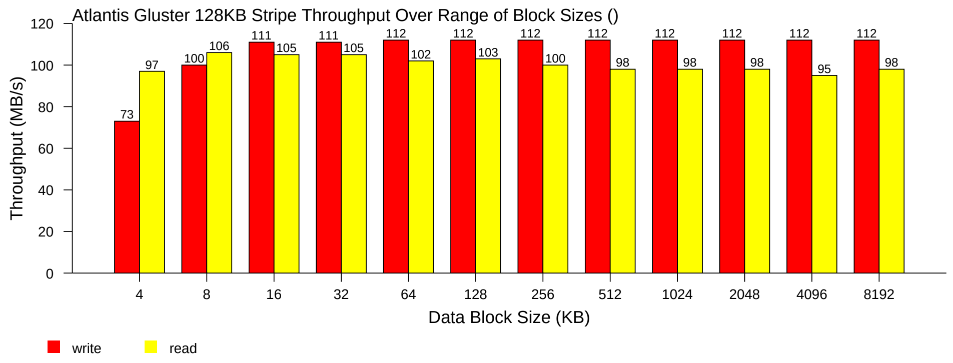
Figure 4.14 shows the aggregate throughput of GlusterFS using simple file distribution as the number of clients increases from one to sixteen. GlusterFS is configured on three storage nodes, so the expected aggregate throughput should be in the 300MB/sec range for the read individual and write individual tests. However, since GlusterFS uses a hash mechanism to distribute files across the storage system, a balanced distribution of files is difficult to achieve when working with a small number of files. This leads to hot spots in the storage system where one storage node has more files and thus more traffic than the other storage nodes.



(a) Simple file distribution



(b) File replication (3x)



(c) File striping (128KB)

Figure 4.13: GlusterFS block-range file transfer using various distribution techniques

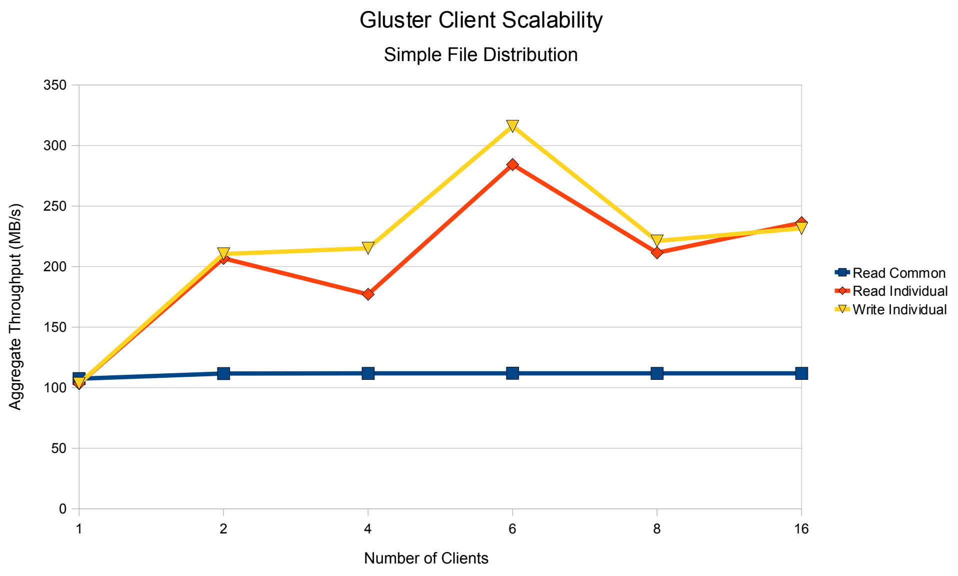


Figure 4.14: GlusterFS client scalability with simple distribution configuration

In the simple file distribution configuration, each file is stored on a single storage node with newly created files assigned to storage nodes based upon a hash function. The slow performance for the read common tests is because all the clients are accessing a single storage node simultaneously. The 100MB/sec shown in Figure 4.14 is the maximum throughput for that storage node.

The fairly ragged throughput results shown in Figures 4.14 and 4.15 raised some concerns during the initial testing, so the tests were run multiple times to ensure the accuracy of the results. It is likely that the throughput drops with four and eight clients as well as the spikes at six clients are the result of the hash mechanism employed by GlusterFS to map files to storage nodes. The file names used in the testing were generated programmatically by incrementing a digit appended the the end of a base file name, resulting in an uneven distribution of client traffic across the backend storage nodes. The effectiveness of the hash algorithm in balancing load

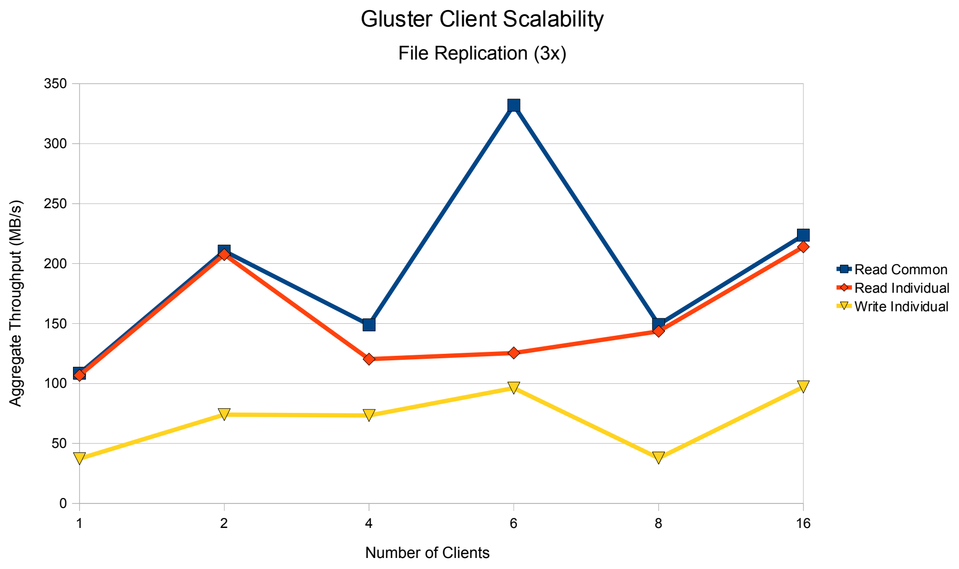


Figure 4.15: GlusterFS client scalability with 3x replication configuration

across the storage nodes will likely improve as the number of clients writing files increases.

A cool feature of the GlusterFS implementation of replication is the fact that client read requests can be balanced across multiple storage nodes [20]. This explains how the read common throughput can be so much higher than for the simple distributed configuration in Figure 4.14.

Figure 4.16 shows a smooth scaling of write individual and read common throughput as the number of clients increases, comparable to that of PVFS. However, the interesting and somewhat unexpected results are the aggregate Read Individual throughput values of only 100MB/s. These results were consistent across multiple tests, but an explanation has not been forthcoming.

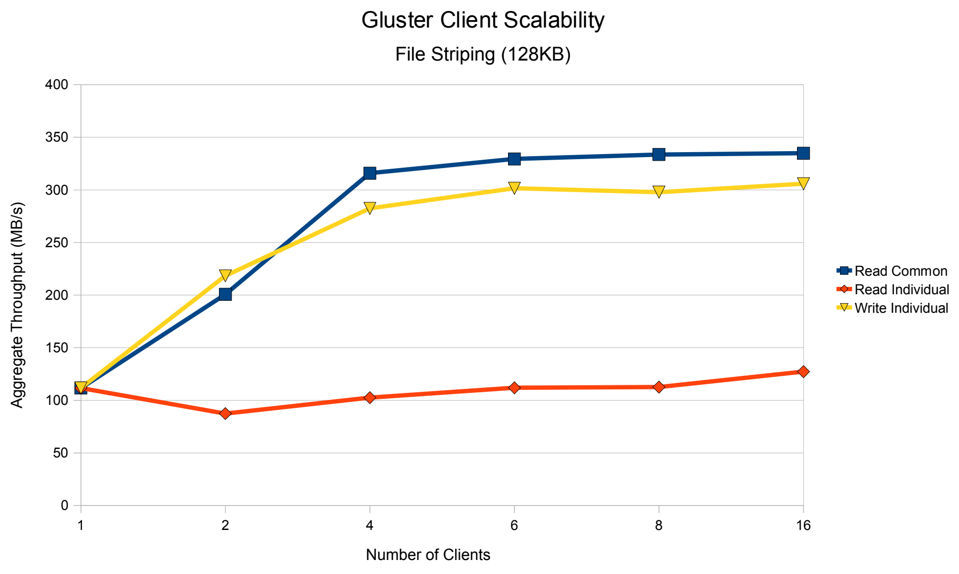


Figure 4.16: GlusterFS client scalability with 128KB stripe configuration

4.7 Hadoop Distributed File System (HDFS)

4.7.1 Background

The Hadoop Distributed File System (HDFS) is a distributed file system specifically designed to be used as the backend datastore for Hadoop applications. Originally designed for the Apache Nutch web search engine project, it has now been broken out as an Apache Hadoop subproject [15]. Until recently, the use of HDFS has been limited to applications that were built using the Hadoop platform; however, the recent addition of a fuse interface allows HDFS to be used as a general purpose distributed file system. HDFS is highly fault-tolerant, designed with the expectation that at any point in time one or more data storage nodes may be unavailable. These traits make this file system extremely appealing to storage system designers. The following sections will explore the design of HDFS and evaluate its suitability for use as a general purpose distributed file system in HPC environments.

4.7.2 Architecture

HDFS is a distributed file system written entirely in Java, which allows it to be used in both Windows and Linux environments. HDFS is designed for high-throughput batch processing instead of low-latency interactive user access. HDFS is tuned for applications with large data sets and files that are gigabytes to terabytes in size. Applications that use HDFS should have a write-once-read-many access pattern to achieve high-throughput levels. HDFS does not support writing to existing files, though in recent releases support for appending to a file has been included. The write-once-read-many access model simplifies data coherency issues inherent in data replications schemes [15].

The two primary components that make up HDFS are the *NameNode* and *DataNode*. The *NameNode* maintains the metadata for all the files and directories in the storage system. It addition, the *NameNode* tracks the number of replicas for each data block and ensures that enough replicas are present in the storage system to satisfy the replication policy requirements. The factor limiting the total number of files that a HDFS file system can store is the amount of memory in the *NameNode*. This is because the *NameNode* maintains an image of the entire file system namespace and file block-map in memory [15]. The *DataNodes* store the data blocks and checksum data for each file in the storage system. When a new data block is created, a checksum value is calculated and stored along with the data block on the *DataNode*. The checksum value is used by the client to verify the data block's integrity and prevents the client from using data that has been silently corrupted [15, 4, 6].

4.7.3 Distribution techniques

HDFS uses a combination of file striping and replication to achieve both high data throughput and fault tolerance. When a client stores a file to HDFS, client side caching is used to minimize the impact of network latency. Files are initially written to a temporary location on the client. When the amount of data reaches the pre-defined block size (64MB by default), a checksum is calculated for the block and then the block is written to one of the *DataNodes*. While the block is being transferred to the *DataNode* from the client, the application has already begun writing another block to the local client size cache. Unlike GlusterFS, which requires the client to write multiple copies of a file when replication is enabled, HDFS uses a technique called replication pipelining. Once the first block has been successfully copied to a *DataNode*, the *DataNode* will begin transferring the block to another *DataNode* within the HDFS file system. The set of *DataNodes* where the blocks for a file (as well as the block replicas) should be written is stored in a block-map generated by the *NameNode* and used by the client to assign data blocks to *DataNodes*. This process will continue until the minimum number of replicas are created [15].

In addition to replication pipelining, HDFS introduces a couple of other unique features to improve throughput and fault tolerance. First is an awareness of data locality. HDFS is designed to be used for extremely large-scale deployments with data distributed across servers in multiple racks and possibly across multiple data centers. This ensures that if a rack of servers or even an entire data center is destroyed, the data will remain available. Another benefit of this knowledge of data locality is that it leverages both the fact that any replica can be used for read operations and that there is typically more bandwidth available within a rack than between racks. As a result, client file operations can be optimized to use data blocks from within the same rack as the client instead of pulling from adjacent racks or possibly from a remote data center.

The other unique feature of HDFS that improves throughput and fault tolerance is that the *NameNode* actively monitors the replicas of each data block. In the event that a hard disk or even an entire *DataNode* fails, the *NameNode* will issue commands to *DataNodes* containing replicas of the lost blocks causing them to begin replicating with other *DataNodes*. This results in a restoration of the desired level of fault tolerance and maintains the appropriate number of replicas that can be used to distribute read operations across the file system [15].

4.7.4 Feature summary

Table 4.7: Summary of HDFS design features

Software Interfaces	Native Java API, C wrapper for Java HDFS API, WebDAV, and FUSE-HDFS [15, 28]
File Distribution Techniques	Data Block Replication [15]
Network Interconnects	TCP/IP over GigE, 10GigE, or Infiniband

4.7.5 Performance characteristics

HDFS has been designed specifically for use with Hadoop applications that use the native Java API to access the file system; however, there has recently been interest in the use of HDFS as a general purpose storage system for HPC environments. There are a number of projects working to develop a VFS interface for Linux. The most promising and actively developed is fuse-dfs, included in the contrib folder of the hadoop source code. Marissa Hollingsworth of Boise State University has conducted extensive testing comparing the performance of applications that access HDFS via the fuse-dfs interface versus the native Java API. Hollingsworth demonstrates up to a 50% slowdown in the application using the fuse-dfs interface when compared to the native Java API [24]. These results are a bit discouraging, but in the interest of being thorough, the basic transfer, block-range throughput, and client-scalability tests are used to evaluate the suitability of HDFS for use as a general purpose storage system in HPC environments.

HDFS test environment configuration

Two different storage systems were used to evaluate the performance characteristics of HDFS. The basic file and block-range file transfer tests were performed using the Atlantis research cluster. Table 4.8 shows the configuration of HDFS on Atlantis. HDFS was tested more than a year after the other file systems presented in this chapter. During that time, Atlantis was removed from the HPC lab at Boise State University, and the client-scalability testing had to be performed using three nodes from the Gene Sequence Information System (GeneSIS) to run HDFS. These nodes have faster processors, more processing cores, more memory, and faster disk subsystems than the storage nodes in Atlantis. As a result, the throughput values may be slightly higher than those for the other distributed file systems presented in this chapter; however, the client-scalability testing still utilizes processing nodes of the Beowulf cluster over a single-channel gigabit Ethernet interconnect. This configuration uses two storage nodes for data, so the expected aggregate throughput is 200MB/sec, similar to that of Lustre. Table 4.9 shows the configuration of HDFS on the Gene Sequence Information System (GeneSIS).

Table 4.8: Summary of HDFS configuration on Atlantis

atlantis01	NameNode
atlantis02	DataNode
atlantis03	DataNode
atlantis00	FUSE-DFS Client

Table 4.9: Summary of HDFS configuration on GeneSIS

gen01	NameNode
gen02	DataNode
gen03	DataNode
Beowulf Cluster	FUSE-DFS Client

Basic file transfer

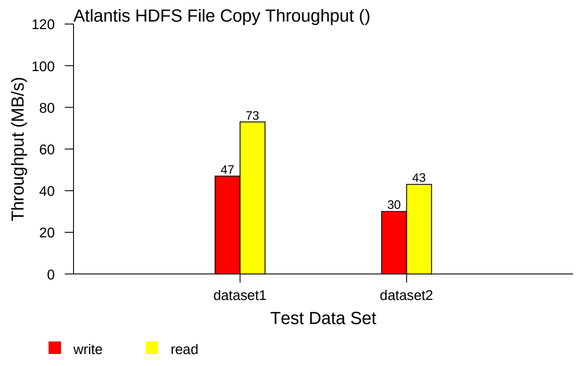


Figure 4.17: HDFS basic file transfer with 64MB blocks and 2x replication

The throughput results shown in Figure 4.17 for both dataset1 and dataset2 are quite good in comparison to the throughput results for the other file systems that were tested. This demonstrates that the fuse-dfs client is optimized to allow standard tools such as `cp` to have good performance without modification. This is a sharp contrast to the results for PVFS shown in Figure 4.5. It also shows that the HDFS *NameNode* is able to deal with large numbers of files without significant impact.

Unfortunately, the first attempts to run the basic file transfer test on HDFS failed with the errors similar to the following:

```
/bin/cp: preserving times for ‘...’: Input/output error
```

The basic file transfer test uses `cp -a` to transfer both dataset1 and dataset2. The error appears to be related to preserving time stamp information for each directory. Replacing `cp -a` with `cp -r` resolved the issue.

Another issue occurred when attempting to overwrite existing files on the HDFS file system.

```
cp: overwrite ‘...’? y
cp: writing ‘...’: Input/output error
```

This issue appears to be related to the previously mentioned fact that HDFS does not allow the modification of files once they have been written to the file system. The issue was resolved by first removing the existing files, then running the file transfer.

Block-range file transfer

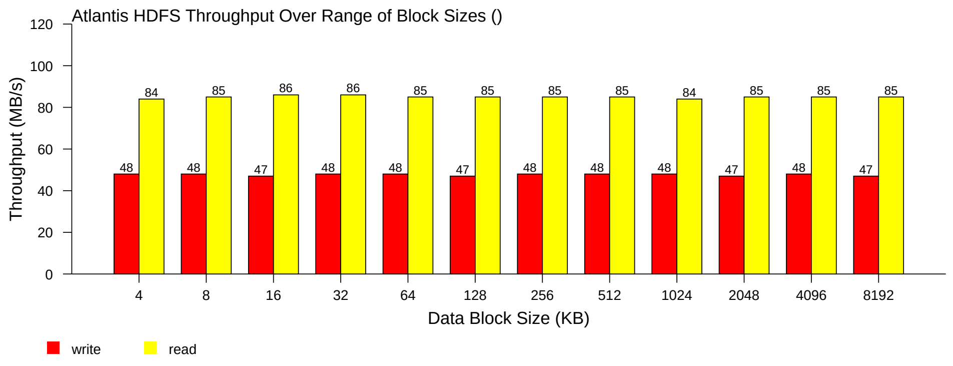


Figure 4.18: HDFS block-range file transfer with 64MB blocks and 2x replication

The throughput results for the block-range file transfer test shown in Figure 4.18 demonstrate that HDFS performs consistently for block I/O operations across a wide range of block sizes. These results also show that in spite of the innovative replication pipelining implemented in HDFS, there is still a significant performance penalty for write operations. This is clearly why the HDFS documentation emphasizes the write-once-read-many access pattern.

Client scalability

The client scalability test failed to run with HDFS. The `fs-perf` tool uses MPI and ROMIO to perform the synchronized read/write operations; however, all attempts to run `fs-perf` on HDFS result in errors similar to the following:

```
Node 10: Read Error at 0.000000 secs: 212399628
```

This is not the result of using FUSE since the GlusterFS testing also used a FUSE interface. It appears to be either an issue with the implementation of the fuse-dfs client, or an issue with ROMIO attempting to perform I/O operations in a manner that is not compatible with HDFS.

4.8 Conclusion

Each of the parallel distributed file systems discussed in this chapter has specific target applications where they can yield the best performance results. PVFS is a good choice for HPC environments with applications that perform large sequential I/O operations and perform a significant amount of parallel I/O using either the kernel VFS interface or the MPI ROMIO interface. The demonstrated client scalability as well as developer documentation indicates that PVFS is suitable for HPC installations with hundreds or even thousands of clients. The fact that the storage processes operate entirely in user space makes PVFS a good choice for environments that use custom kernels or Linux distributions that are not supported by Lustre.

In general, Lustre is a better choice than PVFS because it not only excels in the same areas as PVFS (large sequential I/O), but the read and write caching allow for extremely good performance for small sequential I/O operations as well. In addition, Lustre provides well-documented tools for allowing the end-user to adjust file distribution techniques on the fly. However, Lustre takes a bit of a heavy-handed approach requiring custom kernels on the storage nodes, which will likely be an issue if the cluster is not running a supported distribution. Another factor to consider with Lustre is that it requires dedicated block devices for the backend MDT and OSTs, which can make it difficult to integrate Lustre into existing storage systems. Lustre has also been demonstrated to scale extremely well, both in the number of clients it can support as well as the aggregate throughput and storage capacity it is able to provide. One of the keys to achieving good performance with Lustre is to use simple file distribution when dealing with many small files and use file striping for large files.

The file striping capabilities of GlusterFS cannot compete with PVFS or Lustre in environments performing large sequential I/O operations. The lack of a native ROMIO interface limits MPI based applications that perform complex I/O access patterns. However, the simple file distribution of GlusterFS had the best throughput results of either file system when working with large numbers of small files. Except for HDFS, GlusterFS is the only one of the tested distributed file systems to provide file redundancy at the file system level. These capabilities of GlusterFS make it a good fit for HPC environments with large numbers of small files, especially when the majority of I/O operations are read operations.

HDFS has a lot of potential for use in HPC environments, both from a reliability and a performance standpoint. Unfortunately, there are a number of limitations in the fuse-dfs client that make HDFS unsuitable for use as a general purpose file system. These limitations include relatively slow write performance compared to other parallel distributed file systems, inconsistent behavior with user permissions, and flakey behavior when removing directories and checking disk space usage, not to mention the previously discussed I/O errors that were encountered while performing relatively standard file operations. The fuse-dfs client is an excellent tool for moving data into and out of HDFS and it may be suitable for specific applications, but it is not suitable for use as a general purpose file system.

The distributed file systems examined in this chapter are designed for applications that perform sequential I/O operations; however, there are many other types of distributed file systems available, both open source and proprietary. Each has its own design goals and performance characteristics. Some are targeted for specific development platforms such as HDFS while others focus on optimizing sequential I/O access such as Lustre or PVFS. Before selecting a distributed file system for a storage system, evaluate the performance and scalability capabilities of the various file systems based upon type of I/O operations it will be expected to satisfy in the environment in which it is deployed.

CHAPTER 5

IDENTIFYING APPLICATION PERFORMANCE CONSTRAINTS USING I/O PROFILES

5.1 Introduction

This chapter presents a technique for identifying application performance limitations by generating a profile of an application's I/O characteristics and evaluating it in the context of an established I/O performance baseline for the test environment. This technique can be used by software developers interested in optimizing an application's I/O throughput. It can also be used by system engineers seeking to understand an application's I/O capabilities so that informed decisions can be made in the design of processing and storage systems.

Identifying the performance limitations of an application using I/O profiling consists of several steps. The first is to establish a baseline for the I/O performance of the system running the application. Next, use system monitoring tools to capture system performance statistics while the application is running. It is helpful to use a graphing utility to plot both the baseline and application performance data to visualize the problem. Finally, compare the application performance data against the established baseline data. It should be possible to identify the factors limiting application performance. Application performance limitations are typically the result of one of the following constraints: CPU, memory, network I/O, or block I/O. Application constraints that are not detectable using this technique include race-conditions and dead-locks as well as embedded sleep statements and blocking for user input.

Some application performance issues can be quickly diagnosed using real-time system monitoring tools while others may require a more in-depth analysis. These approaches are not mutually exclusive, and this chapter demonstrates the use of both in evaluating an application's I/O characteristics. After reading this chapter, the reader should be able to:

1. Generate an I/O performance baseline for a system
2. Capture performance statistics to generate an I/O profile for an application
3. Evaluate an application's I/O profile and identify performance constraints

Once a performance constraint has been identified, the actions required to improve application performance depend upon the type of constraint. For instance, the performance of a CPU bound application can be improved by running a code profiler to identify the processing intensive code and refactoring the code to make it more efficient, by leveraging multiple processor cores to perform calculations in parallel, or by purchasing a faster processor. The performance of a memory bound application can be improved by purchasing additional memory, by refactoring code to use more efficient data structures, or by dividing the problem into several smaller problems that can each be solved independently requiring a smaller memory footprint instead of attempting to solve a single large problem.

Improving the performance of an I/O bound application can be a difficult nut to crack. The purchase of a faster network interconnect or faster disk subsystems may help improve performance in some cases, but this is not guaranteed. As with CPU bound and memory bound applications, there are both hardware and software based solutions to I/O bound applications, but in the case of I/O bound applications it is often necessary to develop a hybrid approach.

Section 5.2 describes how to generate a baseline for I/O throughput and I/O operations per second (IOPS) that a system is able to sustain. Section 5.3 describes how to determine whether an application is fully utilizing the I/O potential of the system by monitoring the system performance counters while the I/O bound application is running. *If the I/O performance of the application is reaching the baseline levels, then there is strong evidence that upgrading hardware will increase application performance. However, if the application I/O performance falls drastically short of the baseline, it is then necessary to investigate why. This investigation will require code profiling and most likely refactoring to make I/O operations more efficient.* Section 5.4 walks the reader through this process using a demonstration application called *seqprocessor*.

Section 5.6 gives some general recommendations on how to tune an application's I/O operations. The examples in this chapter focus on applications that perform sequential I/O operations but Section 5.7 describes how the same technique can be applied to applications that perform random I/O operations. Section 5.8 discusses how the I/O profiling technique described in this chapter can be extended to parallel applications.

5.2 Establish an I/O performance baseline

5.2.1 Set up the environment

To profile the I/O performance of an application, set up a dedicated workstation to use for running the application. The workstation should have sufficient CPU power and memory capacity to comfortably run the application as well as a disk subsystem dedicated for application I/O. There should also be a minimal number of services enabled to minimize CPU and memory contention.

The reasons for the dedicated disk subsystem are twofold. First, to ensure the best possible I/O performance for the application, its I/O requests should not be interleaved (queued, scheduled) along with the I/O requests of other system processes. This is to minimize seek time for sustained sequential I/O operations since I/O operations from other applications can move the disk heads out of position. It also prevents application I/O requests from getting pushed lower in the queue due to higher priority I/O operations from system processes. The second reason for using a dedicated disk subsystem for application I/O is that the kernel block device counters measure throughput and IOPS at the block device level. It is difficult to separate I/O requests for one application from those of another if they both are accessing the same block device. Having a dedicated block device simplifies application performance monitoring.

When establishing an I/O performance baseline, it is a good idea to document a few details about the test system. This is useful for preserving key details about the environment to aid with comparisons when profiling applications on other systems. Appendix C provides a worksheet for documenting the test environment configuration. The system information can be retrieved from `/proc/cpuinfo`, the `free` command, the system BIOS, and the component manufacturer's websites. The information regarding the dedicated disk subsystem used by the application can be found using the `mount` command, `/proc/mdstat`, `pvdisplay`, and `/etc/fstab`.

The benchmark and application profiling results presented in the following sections were generated using two of the nodes from the Atlantis research cluster. The configuration details for Atlantis can be found in Appendix D.

5.2.2 Benchmark the environment

Before attempting to capture an application's I/O profile, use standard benchmarking tools to identify the I/O performance capabilities of the test system. This information will be used as a baseline to gauge application I/O performance. Table 5.1 provides a list of the benchmarking tools discussed in this section.

Many experts consider system benchmarks to be of little value for gauging how an application will perform on a given system [29]. The issue with I/O benchmarks is that they provide results for a specific type of I/O operation. If the application performs I/O operations in a similar manner, then it is possible that the results of the benchmark will reflect actual application performance. However, more often the type of I/O operations performed by the benchmark do not in any way resemble how the application will run; therefore, the benchmark results are meaningless.

For benchmark results to be of value, acquire information about how the benchmark is performed. This information should include the method for performing I/O operations (`fgetc`, `fread`, memory mapped I/O), the file size used for testing, the block size used for the I/O operations, the access pattern (sequential, random, backward, strided), the type of processor and L2 cache size, and the amount of RAM. If the I/O method and access pattern of the benchmark do not match the

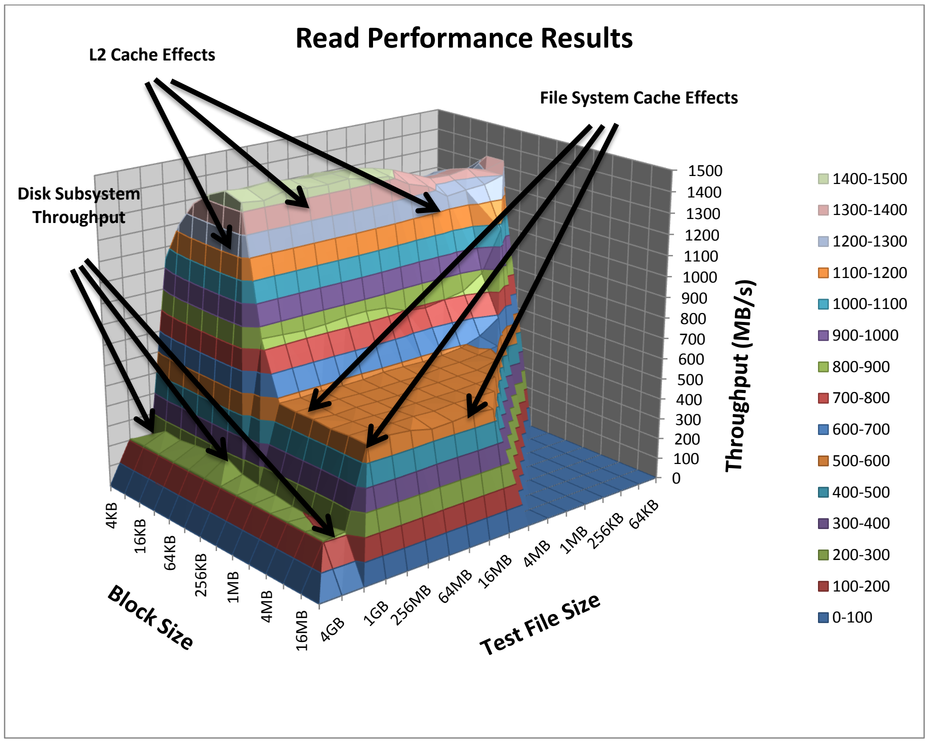
Table 5.1: Tools for benchmarking disk subsystems and network interconnects

Tool Name	Measures	Description
`bonnie++`	block I/O	Bonnie++ performs tests that report throughput and CPU utilization for character and block based read, write, and rewrite operations. Bonnie++ also tests metadata performance by measuring the amount of time required to create, stat, and delete very large numbers of files [10].
`iozone`	block I/O	Iozone is a file system benchmark utility with extensive testing and reporting capabilities. The tests use various mechanisms for performing both sequential and random I/O operations. In addition to outputting throughput results, `iozone` can be configured to output IOPS and latency results as well [8].
NetPIPE	network I/O	NetPIPE measures throughput and latency for a range of message sizes and supports a number of network communication protocols and message passing libraries including TCP, Myrinet, Infiniband, MPI, PVM and SHMEM [7].

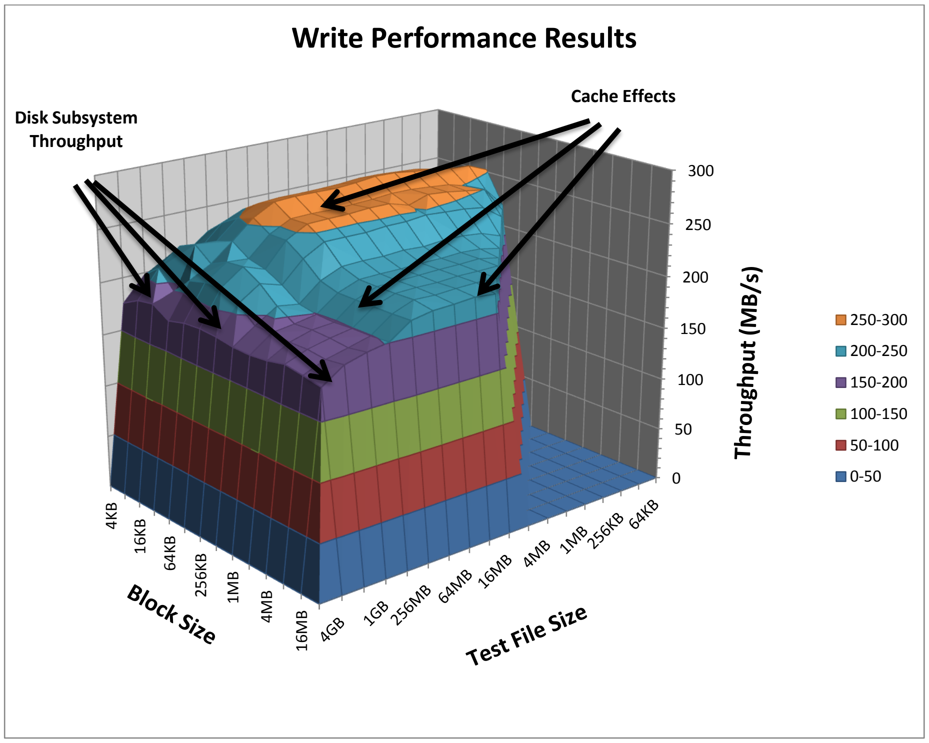
application, the benchmark results will not accurately reflect the expected application performance. If the block size is smaller than the L2 or L3 cache size or the file size is smaller than the amount of system memory (RAM), the read throughput of the benchmark will be significantly higher than what is reasonable to expect for a typical application.

Iozone

Figures 5.1 and 5.2 were generated by running `iozone` upon a dedicated disk subsystem (md0) on *atlantis01*. These figures are labelled to indicate both the cache effects as well as the “real” throughput that the disk subsystem can sustain. Notice how the throughput for block sizes of 4KB up to 128KB in Figure 5.1a is well over 1GB/s up



(a) Sequential Read



(b) Sequential Write

Figure 5.1: Iozone test using 4KB to 16MB block sizes on files up to 4GB

until the point that the file size is no longer able to fit in main memory. This is an artificial result of the benchmarking process. The benchmark first writes the file to disk and in the process the file is cached in the L2 processor cache as well as the main system memory. For file sizes less than the total amount of system memory, no data has to be retrieved from the disks. This is the reason for extremely high throughput levels and why these results do not reflect actual application performance. This would only be a valid representation of actual application performance if the application first wrote the file to disk, then read it back in. Most applications, however, read data from the disk, do some work, then write data back to the disk. As a result, they are not able to reach the 1GB/s+ throughput results.

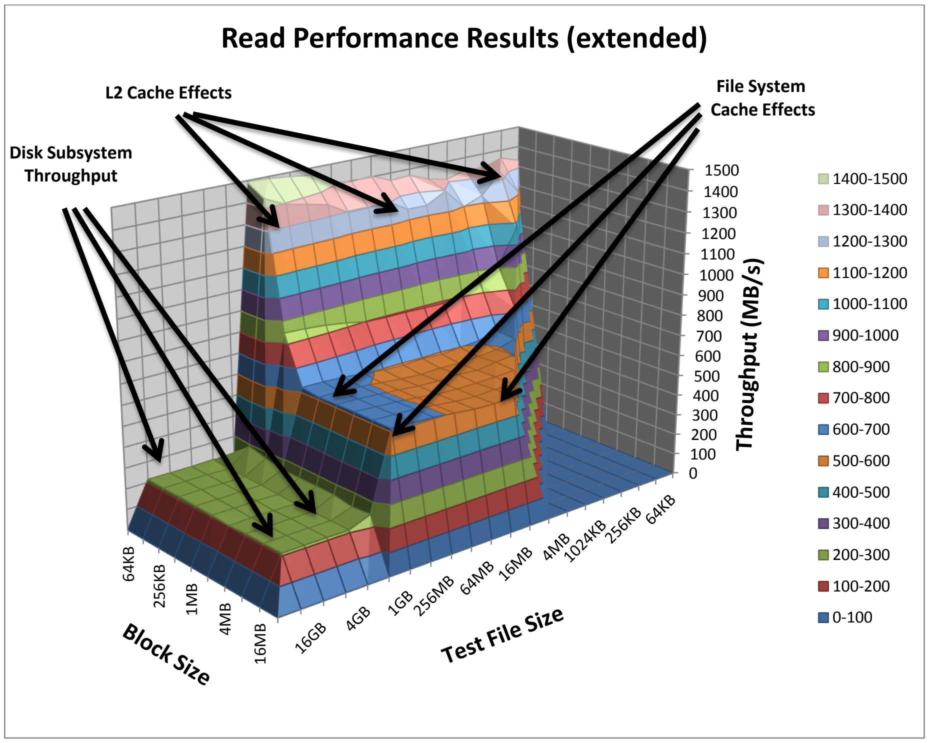
The following `iozone` options were used to generate the results shown in Figure 5.1:

```
iozone -az -g 4G -i 0 -i 1 -i 2
```

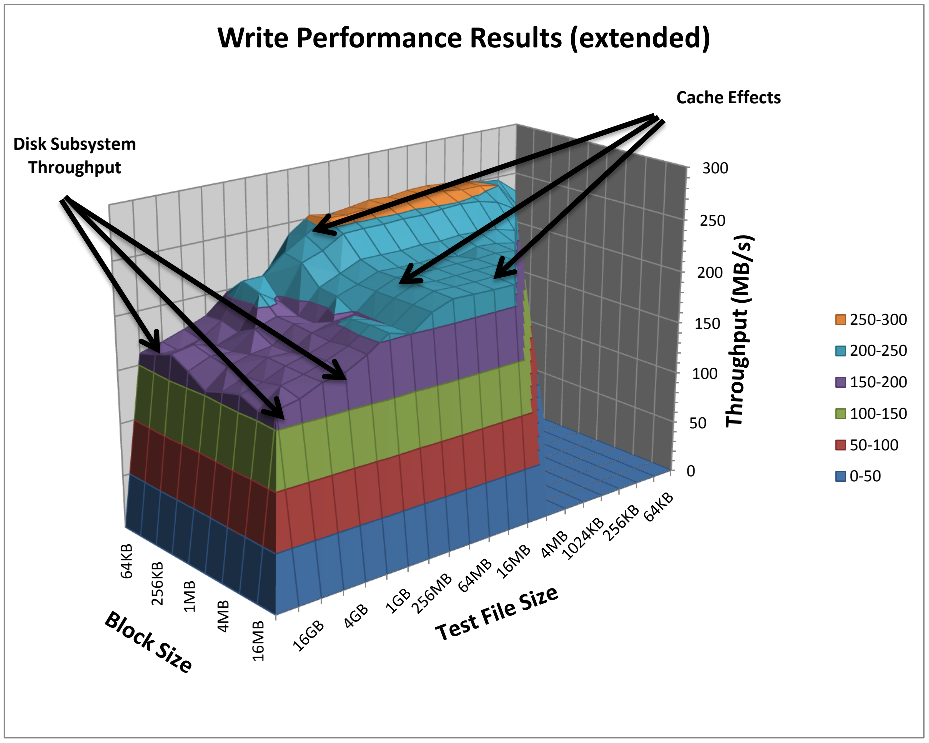
To more accurately reflect expected I/O throughput for an application, the benchmark should use a file size that is at least two times the total amount of system memory. Figure 5.2 also shows read and write throughput results, but this time extended to a file size of 32GB. Notice how the throughput drops to just over 200MB/s once the file size increases to more than 2GB. The reason for this is that the larger file sizes negate the caching effects of the processor and operating system and expose the true throughput capabilities of the underlying hardware.

The following `iozone` options were used to generate the results shown in Figure 5.2:

```
iozone -a -g 32G -i 0 -i 1 -i 2
```



(a) Sequential Read



(b) Sequential Write

Figure 5.2: Iozone test using 64KB to 16MB block sizes on files up to 32GB

Bonnie++

Tables 5.2 and 5.3 show the output generated by `bonnie++` running on the same dedicated disk subsystem used for the `iozone` testing. At first glance, this output can appear very cryptic, but once a person becomes familiar with the format, this "slice" of information can tell quite a lot. First, this disk subsystem can sustain 147MB/s for block write and 193MB/s for block read operations. These results are based upon an 8KB block size and a file size of 4GB (twice the total system memory) [10]. Compare these block results to the character write and read throughput values of 33MB/s and 39MB/s, respectively. The metadata operations for creating, reading, and deleting files can be informative as well. These results are obtained by creating, reading, and deleting 500,000 files on an ext3 file system with journaling enabled. It is interesting to note that it is four to five times faster to create a file than it is to delete it.

Table 5.2: Bonnie++ throughput results for md0 on *atlantis01*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
33	99	171	99	72	32	39	97	183	27

Table 5.3: Bonnie++ IOPS results for md0 on *atlantis01*

Bonnie++ Metadata Performance					
Sequential (ops/sec)			Random (ops/sec)		
create	read	delete	create	read	delete
11170	194149	2794	9483	235479	1978

Due to the relatively long runtime of `iozone` and the complexity of output, `bonnie++` is a better choice for system engineers who are wanting to quickly verify I/O performance of a given disk subsystem. Cases where `bonnie++` is a good choice are the investigation of disk/controller performance issues as well as comparing and contrasting the performance of various RAID configurations. Iozone is a good choice when a more detailed investigation of I/O block size and access pattern is required. An example would be developing an application to leverage the performance characteristics of a given disk subsystem, or tuning a disk subsystem for a specific type of I/O operation.

NetPIPE

Although the applications demonstrated in the following sections will only be performing local I/O operations, it is valuable to generate a baseline of the network interconnect throughput and latency. NetPIPE is an excellent tool for accomplishing this. Figure 5.3 shows the network throughput between two nodes in the Atlantis research cluster using gigabit Ethernet. Notice the poor throughput results for packets with small block sizes and how the throughput increases as the block size increases. This baseline profile is important to keep in mind when evaluating applications that perform network I/O or block I/O over a network file system.

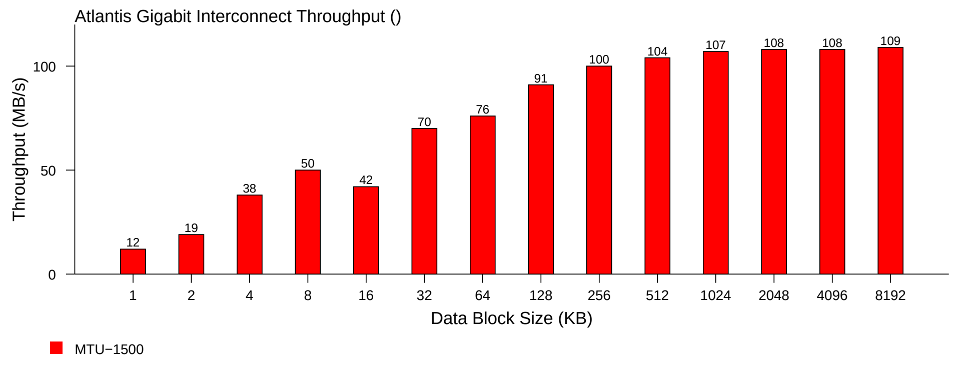


Figure 5.3: NetPIPE throughput results over GigE link with the default MTU of 1500 bytes

5.3 Generate an I/O profile for an application

5.3.1 Tools of the trade

The tools described in Table 5.4 fall into two basic categories: real-time monitoring and background monitoring. The real-time tools are designed to be run while the application in question is running and display the performance results from the various system counters in real-time. Each tool targets a specific subset of the available system counters, I/O, memory, or CPU usage. The background tools are designed to capture data from any or all of the system counters in synchronized snapshots. They can be run as system services with the results written out to a file that can later be manipulated in a spreadsheet. The tools in Table 5.4 are all available in CentOS in either the base or EPEL yum repositories.

Table 5.4: Tools for monitoring system utilization

Tool Name	Usage	Monitors
`top`	real-time	system load, CPU and memory utilization, as well as memory and CPU usage by individual processes [54]
`iostat`	real-time	CPU and block I/O [21]
`mpstat`	real-time	CPU [21]
`pidstat`	real-time	process specific CPU, memory, and block I/O statistics [21]
`vmstat`	real-time	processes, memory, paging, block IO, traps, and CPU [53]
`bwm-ng`	real-time	network and block I/O [23]
`sar`	background	all the data reported by `iostat`, `mpstat`, and `pidstat` as well as many other counters [21]
`collectl`	real-time, background, and remote	extensive system real-time and background system monitoring capabilities which encompass the functionality of all the tools mentioned above, but also includes the ability to directly monitor Infiniband and Lustre subsystems [44]

For this discussion, application lifetime refers to the period of time beginning when an application is executed and ending when the application terminates. At any point during an application's lifetime, real-time system monitoring tools can be used to monitor current I/O throughput, IOPS, CPU, and memory usage. Unfortunately, these techniques only give a limited view of an application's I/O characteristics. This is especially true for applications with complex I/O patterns with periods of heavy read or write activity and periods of heavy CPU usage. To accurately characterize the I/O profile of these more complex applications requires capturing snapshots of the kernel level performance counters in short intervals, evenly distributed throughout the life of the application.

One issue with capturing data at frequent intervals comes from the fact that many I/O and processing operations occur in bursts. For the I/O profiling charts presented in the following sections, `collectl` was configured to capture data in one-second intervals. Plotting this data directly yielded nearly unreadable results with huge spikes in both CPU utilization and I/O throughput followed by equally drastic drops. To resolve this issue, the collected data was massaged using a running average over ten sample periods. This averaging had a smoothing effect on the sample data and produced readable charts.

5.3.2 Profile of an I/O bound application

Figures 5.4 and 5.6 show the I/O profiles for `dd` read and write operations, respectively. These were chosen as simple examples to demonstrate what the profile of an I/O bound application looks like. Figure 5.4 was generated using the output of `collectl`, collected during a `dd` operation that read a 32GB file from a dedicated disk subsystem on *atlantis01* and wrote it to `/dev/null`. Notice how the read I/O throughput is consistently in the 175MB/s to 190MB/s range. These results are consistent with the baseline `iozone` results shown in Figures 5.1 and 5.2 as well as the `bonnie++` results shown in Table 5.2.

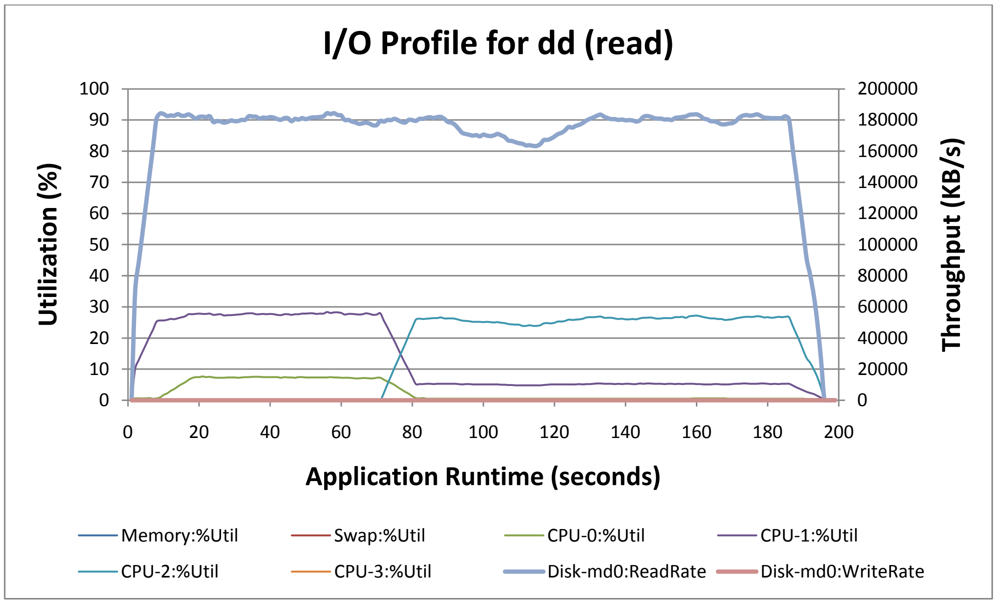


Figure 5.4: I/O profile of `dd` reading a 32GB file using 1MB blocks

Figure 5.5 is the output from the `top` command, captured during the previously mentioned `dd` read operation. Notice now the CPU utilization for `dd` is only 43.5% and that it appears that the bulk of the processing working is being handled by CPU-2 executing system calls with an I/O wait time of 54.2%. Another point worth mentioning is that an application is not bound to a single processing core for its entire lifetime. Figure 5.4 shows that the processing for the `dd` command was effectively migrated from CPU-1 to CPU-2 just before the eighty-second mark.

Figure 5.6 was generated using the output of `collectl`, collected during a `dd` operation that generated a 32GB file read from `/dev/zero` and written to a dedicated disk subsystem on *atlantis01*. Here again, the write rate shown for the dedicated disk

```
top - 12:37:59 up 5 days, 55 min,  3 users,  load average: 0.81, 0.86, 1.51
Tasks: 136 total,   1 running, 135 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.3%us,  6.0%sy,  0.0%ni, 93.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us, 42.2%sy,  0.0%ni,  0.0%id, 54.2%wa,  0.3%hi,  3.3%si,  0.0%st
Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2075608k total,  2023076k used,    52532k free,     2444k buffers
Swap:  2031608k total,       68k used,  2031540k free,  1936636k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19441 root      18   0  4908 1568  472 D 43.5  0.1   0:16.87 dd
  173 root      10  -5     0    0    0 S  5.6  0.0   7:50.09 kswapd0
```

Figure 5.5: Output from the `top` command while the `dd`(read) command was running

subsystem is consistent with the previously generated baseline. Even with the running average of the sample data, curves for CPU utilization and disk subsystem throughput are not as smooth for the write operation as for the read operation. Also, there is a lot more overall CPU activity.

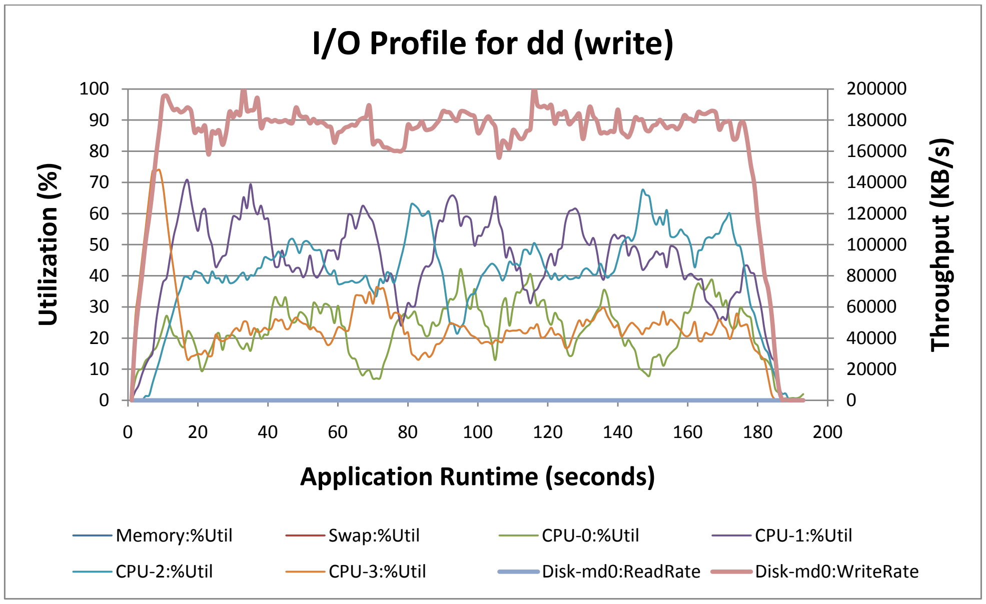


Figure 5.6: I/O profile of dd writing a 32GB file using 1MB blocks

```
top - 12:49:01 up 5 days,  1:07,  3 users,  load average: 2.07, 1.53, 1.51
Tasks: 136 total,   2 running, 134 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us, 37.5%sy,  0.0%ni, 51.5%id, 11.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us, 45.7%sy,  0.0%ni, 48.7%id,  5.6%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us, 30.3%sy,  0.0%ni, 23.7%id, 38.0%wa,  0.0%hi,  8.0%si,  0.0%st
Cpu3  :  0.0%us, 33.7%sy,  0.0%ni, 47.7%id, 18.7%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2075608k total,  2023804k used,    51804k free,     2296k buffers
Swap:  2031608k total,       68k used,  2031540k free,  1898060k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19512 root      20   0  4908 1572 1496 R 91.1  0.1   0:56.92 dd
19312 root      15   0     0    0    0 D 34.2  0.0   0:54.28 pdflush
  173 root      10  -5     0    0    0 S 15.0  0.0   8:29.16 kswapd0
 1465 root      10  -5     0    0    0 D  8.3  0.0  22:19.80 kjournald
```

Figure 5.7: Output from the top command while the dd(write) command was running

Figure 5.7 shows the output from `top` during the `dd` write operation. Notice that even though the `dd` process has 91.1% CPU utilization, no single processing core is spending more than 50% of its time actively processing, while all the cores are spending time waiting for I/O. An observation that deserves further explanation is the amount of processing time utilized by the `pdflush`, `kswapd`, and `kjournald` processes.

When data is written to disk on a Linux system, it is first stored in the page cache. Pages that are stored in the cache, but not yet committed to disk, are considered dirty. Once a certain threshold is reached, `pdflush` is called to flush the dirty pages to disk [45]. The output from `top` shows that during the `dd` write operation, the total size of the page cache was more than 1.8GB of memory. To allow the transfer of a 32GB file, the `kswapd` process continually removes older (non-dirty) pages from the cache so that space will be available to cache new pages. Finally, as the file data is written to the ext3 file system the journal must be updated to ensure that it will be able to quickly recover from errors. This is the responsibility of `kjournald`.

5.4 Case Study: *seqprocessor*

To demonstrate the I/O profiling technique, this section steps through the process of diagnosing and improving the I/O performance of an application. The application used for this demonstration is called *seqprocessor* and it was developed to simulate the I/O characteristics of several bioinformatics research applications in our HPC lab at Boise State University. *Seqprocessor* reads DNA sequence data from a text file, performs some statistical processing of the data, then writes the results back to disk.

The sequence data files range in size from a few kilobytes up to tens of gigabytes. Given the amount of data that must be processed, it is important to optimize the throughput as much as possible. For the purposes of this demonstration, the processing portion of the code has been removed to ensure that any bottlenecks that are identified are directly related to the I/O portions of the applications. Therefore, the throughput results will be best-case results for the *seqprocessor* application. The source code for the three different versions discussed in this section is provided in Appendix E.

To generate the I/O profile, `collectl` is first started in a separate terminal on the test system, *atlantis01*. The dedicated disk subsystem is `/dev/md0`, so the following command will be used to collect the CPU, memory, and disk statistics:

```
collectl -smDC --dskfilt md0 -P > seqprocessor-1.profile
```

With collectl running in the background, start *seqprocessor* in another terminal. The following command uses *seqprocessor* to process a 5GB text file containing sequence data from the est_human dataset:

```
./seqprocessor-1 /mnt/data_disk1/est_human /mnt/data_disk1/est_human.processed
```

```
top - 11:23:40 up 5 days, 23:41,  3 users,  load average: 1.18, 0.43, 0.14
Tasks: 142 total,   2 running, 140 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.7%us,  0.3%sy,  0.0%ni, 98.7%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 86.8%us, 11.6%sy,  0.0%ni,  0.0%id,  1.3%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu2  :  0.3%us,  1.7%sy,  0.0%ni, 98.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.3%us,  2.0%sy,  0.0%ni, 86.1%id, 11.6%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2075608k total,  2023312k used,    52296k free,    82220k buffers
Swap:  2031608k total,       68k used,  2031540k free,  1821820k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
23861 root      25   0  1660  356  292 D 99.1  0.0   0:41.70 seqprocessor-1
  173 root      10  -5     0    0    0 R  1.7  0.0   8:46.97 kswapd0
 1465 root      10  -5     0    0    0 D  1.3  0.0  22:28.36 kjournald
23823 root      15   0 15196  11m 1928 S  1.0  0.5   0:00.85 collectl
19312 root      15   0     0    0    0 S  0.7  0.0   1:11.72 pdflush
```

Figure 5.8: `top` output from `seqprocessor-1` application

The data being processed are strings of the characters A,T,G, and C, so it seems reasonable to use the `fgetc()` and `fputc()` system calls to iterate through these files. The source code for the first version of *seqprocessor* can be found in Listing E.1. Notice that this code performs no processing of the data and essentially performs a file copy operation using character I/O.

While the *seqprocessor* application is running, real-time monitoring tools can be used to get a peek at how the application is performing. Figure 5.8 shows the output from the `top` command. Notice how the CPU utilization for `seqprocessor-1` is 99.1% and that the bulk of the processing is isolated to a single processing core (CPU-1). In a single threaded application, if the sum of %user and %system is greater than 80% with an I/O wait of less than 10% then the application is CPU bound [47].

Figure 5.9 shows the output from the `iostat` command. Observe that both the read and write throughput values are only 20MB/s, just over 60% of the baseline values established with `bonnie++` for character I/O throughput (Table 5.2).

```
Linux 2.6.18-194.3.1.el5_lustre.1.8.4 (atlantis01)      02/23/2011

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          19.83    0.00    5.25    7.45    0.00   67.47

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
md0            6040.40     20142.40     20151.20     201424     201512
```

Figure 5.9: `iostat` output from `seqprocessor-1` application

Once the *seqprocessor* application has finished, `collectl` can be stopped and the results used to generate an I/O profile. The output for `collectl` is quite verbose and requires some manipulation with `awk` and/or a spreadsheet to isolate the desired data points. Figure 5.10 shows the I/O profile for version 1 of the *seqprocessor* application. From this profile it is easy to see that the application is CPU bound, and even though `iostat` showed a throughput of 20MB/s at one point during the application's lifetime, the sustained throughput is closer to 16MB/s. At first it may seem that the application is only achieving 50% of the expected throughput from baseline in Table 5.2; however, a sustained throughput of 16MB/s for both read and write operations makes sense. Let's examine why.

In addition to the baseline throughput values, Table 5.2 includes the CPU utilization required to sustain the associated throughput level. The throughput levels shown for character I/O in Table 5.2 require 99% CPU utilization. The input and output tests performed by `bonnie++` are uni-directional, meaning that it is either reading or writing but not doing both at the same time. In contrast, the *seqprocessor* application is bi-directional, meaning that it reads and writes simultaneously. It is not a great leap to conclude that the Xeon processors in *atlantis01* can sustain an aggregate throughput of 32MB/s when performing character I/O with a single thread. When attempting to perform both read and write character operations simultaneously, it

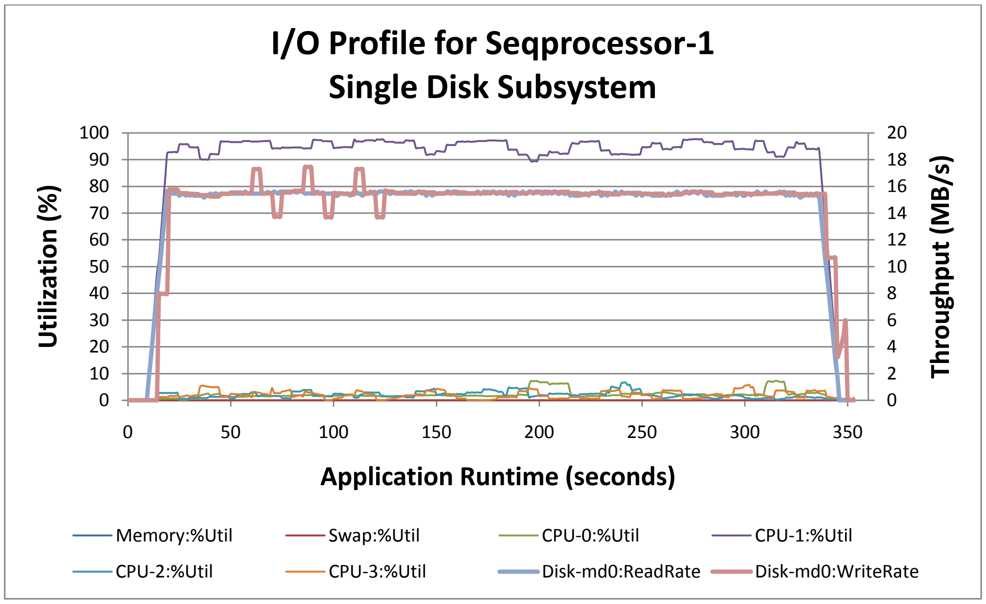


Figure 5.10: I/O profile of `seqprocessor-1` with a single disk subsystem for both read and write operations

makes sense that the *seqprocessor* application would only sustain 16MB/s in each direction.

At this point there are a couple of options to improve application performance. The first is to purchase a faster CPU. The second is to modify the application to perform block I/O operations instead of the character I/O operations. Purchasing a faster CPU may incrementally improve the application performance, but reason suggests that there may be a programatic solution. If this were a commercial, closed-source application the options would be limited; however, in this instance the source code is available so down the rabbit hole we go!

There are a variety of tools that can be used to profile CPU bound applications, and many integrated development environments have this functionality built in. On Linux systems, several popular options include `oprofile`, `gprof`, and `strace`; however, given the system baseline, the application source code, and the I/O profile, inductive reasoning suggests that modifying the application to perform block I/O instead of character I/O will yield a substantial performance increase. To accomplish this, `fread()` and `fwrite()` are used instead of `fgetc()` and `fputc()`. Listing E.2 shows the updated source code.

Figure 5.11 shows the I/O profile for version two of the *seqprocessor* application running with a 1MB I/O buffer. Notice that the throughput levels have increased by more than a factor of four to 70MB/s, while at the same time the CPU utilization has decreased to just over 50%. This example is still performing bi-directional I/O to a single disk subsystem, and a disk subsystem using rotational media is uni-directional by its very nature — meaning that it is unable to read and write data simultaneously. The baseline for reading from and writing back to the same disk subsystem can be found in the rewrite column of Table 5.2, indicating that version two of the *seqprocessor* application is achieving more than 94% of the baseline I/O throughput. These results indicate that this version of *seqprocessor* is I/O bound.

The disk subsystem on *atlantis01* is capable of sustaining more than 170MB/s for uni-directional write operations and more than 180MB/s for uni-directional read operations. In an attempt to leverage the uni-directional nature of the rotation media, the *seqprocessor* application was moved to *atlantis02*, which has two dedicated disk subsystems identically configured to the disk subsystem on *atlantis01*. The baseline performance results for these disk subsystems are provided in Appendix D. Figure 5.12 shows the I/O profile for *seqprocessor* version 2 running on *atlantis02*.

Notice the improved I/O throughput of 115MB/s but still only 60 to 65% of the baseline. It is clear that this application is not I/O bound, but it is not clear what is limiting the performance. These charts are generated using a running average

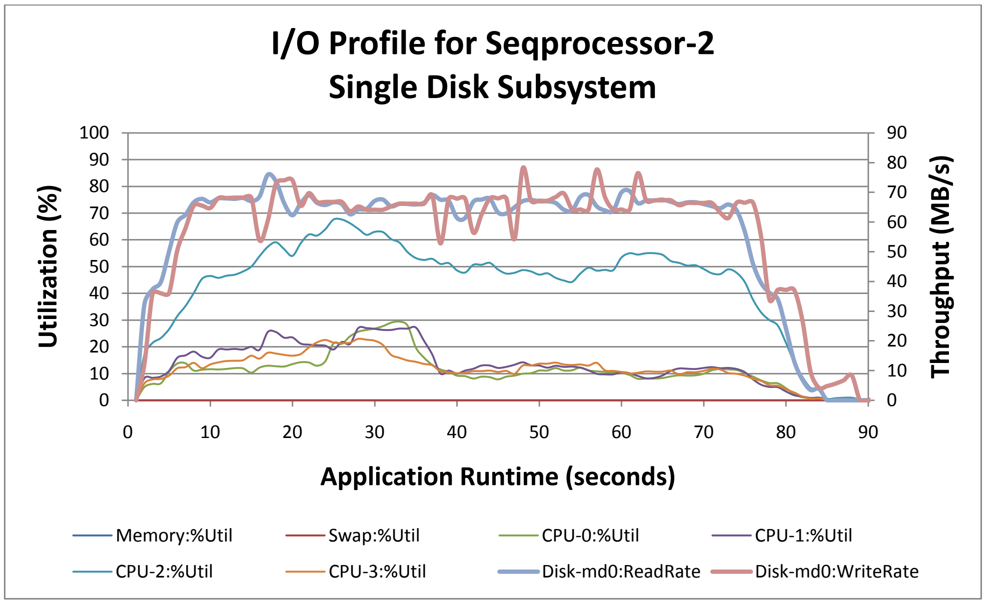


Figure 5.11: I/O profile of `seqprocessor-2` with a single disk subsystem for both read and write operations

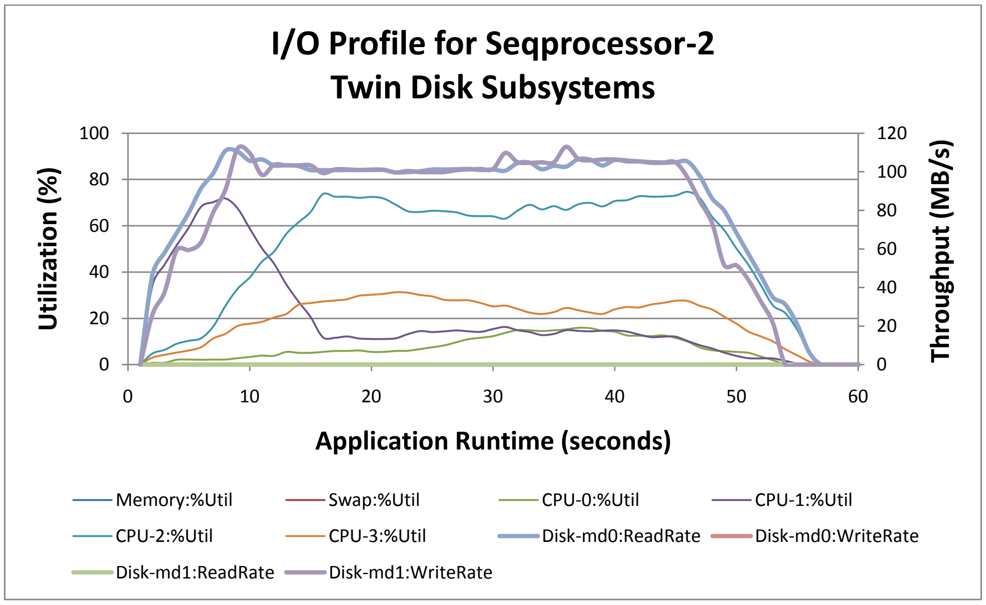


Figure 5.12: I/O profile of `seqprocessor-2` with separate disk subsystems for read and write operations

```
top - 16:53:48 up  2:44,  5 users,  load average: 1.60, 1.59, 1.31
Tasks: 152 total,   2 running, 150 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.3%us,  1.3%sy,  0.0%ni, 95.4%id,  3.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.3%us, 18.7%sy,  0.0%ni, 79.7%id,  1.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.0%us, 21.0%sy,  0.0%ni, 63.0%id,  8.7%wa,  0.3%hi,  7.0%si,  0.0%st
Cpu3  :  0.3%us, 83.4%sy,  0.0%ni,  4.3%id, 10.3%wa,  0.0%hi,  1.7%si,  0.0%st
Mem:   2075608k total,  2022904k used,    52704k free,     3260k buffers
Swap:  2031608k total,       96k used,  2031512k free,  1906328k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4715 platypus  25   0  2688 1400  308 R 89.4  0.1   0:46.60 seqprocessor-2
  174 root      15   0     0    0    0 D 15.9  0.0   1:57.83 pdflush
  175 root      10  -5     0    0    0 S 13.0  0.0   2:27.11 kswapd0
 3754 root      15  -5     0    0    0 D  7.0  0.0   0:45.02 kjournald
  173 root      15   0     0    0    0 D  1.0  0.0   2:02.82 pdflush
```

Figure 5.13: `top` output from `seqprocessor-2` application with separate disk subsystems for read and write operations

over a ten-second period. This effect masks much of the thrashing that is happening behind the scenes with large spikes in CPU utilization. This may explain part of the bottleneck. Figure 5.13 shows the output from `top`. It is clear that CPU-3 is performing a lot of work with only 4.3% idle time. The higher throughput rates increase the amount of work for `pdflush`, `kswapd`, and `kjournald`. It is not as clear as in the first version of *seqprocessor*, but the application appears to be CPU bound once again.

Taking a cue from the producer/consumer problem (commonly used in Operating Systems texts), the *seqprocessor* application was again rewritten to use circular buffering and separate threads for reading and writing. Figure 5.14 is the I/O profile for version three of the *seqprocessor* application. These results were obtained utilizing 128 1MB buffers. Now the throughput rates for the *seqprocessor* application have increased to 135MB/s, nearly 80% of the baseline values for block I/O.

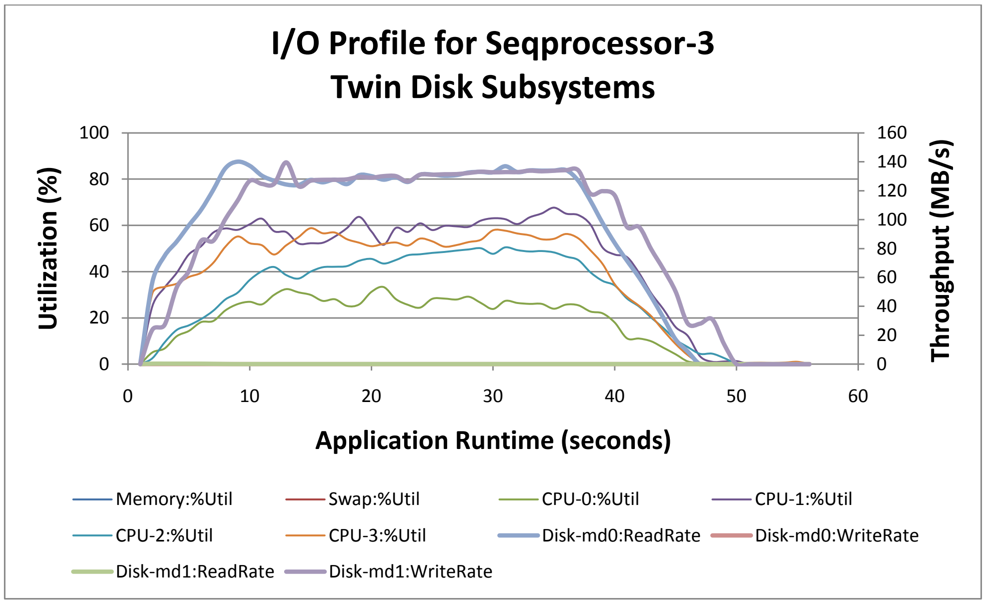


Figure 5.14: I/O profile of `seqprocessor-3` with separate disk subsystems for read and write operations

5.5 Summary of *seqprocessor* performance improvements

Table 5.5 shows the incremental performance improvements with each version of the *seqprocessor* application. The largest single leap in performance was the move from character I/O to block I/O in `seqprocessor-2`. This single software change resulted in a 4x speedup in application runtime and a 466% improvement in I/O throughput. The demonstrated I/O profiling technique allowed for the identification of the performance limiting factors for each version of the *seqprocessor* application. Addressing the limiting factors required a hybrid approach utilizing both hardware and programatic solutions, the combined result of which is 7x speedup in application runtime and an 875% improvement in I/O throughput.

Table 5.5: Summary of *seqprocessor* performance improvements

App Version	Changes	Total Speedup	Throughput Improvement	Limiting Factor
`seqprocessor-1`	-	1.0	100%	CPU
`seqprocessor-2`	block I/O	4.1	466%	Rewrite I/O
`seqprocessor-2`	dedicated read/write disk subsystems	6.5	766%	CPU
`seqprocessor-3`	circular buffering and separate read and write threads	7.0	875%	Write I/O

5.6 Tuning application I/O operations

When developing applications in C, it is possible to adjust the size of the I/O operations in a couple of ways. The first is to simply use `fread()` or `fwrite()` with the desired block size. However, it is not always convenient to work with such large blocks of data. In these cases, `setvbuf()` can be used behind the scenes to optimize the size of the I/O operations.

When `fopen()` is called to open a file, a small block buffer is allocated (4KB). Any calls to `fread()` or `fgetc()` will first attempt to pull the desired data from the buffer. If the data is available, it will be returned to the application. If the data is not available, a system call is made to `read()`, which retrieves the data from disk into the buffer and then returns it to the application. This has the effect of minimizing the number of expensive calls to `read()`. The same process applies to calls to `fwrite()` or `fputc()`. Data is first written to the buffer, then flushed to disk when the buffer is full. The buffers can be manually flushed using `fflush()` [17]. Using `setvbuf()` to increase the size of this I/O buffer can result in significant performance increases. This is especially true in environments where data is transferred over high latency interconnects such as gigabit Ethernet.

Even with buffering, the use of character operations such as `fgetc()` and `fputc()` should be discouraged in applications wishing to achieve high levels of throughput while processing gigabytes of data. To transfer a 4GB file using `fread()` with 1MB blocks would require 4,000 function calls. To transfer the same file with `fgetc()` would require 4,000,000,000 function calls, or 1 million times more work. These number are derived from the fact that a call to `fgetc()` returns a single byte of data. Assuming the data encoding of the character data on the disk also uses one byte to represent a single character, four billion calls to `fgetc()` would need to be made to retrieve four billion bytes of data.

There has been some debate in our lab as to whether or not `gcc` can or will automatically optimize code that makes use of character I/O by either replacing large numbers of calls with a single block I/O call or by moving the calls inline to minimize the stack thrashing involved. The versions of *seprocessor* used to generate the I/O profiles in this chapter were compiled using `gcc` with an optimization level of `-O3`. To answer this question of optimization, the author attempted to use `gprof` to profile the glibc calls made by the various versions of the *seqprocessor* application. Unfortunately, the current version of glibc is no longer provides the `gprof` profiling extensions. In the end, the performance results demonstrated in this chapter clearly show that character I/O is not the way to go for high performance I/O throughput.

5.7 Profiling random I/O application

The examples in this chapter have focused on applications that perform sequential I/O operations; however, the same techniques can be applied to applications that perform random I/O operations. Applications that do not perform large sequential operations do not benefit as much from the read-ahead and write-back caches of modern disk drives and operating systems. Instead, they rely upon low latency I/O and the ability to sustain a high number of I/O operations per second (IOPS).

The `bonnie++` baseline results for IOPS shown in Table 5.3 represent metadata operations for creating, opening, and deleting files. These values depend as much upon the file system (ext3, reiserfs, xfs) as they do upon the underlying disk subsystem. Applications that work with a large number of small files would use these performance measurements as a baseline. Examples would be high traffic proxy/caching servers and mail servers.

In addition to measuring sequential I/O throughput, `iozone` can be used to generate a baseline for random I/O operations with the results displayed as either KB/s or IOPS. These values depend upon the latency of the underlying disk drives and benefit greatly from high RPM disks or SSDs with low seek times. Applications that work with large numbers of small files or that use data structures that store data in non-contiguous blocks within a file can use these results as a baseline. Examples are database servers and applications that use on-disk B-tree data structures.

The procedure for creating an I/O profile for an application performing random I/O is similar to that of an application performing sequential I/O. First, start `collectl` and redirect the output to a file, then start the application. When the application is finished, stop `collectl` and use `awk` or a spreadsheet application to select the desired counters. For applications performing random I/O operations, choose reads and writes per second instead of KBs per second. Examining the queue length and average service time counters is also extremely valuable in diagnosing I/O bottlenecks in applications that perform random I/O operations. Finally, use a graphing tool to plot the results and compare the I/O profile with the baseline results established with `iozone` and `bonnie++`. If the queue length is greater than two times the number of disks in the disk subsystem, check the average service time counters. If it is continually growing, there is an I/O bottleneck [47].

5.8 Profiling parallel I/O applications

Parallel applications are notoriously difficult to troubleshoot for performance issues or application errors. In some cases it is possible to run the application on a single node, in which case the previously discussed I/O profiling techniques would be sufficient. However, most parallel applications are designed to run on clusters consisting of hundreds or thousands of processing nodes. Performance issues that appear on the cluster very likely will not be apparent on a single node.

It is possible to create an I/O profile of a parallel application while it is running on a cluster; however, it can be difficult to isolate application I/O operations and CPU usage from system processes and other jobs running on the cluster. To capture the application's I/O traffic, identify the network interfaces or shared file systems that the application will utilize and monitor only that traffic. `Collectl` natively supports all TCP/IP networks and provides native support for Infiniband. In addition, `collectl` natively supports both NFS and Lustre file systems.

Another issue when collecting data across multiple nodes is keeping the results in sync. `Collectl` provides the ability to be started as a service and accessed remotely via a network socket. `Colmux` is a tool included with the collectl-utils package that connects to `collectl` services and multiplexes and synchronizes the output into a single stream. It has been tested on systems with thousands of nodes [52].

Once the appropriate data has been gathered together into a single synchronized stream, use `awk` or a spreadsheet to select the desired performance counters and then use a graphing utility to plot the application's I/O profile. The I/O profile will be useful for identifying application behavior; however, without a baseline for the cluster I/O subsystems it may be challenging to identify whether a parallel application is reaching its I/O potential.

5.9 Wrapping it up

Understanding how to capture and interpret an application's I/O characteristics is equal parts art and science. Benchmarking the test environment provides a baseline to compare application performance against. Real-time monitoring tools such as `iostat` and `top` can provide glimpses into how the application is currently performing based upon CPU, memory, and I/O utilization, but they do not give a view of the entire life of the application. Capturing snapshots of system performance counters in regular intervals throughput the life of the application using a tool such as `collectl` gives a more complete picture of the application's I/O characteristics. Plotting the values for swap, ram, CPUs, and disk subsystem IOPS and throughput creates an I/O profile that can be used to visually identify bottlenecks that may be missed when using real-time monitoring tools alone.

For a software developer, this technique enables the identification of potential I/O performance issues in the applications he or she writes. For a storage engineer, the I/O profiles produced using this technique provide insight into the performance capabilities of an application and can be leveraged to design storage systems with sufficient per node throughput as well as total aggregate throughput to allow the application to run at its full potential.

CHAPTER 6

CONCLUSION

6.1 Wrapping it up

The goal of this thesis has been to educate the reader on the issues involved in designing a reliable, high-performance storage system. High-performance storage systems are complicated and require expert level knowledge to design and maintain them. Many of the issues discussed in this thesis were experienced first hand by the author while working in the HPC lab at Boise State University. Issues of RAID failures due to latent sector errors and infant disk mortality are a part of life for a storage engineer, and knowing how to design reliable disk subsystems is critical. Understanding the configuration options and performance characteristics of the various parallel distributed file systems is valuable so that a good selection can be made from the get-go. It is difficult to perform an in-place migration from one parallel distributed file system to another while preserving a user's data so it is worth the effort to select an appropriate file system the first time. And finally, diagnosing application performance issues is part of the job. It is heart-sickening to design a beautiful, high-performance storage system just to discover that the end-user's applications perform slower than before. Understanding how to profile the I/O characteristics of end-user applications is extremely valuable in the initial design of storage systems but also in working with the end-user to identify performance constraints.

6.2 Extensions of this research

Several ideas for research topics have come out of this storage systems research. The following is a brief summary of these ideas.

6.2.1 Parallel file transfer

Imagine attempting to copy an entire directory tree containing more than 16TB distributed across one million files between two parallel distributed file systems. The files are a mix of small files (< 1MB) and large files (> 100 MB). Simply running a `cp` command on a single node will require days to complete. It would be significantly faster to use the compute nodes on the cluster to transfer the files in parallel. The file balancing problem is to ensure that the file transfer effort is evenly distributed across all the processing nodes. If the load is unbalanced, then a small number of nodes would be doing the bulk of the work while other nodes remain idle.

The initial load metric could simply be file size, ensuring that all nodes transfer a similar amount of data. However, this will not guarantee that the load is equal since transferring 1GB worth of 1KB files will take significantly longer than transferring a single 1GB file.

The following describes three different implementation strategies. The first is where the master process scans the source directory structure and generates a balanced distribution of work for all slave nodes. The second allows each node to parse the directory structure using a hash mechanism to determine which files it should transfer (this would only balance the number of files transferred per node). The last idea is more of a work-pool approach. In this approach, each slave would contact the master node and request a unit of work, say 10GB worth of files. The master node could track which nodes have been assigned which work units, and when a node completes a given unit of work, it would be given another.

6.2.2 Data management: storage zones and data preservation strategies

Conduct a survey of how other institutions with HPC environments deal with data backup and recovery of their high-performance storage systems. There are many technologies to consider including mountable snapshots, tape backup libraries, and backup to another storage system. The idea is to present the options and hash through the pros and cons of each. Backing up hundreds of terabytes of data is not trivial, and it would be valuable to see how this task is accomplished in a variety of production environments.

In addition to preserving user data, this research could be leveraged to best utilize various types of storage that may be available within a high-performance storage system. For instance, a storage system may contain a mix of high-throughput / low-latency SSDs (also high cost), 15k SAS disks, and 7.2k SAS/SATA disks (low cost). Experiments could be performed to identify which types of data (and the associated applications) are best suited for each type of storage media, with the data grouped into storage zones. It would also be interesting to see if data could be staged automatically by the storage system with less frequently accessed data residing on the low cost 7.2k disks while high frequency data access could be performed upon the SSDs.

6.2.3 Statistical model to calculate ideal number of hot-swap disks to include in a storage system

As a supplement to the charts for probability of encountering an unrecoverable read error while reading a disk subsystem (Figures 3.3a and 3.3b) and the MTTDL chart (Figure 3.4), it would be valuable to develop a model that calculates the ideal number of hot-swap disks to include in a storage system of N disk drives across M controllers. There will likely be hardware limitations requiring that the hot-swap disk(s) be located in the same storage shelf as the arrays it may be substituted into. If a shelf holds 16 disks, how many disks should be reserved as hot-swaps? This may be a moot point, as many storage engineers would be hesitant to allocate more than two disks as hot-swap given the cost of each storage slot, preferring to keep off-line spares available and invest in a monitoring solution to quickly identify failed disk drives.

6.2.4 Disaster recovery of parallel distributed file systems

You had a backup, right? Disk drives fail. Disk subsystems fail. But what happens when the power fails? Parallel distributed file systems can potentially become corrupted from a variety of events including hardware failures, power failures, or a variety of administrator-induced issues (accidents). It would be interesting to experiment with different ways of corrupting a parallel distributed file system, then demonstrate the techniques used to recover data. By providing a guide on how to recover a corrupted storage system, system administrators will have a step by step recovery guide to follow instead of having to figure it out on the fly.

6.2.5 High-availability configurations for parallel distributed file systems

A reliable storage system is one that can survive hardware failures without losing data. A highly available storage system is one that can survive hardware failure without interruption of service to I/O requests. Parallel distributed file systems that do not provide replication functionality, such as PVFS and Lustre, require shared storage on the backend to remain available in the event of hardware failures. It would be interesting to explore the use of DRBD over Infiniband as well as multi-host SAS arrays for the shared backend storage.

6.2.6 Persistent versus non-persistent scratch space in HPC environments

Many applications that run in HPC environments require high-performance scratch space to store intermediate results that are generated while the application is running. They are then processed in some manner to obtain useful insight into the data, but many times are never referenced again. This intermediate data can range from hundreds of gigabytes to several terabytes in size, taking up extremely valuable space that could be utilized by other applications.

From an administrative standpoint, this is a difficult problem because it is not feasible to be continuously adding storage capacity, both from a cost perspective and a system management perspective. So how can an administrator ensure that sufficient high-performance scratch space remains available for end-user applications? One option is to send the users a summary of their data usage and ask them to please cleanup stale data. Unfortunately, this approach has met with limited success in the HPC lab at Boise State University. The other option is to establish system-wide policies for the removal of end-user data from the storage system.

This topic would explore different methods for managing storage usage and data lifetimes for high-performance persistent and non-persistent high-performance scratch space in HPC environments. One option for managing storage usage is disk quotas. Unfortunately, not all parallel distributed file systems support quotas, so it would be important to document which parallel distributed file systems support quotas and document how to implement them. This would help to address the issue of stale data in persistent scratch space because the user would need to remove old data before storing new data to the scratch space. Another approach would be to establish policies, automatically enforced by scripts, that remove any data from the scratch space that is older than n number of days. The users would be notified multiple times before the data was removed from the system, giving them the chance to finish using the data or move it to archive storage.

Another approach to addressing the issue of stale data consuming valuable high-performance scratch space is the use of non-persistent scratch space. In this instance, a temporary parallel distributed file system would be dynamically created across raw storage disks within the storage cluster when a user submits a job to the cluster. This space will be available to the user while the job is running, but will be destroyed once the job completes. This approach forces the users to immediately process the data or move it to archive storage or it will be lost. This technique is used for high-performance scratch space at many research universities and national laboratories.

An area to investigate is which parallel distributed file systems provide quota support and are suitable for use as non-persistent scratch space. It would be extremely valuable to research and document how this issue is addressed in various HPC environments throughout the country. It would also be good to demonstrate different techniques for implementing the discussed approaches including detailed instructions for using them in production environments.

6.2.7 Objective-C inspired dynamically generated non-persistent scratch space for HPC environments

The memory management model for Objective-C on the iOS platform uses a technique called reference counting to determine when the memory for an object should be freed. Upon revisiting this issue of non-persistent scratch space, I realized a similar approach could be leveraged. The previously described implementation of non-persistent scratch space is rather heavy handed, with a use it or loose it approach. What if a storage system implemented a form of reference counting for each dynamically created non-persistent scratch space? Let us see how a system like this would work.

A user submits a batch job to the cluster using a script. One of the first actions performed is to contact the storage system manager (running as a web service) to request a scratch space with a given size, specific striping characteristics including distribution technique and stripe size, and a file system ID. The storage system manager would perform a number of checks to ensure resources were available to create the scratch space, a scratch space with that ID did not already exist, and that the user had permission to allocate a scratch space. Once these criteria are satisfied, the storage manager would generate the appropriate configuration files and start the services on the allocated storage nodes. It would then return an object to the calling script with the file system ID and the appropriate connection information. The script would then mount the dynamic scratch space, implicitly taking ownership of the storage space.

At this point the batch job can perform any necessary I/O operations using its scratch space. Just before the batch job completes, the script can issue a retain or a release message. If a release message is sent, the storage system manager will immediately schedule the scratch space for deallocation. If a retain message is sent, the non-persistent scratch space will be held for a specific period, say two weeks. This will allow more jobs to be run against the data in the non-persistent scratch space. As each job completes, they will have the option of releasing or retaining the scratch space. Data that is accessed frequently will be retained indefinitely while data that is not accessed will be purged.

In the event that a new scratch space is created but not explicitly retained or released, a default retention period will be used that is the same interval as if the data had been explicitly retained. This space will be flagged as being in limbo but will be maintained as long as sufficient resources remain available on the storage system. However, in the event that a request for scratch space comes in that requires more storage resources than are currently available, any space that is in limbo and has no active connections (it is not mounted anywhere), will be released.

6.2.8 Extended application I/O profiling

Chapter 5 demonstrates the usefulness of I/O profiles in identifying the performance constraints in sequential applications. It would be useful to provide examples for applications that perform random (non-sequential) I/O operations as well as for applications running in a clustered environment. There are literally hundreds of system counters that `collectl` can retrieve data from including network file systems and network interconnects, and the I/O results are not limited to simply throughput. Generate I/O profiles for a wide variety of applications. Is there a basic set of categories that all applications fall into? Is it possible to classify an application based upon its I/O profile?

Another aspect to consider is the development of scripts to stream-line the data capture, manipulation, and display procedures to make the I/O profiling technique as painless as possible for the end user. The `collectl` and `colmux` tools are extremely powerful tools with many configuration options. These scripts would wrap up the desired functionality into easy-to-use scripts that would automatically generate data formatted for import directly into a spreadsheet or even generate pre-configured charts with `gnuplot` or `ploticus`. Accompanying these scripts would be a detailed user guide with detailed instructions for setting up the profiling tools and environment as well as many examples covering a variety of application types. These should include applications that perform sequential I/O, random I/O, and parallel I/O operations.

BIBLIOGRAPHY

[1] Adaptec Storage Advisors. Raid reliability calculations. http://www.adaptec.com/blog//2005/11/01/raid-reliability-calculations/, November 2005.

[2] APC. White Paper 13: Experts speak on UPS output Watt, VA, and Power Factor ratings. http://www.apc.com/.

[3] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. *SIGMETRICS Perform. Eval. Rev.*, 35:289–300, June 2007.

[4] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau. An Analysis of Data Corruption in the Storage Stack. In *In Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, 2008.

[5] William S. Baring-Gould. *The Annotated Sherlock Holmes, Volume I.* Clarkson N. Potter, Inc., One Park Avenue, New Your, N.Y. 10016, USA, 1967.

[6] Jeff Bell. Calculating mean time to data loss (and probability of silent data corruption). http://info.zetta.net/blog/bid/45661/, June 2009.

[7] Troy Benjegerdes. Netpipe website. http://bitspjoule.org/netpipe/.

[8] Don Capps. Iozone website. http://www.iozone.org/.

[9] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-File Access in Parallel File Systems. *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 2009.

[10] Russell Coker. Bonnie++ website. http://www.coker.com.au/bonnie++/.

[11] Oracle Corporation. Eliminating Silent Data Corruption in Oracle Databases. http://www.emulex.com/artifacts/373fa749-f5fc-41e3-9b0d-4d475123c2af/eliminating-silent-data-corruption.pdf.

[12] Oracle Corporation. Lustre 1.8 Operations Manual. http://wiki.lustre.org/manual/LustreManual18_HTML/index.html.

[13] Ajay Dholakia, Evangelos Eleftheriou, Xiao-Yu Hu, Ilias Iliadis, Jai Menon, and K.K. Rao. A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *Trans. Storage*, 4:1:1–1:42, May 2008.

[14] National Center for Computational Sciences. Jaguar File Systems. http://www.nccs.gov/computing-resources/jaguar/file-systems.

[15] The Apache Software Foundation. Hdfs architecture guide. http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html, 2009.

[16] Jehan françois Pâris, Thomas J. E. Schwarz, Darrell D. E. Long, and Ahmed Amer. When MTTDLs Are Not Good Enough: Providing Better Estimates of Disk Array Reliability.

[17] Warren Gay. *Linux Socket Programming by Example*. Que, 2000.

[18] GLUSTER. About Us. http://www.gluster.com/company/index.php.

[19] GLUSTER. Storage Server Installation and Configuration. http://gluster.com/community/documentation/index.php/Storage_Server_Installation_and_Configuration.

[20] GLUSTER. Translators. http://gluster.com/community/documentation/index.php/Translators.

[21] Sebastien Godard. Sysstat website. http://sebastien.godard.pagesperso-orange.fr/.

[22] Jim Gray and Catharine van Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. http://research.microsoft.com/pubs/64599/tr-2005-166.pdf, December 2005.

[23] Volker Gropp. Bwm-ng website. http://www.gropp.org/.

[24] Marissa Hollingsworth. Performance and usability of fuse-dfs: A filesystem in userspace module for the hadoop distributed file system, October 2010.

[25] IBM. 709 Data Processing System. http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html.

[26] INTEL. Intel microprocessor export compliance metrics. http://www.intel.com/support/processors/sb/cs-023143.htm.

[27] Intel. Enterprise-class versus desktop-class hard drives. `ftp://download.intel.com/support/motherboards/server/sb/enterprise\_class\_versus\_desktop\_class\_hard\_drives\_.pdf`, April 2008.

[28] Josh Jaques. Mountablehdfs - hadoop wiki. `http://wiki.apache.org/hadoop/MountableHDFS`, March 2010.

[29] Jeffrey Layton. Lies, damn lies and file system benchmarks. `http://www.linux-mag.com/id/7464/`, August 2009.

[30] Juan Loaiza. Optimal storage configuration made easy. www.miracleas.com/BAARF/oow2000_same.pdf.

[31] NASA. Computers in Spaceflight: The NASA Experience. `http://history.nasa.gov/computers/Ch8-2.html`.

[32] Jehan-François Pâris, Ahmed Amer, Darrell D. E. Long, and Thomas Schwarz. Evaluating the Impact of Irrecoverable Read Errors on Disk Array Reliability. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 379–384, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Patrick. The raid reliability anthology. `http://www.servethehome.com/raid-reliability-failure-anthology-part-1-primer/`, August 2010.

[34] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[35] Neil Rasmussen. White Paper 15: Watts and Volt-Amps: Powerful Confusion. `http://www.apc.com/`.

[36] Neil Rasmussen. White Paper 25: Calculating Total Cooling Requirements for Data Centers. `http://www.apc.com/`.

[37] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2007. USENIX Association.

[38] Thomas J. E. Schwarz, Qin Xin, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *In Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418. IEEE, 2004.

[39] Seagate. Barracuda 7200.12 serial ata product manual. http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369b.pdf, February 2009.

[40] Seagate. Cheetah ns.2 data sheet. http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_ns_2.pdf, January 2009.

[41] Seagate. Drive selection guide: A comparison of seagate 7200-rpm drives. http://www.seagate.com/docs/pdf/whitepaper/mb538-drive-selection-guide.pdf, May 2009.

[42] Seagate. Constellation es series product manual. http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/Constellation%203_5%20in/100602414e.pdf, September 2010.

[43] Seagate. Storage solutions guide. http://www.seagate.com/docs/pdf/whitepaper/storage_solutions_guide.pdf, October 2010.

[44] Mark Seger. Collectl website. http://collectl.sourceforge.net/.

[45] Gregory Smith. The linux page cache and pdflush: Theory of operation and tuning for write-heavy loads. http://www.westnet.com/~gsmith/content/linux-pdflush.htm, August 2007.

[46] WDC Support Staff. Difference between desktop edition and raid (enterprise) edition drives. http://wdc.custhelp.com/app/answers/detail/a_id/1397/kw/installing+desktop+edition+hard+drives+in+an+enterprise+environment/session/L3RpbWUvMTI5NzExOTMzNi9zaWQvKjE4clYybWs%3D, February 2011.

[47] Solid Data Systems. Using I/O Signature Analysis to Improve Database Performance. http://www.soliddata.com/resources/pdf/WP_IOSignatures_v3.pdf.

[48] PVFS Development Team. Parallel Virtual File System, Version 2. http://www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc/pvfs2-guide/pvfs2-guide.php, September 2003.

[49] PVFS Development Team. PVFS Tuning. Included with pvfs-2.8.2 source code in the docs folder, March 2009.

[50] Rajeev Thakur, Robert Ross, Ewing Lusk, William Gropp, and Robert Latham. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation, 2010.

[51] TOP500.Org. November 2009: TOP500 Supercomputer Sites. `http://www.top500.org/lists/2009/11`.

[52] unknown. Colmux website. `http://collectl-utils.sourceforge.net/colmux.html`, February 2011.

[53] Henry Ware and Fabian Fredrick. vmstat man page. man vmstat.

[54] James Warner, Albert Cahalan, and Craig Small. Top man page. man top.

[55] Wyatts. Bathtub curve. `http://commons.wikimedia.org/wiki/File:Bathtub_curve.svg`, August 2009.

[56] Qin Xin, Ethan L. Miller, Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for Very Large Storage Systems. *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, 2003.

[57] Qin Xin, Thomas J. E. Schwarz, and Ethan L. Miller. Disk infant mortality in large storage systems. In *In Proc of MASCOTS '05*, pages 125–134, 2005.

APPENDIX A

STORAGE CONFIGURATION WORKSHEET

- Company name:
- Contact individual:
- Date:
- Background
  - What problem is the customer attempting to solve?
  - Why is the customer interested in a high-performance storage solution?
  - Does the customer have an existing storage solution, and if so, what is it?
  - What are the actual/perceived limitations of the existing solution?

    Talk to management, application developers, and system engineers. Are the perceived limitations due to storage capacity, performance, or reliability? List specific items of concern and demonstrate how they will be resolved with the new storage system.
  - What does the customer want?

    This is an open-ended question but should be answered in terms of storage capacity, performance, and redundancy. Again, make a specific list of items and demonstrate how the design of the new storage system meets these requirements.
- Storage capacity and growth
  - How much usable storage capacity is initially required?
  - What is the expected growth over the next two to three years?

- Storage client details
  - How many clients will access the storage system?
  - Is the number of clients expected to grow over the next two to three years?
  - What types of storage clients will need to access the storage system?

    Will it be a homogeneous or heterogeneous mix of storage client systems (Windows, Linux, Unix, or Mac?)
  - Identify which clients will require high-throughput access to the storage system.
- Data details
  - Identify the classes of data to be stored on the storage system.
  - Define storage zones to manage data with similar storage requirements.
    - Data that will benefit from file striping
    - Data that will benefit from simple distribution
    - Data that will require a specific level of replication
    - Data that will require backup/archival to external media
  - Diagram how data will flow through the system.

    How will data be added to the storage system? How will data be removed/cleaned up from the storage system?
  - Who will manage the data in each storage zone?
- Application details
  - What applications are going to access the storage system?

    Get a list of all applications that will access data residing on the storage system. Each of the applications on the list should be associated with the storage clients they will run on as well as the data/storage zones they will access.
  - What are the I/O characteristics of the applications accessing this storage system?

    It is helpful to generate an I/O profile for each application. It needs to show network and block device I/O throughput as well as CPU and memory utilization in snapshots captured every five to thirty seconds throughout the run-time of the application. This information can be used to identify performance constraints within the application while providing insight into what throughput levels the application is capable of sustaining. Beyond this, it is helpful to know access details such as sequential vs random accesses and the chunk size of each access. Unfortunately, these details can be hard to come by; however, some inferences can be made based upon the type of data that the application is reading from or writing to.

- Disaster recovery
  - Is the data stored on the storage system critical to business operation?

    Can the data be regenerated if deleted, corrupted, or lost due to hardware failure?

    How should the critical data on the storage system be backed up?
  - Is access to the storage system critical to business operation?

    Are there specific up time requirements the system must meet? (acceptable hours of unscheduled downtime per month)
- Facility
  - Where will the system be housed?

    Co-location facility, on-site?
  - What is the available capacity for power?

    Is there a site UPS system available? Is there a backup generator available?
  - What is the available capacity for cooling?
- Budget
  - What is the budget for the initial system?
  - What is the monthly budget for power and cooling costs?
  - What is the budget for maintenance?

    Is the customer planning on purchasing vendor maintenance agreements? If not, is there a budget for spare components?
  - Is there a line item in the budget for an engineer's salary to maintain the storage system?

APPENDIX B

DATA CLASSIFICATION

The following sections attempt to classify data based upon the type of data being stored, the file format used, and the size of the files. The types of files described below are all data types that we have had to deal with at one point or another in the HPC lab at Boise State University. The conclusions were not derived from any specific research testing but rather were collected from knowledge of the formats and observations of the associated applications. More scientific results could be generated with extensive testing of representative data sets and the generation of I/O profiles for the associated applications; however, the data classifications provided in the following sections provide a good jumping off point to get the reader started thinking about his or her own data.

B.1 Large multimedia files (greater than 100MB)

- Typically sequential access
- Video formats such as divx, mpeg, dv
- Audio formats such as wav, flac, raw
- These formats can be processed as a real-time stream requiring low bit-rates (less than 100Mbit)
- These formats can be processed as blocks or chunks utilizing high bit-rates (greater than or equal to 100Mbit)

- Suffers from interconnect/storage system latency when processed using small block sizes
- Benefits from read-ahead caching
- Benefits from write-back caching
- Benefits from striping

B.2 Large text files (greater than 100MB)

- Typically sequential access
- Processed as chars, lines, or blocks utilizing high bit-rates (greater than or equal to 100Mbit)
- Suffers from interconnect/storage system latency when processed using chars, lines, or small block sizes
- Benefits from read-ahead
- Benefits from write-back caching
- Benefits from striping

B.3 Large compressed files (greater than 100MB)

- Typically sequential access
- Formats such as zip, gzip, bzip2
- NetCDF (mix of random and sequential depending upon organization of data in file)
- These formats can be processed as chunks utilizing high bit-rates (greater than or equal to 100Mbit)
- Suffers from interconnect/storage system latency when processed using small block sizes
- Benefits from read-ahead caching
- Benefits from write-back caching
- Benefits from striping

B.4 Large database files (greater than 100MB)

- Typically random access
- MySQL Datastore
- B-tree, db2
- Lucene Indexes
- Virtual Disks: vmware, virtualbox, mounted ISO images
- Various processing methods including pages and chunks
- Suffers from striping overhead
- Suffers from gigabit Ethernet latency
- Benefits from high performance low latency disk drives
- Benefits from high performance low latency network interconnect

B.5 Medium multimedia files (1MB - 100MB)

- Typically sequential access
- Graphic Formats: Jpeg, BMP, TIFF, PDF, etc.
- Audio Formats: wav, MP3, OGG, FLAC
- Video Formats: mpeg, divx, dv
- These formats can be processed as blocks or chunks utilizing high bit-rates (greater than or equal to 100Mbit)
- Can benefit from striping depending upon client access pattern by eliminating hot spots
- Benefits from simple distribution by eliminating striping overhead
- Benefits from read-ahead caching
- Benefits from write-back caching
- Good candidate for file replication

B.6 Medium text files (1MB - 100MB)

- Typically sequential access
- Processed as chars, lines, or blocks utilizing high bit-rates (greater than or equal to 100Mbit)
- Suffers from interconnect/storage system latency when processed using chars, lines, or small block sizes
- Benefits from read-ahead caching
- Benefits from write-back caching
- Benefits from simple distribution by eliminating striping overhead
- Good candidate for file replication

B.7 Medium compressed files (1MB - 100MB)

- Typically sequential access
- Formats such as zip, gzip, bzip2
- NetCDF (mix of random and sequential depending upon organization of data in file)
- These formats can be processed as chunks utilizing high bit-rates (greater than or equal to 100Mbit)
- Suffers from interconnect/storage system latency when processed using small block sizes
- Benefits from simple distribution by eliminating striping overhead
- Benefits from read-ahead caching
- Benefits from write-back caching
- Good candidate for file replication

B.8 Medium database files (1MB - 100MB)

- Typically random access
- MySQL Datastore
- B-tree, db2
- Lucene Indexes
- Virtual Disks: vmware, virtualbox, mounted ISO images
- Various processing methods including pages and chunks
- Suffers from striping overhead
- Suffers from gigabit Ethernet latency
- Benefits from high performance low latency disk drives
- Benefits from high performance low latency network interconnect

B.9 Small files (less than 1MB)

- Video, Text, Audio, Picture, Database
- Processed as char, line, chunk, block
- Suffers from striping overhead
- Suffers from gigabit Ethernet latency
- Benefits from high performance low latency disk drives
- Benefits from high performance low latency network interconnect
- Benefits from simple distribution
- Good candidate for file replication

B.10 Large number of files (small or large)

- Metadata operations benefit greatly from high performance low latency disk drives housing metadata
- Client connectivity to metadata server benefits greatly from low latency interconnect
- Extra memory in the metadata server allows for caching of metadata and increases performance
- Multiple metadata servers decreases the bottleneck that occurs when many clients are performing metadata operations simultaneously

APPENDIX C

APPLICATION I/O PROFILING WORKSHEET

- Application name:
- Test date:
- System details

  Type of processor:

  Total number of cores (across all processors):

  Total amount of system memory:

  Disk subsystem RAID configuration:

  Number of disks in disk subsystem:

  Network interface details:

- Baseline of test environment I/O performance:

  Record `bonnie++` throughput and metadata baseline in the tables below

  (Optional) Attach 3D Surface plots of `iozone` results for sequential and random throughput and IOPS testing

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU

Bonnie++ Metadata Performance					
Sequential (ops/sec)			Random (ops/sec)		
create	read	delete	create	read	delete

- Application I/O Profiles

  Generate I/O profiles for the application using the technique described in Chapter 5 and attach them to this worksheet

- Evaluation

  Compare the I/O profile against the established baseline values for the test environment.

  Is I/O activity evenly distributed over the lifetime of the application or are there times of high I/O activity and times of low I/O activity?

  Use a marker to identify the critical features of the I/O profile, specifically times when the application is CPU bound, memory bound, or I/O bound.

APPENDIX D

ATLANTIS RESEARCH CLUSTER CONFIGURATION

D.1 Storage node specifications

Table D.1 shows the specifications for the nodes in the Atlantis research cluster. The first four Atlantis nodes (*atlantis0[0-3]*) were purchased in 2008 from ebay for a grand total of $800.00 (including shipping). These nodes were used extensively for much of the early work with distributed file systems as well as the work on application profiling, the results of which are presented in Chapters 4 and 5. *Atlantis04* and *atlantis05* were purchased in 2010 from ebay for a grand total of $200.00 (including shipping). The high disk density of these nodes (16 disks each), allowed for experimentation with Linux software RAID configurations, RAID scrubbing, and RAID recovery techniques. All in all, a solid investment.

Table D.1: Atlantis research cluster node specifications

	atlantis00	*atlantis0[1-3]*	*atlantis0[4-5]*
Processor	2x 2.8GHz Xeon	2x 2.6GHz Xeon	3.2GHz P4
Memory	2GB ECC DDR	2GB ECC DDR	1GB DDR
Network	2xGigE	2xGigE	GigE
RAID Controller	3ware(SATA), Adaptec(U320)	Adaptec(U320)	2x3ware(PATA)
Hard Disks	4x500GB SATA, 4x36GB 15k SCSI	4x36GB 15k SCSI	16x80GB PATA

D.2 Network diagram

For much of its life, the Atlantis research cluster was housed in the Beowulf HPC Lab at Boise State University. Figure D.1 represents how the storage nodes of Atlantis were linked with the 60 node Beowulf cluster. This configuration allowed a wide range of options for measuring file system scalability and efficiency as the number of clients increased.

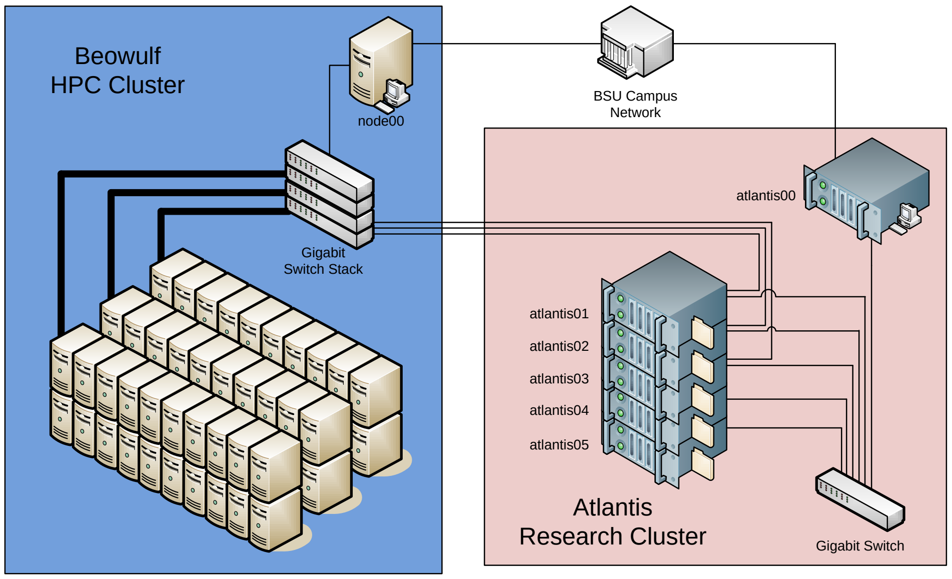


Figure D.1: Network layout of Atlantis research cluster

D.3 Chapter 4 RAID configuration and performance baseline

The nodes of the Atlantis research cluster were stripped down and rebuilt multiple times over the course of this thesis. For the parallel distributed file system research in Chapter 4, the four 15k RPM SCSI disks on *atlantis0[0-3]* were configured for RAID-0 striping using the onboard Adaptec hardware RAID controller. In addition to being used as the backend datastore for the distributed file systems, this disk subsystem also was used for the Linux system partitions. Tables D.2, D.3, D.4, and D.5 show the baseline block throughput results for these nodes. These results were gathered using `bonnie++`, but at the time of this testing, emphasis was mostly on block throughput, not on character I/O or metadata performance.

Table D.2: Chapter 4 throughput results for md0 on *atlantis00*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
–	–	155	99	83	36	–	–	219	31

Table D.3: Chapter 4 throughput results for md0 on *atlantis01*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
–	–	152	99	86	32	–	–	225	34

Table D.4: Chapter 4 throughput results for md0 on *atlantis02*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
–	–	161	86	85	35	–	–	223	34

D.4 Chapter 5 RAID configuration and performance baseline

The I/O profiling research for Chapter 5 required a dedicated disk subsystem to ensure accurate results. For this research, *atlantis01* and *atlantis02* were rebuilt with

Table D.5: Chapter 4 throughput results for md0 on *atlantis03*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
–	–	162	86	87	36	–	–	216	33

a single disk (sda) used for the system partitions. The remaining three 15k RPM SCSI U320 disks in each node were configured using Linux software RAID (md0). The disks were configured for RAID-0 striping with a chunk size of 256KB. As the research progressed, it became apparent that another dedicated disk subsystem would be needed to fully demonstrate the profiling process. To this end, a second Adaptec SCSI controller card was added to *atlantis02* along with three additional 15k RPM SCSI U320 disks, also configured for RAID-0 striping with Linux software RAID (md1).

Tables D.6 and D.7 show the baseline throughput and IOPS for dedicated disk subsystem (md0) on atlantis01. Next, Tables D.8 and D.9 show the baseline throughput and IOPS for the primary dedicated disk subsystem (md0) on atlantis02. Finally, Tables D.10 and D.11 show the baseline throughput and IOPS for the secondary dedicated disk subsystem (md1) on *atlantis02*.

Table D.6: Chapter 5 throughput results for md0 on *atlantis01*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
33	99	171	99	72	32	39	97	183	27

Table D.7: Chapter 5 IOPS results for md0 on *atlantis01*

Bonnie++ Metadata Performance					
Sequential (ops/sec)			Random (ops/sec)		
create	read	delete	create	read	delete
11170	194149	2794	9483	235479	1978

Table D.8: Chapter 5 throughput results for md0 on *atlantis02*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
36	99	199	98	76	29	42	97	196	25

Table D.9: Chapter 5 IOPS results for md0 on *atlantis02*

Bonnie++ Metadata Performance					
Sequential (ops/sec)			Random (ops/sec)		
create	read	delete	create	read	delete
16194	213798	3307	13476	260769	2299

Table D.10: Chapter 5 throughput results for md1 on *atlantis02*

Sequential Output						Sequential Input			
Char		Block		Rewrite		Char		Block	
MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU
35	97	187	93	80	31	41	94	195	25

Table D.11: Chapter 5 IOPS results for md1 on *atlantis02*

Bonnie++ Metadata Performance					
Sequential (ops/sec)			Random (ops/sec)		
create	read	delete	create	read	delete
16095	214388	2882	13674	261441	2018

APPENDIX E

SEQPROCESSOR APPLICATION SOURCE CODE

E.1 Seqprocessor version 1

```c
/*
 * File:    seqprocessor-1.c
 * Author: Luke Hindman
 * Created on September 29, 2010, 11:45 AM
 */

/* Allow fopen to handle files larger than 2GB */
#define _FILE_OFFSET_BITS 64

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage(char * program) {
    fprintf(stderr,"Usage:  %s <source file> <destination file> [buffer
        size]\n",program);
    exit(EXIT_FAILURE);
}

/*
 *  Read from a FASTA DNA sequence file and write to a different file
 *  This version uses fgetc() and fputc() for I/O operations.
 */
int main(int argc, char** argv) {
    FILE * src_file;
    FILE * dest_file;
    int current_char;

    if (argc < 3)
        usage(argv[0]);

    src_file = fopen(argv[1],"r");
    if (src_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    dest_file = fopen(argv[2],"w");
    if (dest_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    /*Tight loop for transferring data from src to destination*/
    while ( (current_char=fgetc(src_file)) != EOF ) {
        fputc(current_char,dest_file);
    }

    fclose(src_file);
    fclose(dest_file);

    return (EXIT_SUCCESS);
}
```

Listing E.1: Seqprocessor source code (version 1)

E.2 Seqprocessor version 2

```
/*
 * File:   seqprocessor-2.c
 * Author: Luke Hindman
 * Created on September 29, 2010, 11:45 AM
 */

/* Allow fopen to handle files larger than 2GB */
#define _FILE_OFFSET_BITS 64

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage(char * program) {
    fprintf(stderr,"Usage:  %s <source file> <destination file> [buffer
        size (KB)]\n",program);
}

/*
 *  Read from a FASTA DNA sequence file and write to a different file
 *
 *  This version uses fread() and fwrite() with a variably sized buffer
 *   to transfer data from src to destination.  The default buffer size
 *   is 4KB.
 *
 */
int main(int argc, char** argv) {
    FILE * src_file;
    FILE * dest_file;
    size_t buffer_size;

    if (argc < 3) {
        usage(argv[0]);
        exit(EXIT_FAILURE);
    } else if (argc >= 4)
        buffer_size = atoi(argv[3]) * 1024;
    else
        buffer_size = 4 * 1024;  /* 4KB character buffer */

    /* Allocate buffer*/
    char *buffer = (char *) malloc(sizeof(char) * buffer_size);

    src_file = fopen(argv[1],"r");
    if (src_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    dest_file = fopen(argv[2],"w");
    if (dest_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    /*Tight loop for transferring data from src to destination*/
    int count;
    while ( (count = fread(buffer, sizeof(char), buffer_size / sizeof(
        char), src_file)) > 0 ) {
        fwrite(buffer,sizeof(char), count, dest_file);

    }

    free(buffer);
    fclose(src_file);
    fclose(dest_file);

    return (EXIT_SUCCESS);
}
```

Listing E.2: Seqprocessor source code (version 2)

E.3 Seqprocessor version 3

```
/*
 * File:   seqprocessor-3.c
 * Author: Luke Hindman
 * Created on September 29, 2010, 11:45 AM
 */

/* Allow fopen to handle files larger than 2GB */
#define _FILE_OFFSET_BITS 64

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define NUM_BUFFERS 8  /*Number of buffers */
#define BUFFER_SIZE 4  /*Size of each buffer in KB */
#define EMPTY 150 /*Reader thread should fill the buffer*/
#define FULL 250  /*Writer thread should empty the buffer*/
#define DONE 350  /*Reader thread is done reading */
#define TRUE 1
#define FALSE 0



pthread_t reader_thread;
pthread_t writer_thread;
pthread_mutex_t reader_done_mutex;
int reader_done;

unsigned int num_buffers;
size_t buffer_size;

pthread_mutex_t * buffer_mutex;
char * buffer_store;
char ** buffer;
int * buffer_status;
size_t * buffer_used;
pthread_cond_t * buffer_cond_var;


void *reader(void *arg) {
    FILE * src_file = (FILE *) arg;
    int i = 0;
    unsigned long int read_count = 0;
    unsigned long int last_read_count = 0;
    if (src_file != NULL) {
        while (!feof(src_file) && !ferror(src_file) ) {
            pthread_mutex_lock(&buffer_mutex[i]);
            while (buffer_status[i] != EMPTY) {
                /* Wait until a buffer is available in which to
     * store data */
                pthread_cond_wait(&buffer_cond_var[i],&buffer_mutex[i]);
            }
            buffer_used[i] = fread(buffer[i], sizeof(char), buffer_size,
                src_file) ;
            read_count += buffer_used[i];
            last_read_count = buffer_used[i];
            buffer_status[i] = FULL;
            pthread_mutex_unlock(&buffer_mutex[i]);
            /* Send signal to writer thread that a buffer is available
             * from which to read data for output to disk. */
            pthread_cond_signal(&buffer_cond_var[i]);
            i = (i + 1) % num_buffers;
        }
        pthread_mutex_lock(&buffer_mutex[i]);
        fprintf(stderr,"Reader is Done!(%d,%d)\n",i,buffer_status[i]);
        pthread_mutex_unlock(&buffer_mutex[i]);
        pthread_mutex_lock(&reader_done_mutex);
        reader_done = TRUE;
        pthread_mutex_unlock(&reader_done_mutex);

        /* Send signal to writer thread incase it is blocking on the
         * last bucket.  This should prevent dead-lock in this case. */
        pthread_cond_signal(&buffer_cond_var[i]);
    }
    fprintf(stdout,"ReadCount: %d\n",read_count);
    fprintf(stdout,"LastReadCount: %d\n",last_read_count);
    pthread_exit((void*) 0);
}

void *writer(void *arg) {
    FILE * dest_file = (FILE *) arg;
    int i = 0;
    unsigned long int write_count = 0;
    if (dest_file != NULL) {
        while ( 1 ) {
            pthread_mutex_lock(&buffer_mutex[i]);
            while (buffer_status[i] == EMPTY) {
                /* Check to see if the reader thread is done */
                pthread_mutex_lock(&reader_done_mutex);
                if (reader_done == TRUE) {
                    pthread_mutex_unlock(&reader_done_mutex);
                    fprintf(stdout,"WriteCount: %d\n",write_count);
                    pthread_exit((void*) 0);
                }
                pthread_mutex_unlock(&reader_done_mutex);

                /* Wait until a buffer is available from which read data
                 * to write to disk*/
                pthread_cond_wait(&buffer_cond_var[i],&buffer_mutex[i]);
            }

            write_count += fwrite(buffer[i],sizeof(char), buffer_used[i
                ], dest_file);
            buffer_status[i] = EMPTY;
            pthread_mutex_unlock(&buffer_mutex[i]);
            /* Send signal to reader thread that a buffer is empty. */
            pthread_cond_signal(&buffer_cond_var[i]);
            i = (i + 1) % num_buffers;
        }
    }
    fprintf(stdout,"WriteCount: %d\n",write_count);
   pthread_exit((void*) 0);
}

void usage(char * program) {
    fprintf(stderr,"Usage:  %s <source file> <destination file> [number
        of buffers] [buffer size (KB)]\n",program);
}

/*
 *  Read from a FASTA DNA sequence file and write to a different file
 *
 *  This version is multi-threaded and uses circular buffering as well
 *  as fread() and fwrite() with a variably sized buffers to transfer
 *  data from src to destination.  The default buffer size is 4KB.
 */
int main(int argc, char** argv) {
    FILE * src_file;
    FILE * dest_file;

    int i;
    pthread_attr_t attr;
    void *status;

    if (argc < 3) {
        usage(argv[0]);
        exit(EXIT_FAILURE);
    } else if (argc >= 5) {
        num_buffers = atoi(argv[3]);
        buffer_size = atoi(argv[4]) * 1024;
    } else if (argc >= 4) {
        num_buffers = atoi(argv[3]);
    } else {
        buffer_size = BUFFER_SIZE * 1024 ;
        num_buffers = NUM_BUFFERS;
    }

    /* Allocate Arrays */
    fprintf(stderr,"Allocating global arrays!\n");
    buffer_mutex = (pthread_mutex_t *) malloc (sizeof(pthread_mutex_t) *
         num_buffers);
    buffer_store = (char *) malloc(sizeof(char*) * num_buffers *
        buffer_size);
    buffer = (char **) malloc(sizeof(char*) * num_buffers);
    for (i=0; i < num_buffers; i++)
        buffer[i] = buffer_store + i * buffer_size;

    buffer_status = (int *) malloc (sizeof(int) * num_buffers);
    buffer_used = (size_t *) malloc(sizeof(size_t) * num_buffers);
    buffer_cond_var = (pthread_cond_t *) malloc (sizeof(pthread_cond_t)
        * num_buffers);

    /* Initialize mutex and condition variables */
    fprintf(stderr,"Initializing Mutexes!\n");
    for(i=0; i < num_buffers; i++){
        pthread_mutex_init(&buffer_mutex[i], NULL);
        pthread_cond_init(&buffer_cond_var[i], NULL);
    }


    /* Setup a mutex and flag so the writer thread can
     *   determine when the reader thread is finished.  */
    pthread_mutex_init(&reader_done_mutex,NULL);
    pthread_mutex_lock(&reader_done_mutex);
    reader_done = FALSE;
    pthread_mutex_unlock(&reader_done_mutex);


    /* Initialize the status for all buffers to EMPTY.
     * This code does not need to be locked with mutexes
     * because it is processed before the the threads are
     * started. */
    fprintf(stderr,"Initializing Buffer Status!\n");
    for (i = 0; i < num_buffers; i++) {
        buffer_status[i] = EMPTY;
    }

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    fprintf(stderr,"Opening Input and Output Files!\n");
    src_file = fopen(argv[1],"r");
    if (src_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    dest_file = fopen(argv[2],"w");
    if (dest_file == NULL) {
        perror(NULL);
        return (EXIT_FAILURE);
    }

    fprintf(stderr,"Starting Threads!\n");
    pthread_create(&reader_thread, &attr, reader, (void *)src_file);
    pthread_create(&writer_thread, &attr, writer, (void *)dest_file);
    pthread_attr_destroy(&attr);
    pthread_join(reader_thread, &status);
    pthread_join(writer_thread, &status);
    printf ("Transfer Complete!\n");

    /* Cleanup */
      fclose(src_file);
    fclose(dest_file);

    for (i = 0; i < num_buffers; i++)
        pthread_mutex_destroy(&buffer_mutex[i]);

    free(buffer_mutex);
    free (buffer_status);
    free (buffer);
    free (buffer_store);
    free (buffer_used);
    free (buffer_cond_var);

    pthread_exit(NULL);
    return (EXIT_SUCCESS);
}
```

Listing E.3: Seqprocessor source code (version 3)