8-2009

A Study for Scalable Directory in Parallel File Systems

Yang Wu
Clemson University, wuyang@clemson.edu

A STUDY FOR SCALABLE DIRECTORY IN PARALLEL FILE SYSTEMS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Yang Wu
August 2009

Accepted by:
Dr. Walter B. Ligon, III, Committee Chair
Dr. Adam W. Hoover
Dr. Richard R. Brooks

Abstract

One of the challenges that the design of parallel file system for HPC(High Performance Computing) has to face today is maintaining the scalability to handle the I/O generated by parallel applications that involve accessing directories containing a large number of entries and performing hundreds of thousands of operations per second. Currently, highly concurrent access to large directories is poorly supported in parallel file systems. As a result, it is important to build a scalable directory service for parallel file systems to support efficient concurrent access to larger directories.

In this thesis we demonstrate a scalable directory service designed for parallel file systems(specifically for PVFS) that can achieve high throughput and scalability while minimizing bottlenecks and synchronization overheads. We describe important concepts and goals in scalable directory service design and its implementation in the parallel file system simulator–HECIOS. We also explore the simulation model of MPI programs and the PVFS file system in HECIOS, including the method to verify and validate it. Finally, we test our scalable directory service on HECIOS and analyze the performance and scalability based on the results.

In summary, we demonstrate that our scalable directory service can effectively handle highly concurrent access to large directories in parallel file systems. We are also able to show that our scalable directory service scales well with the number of I/O nodes in the cluster.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 File Systems

A file system is the part of an operating system that deals with the management of files, i.e., how are they structured, named, accessed, used, protected, implemented, and managed. From a user's point of view, as is stated by Tanenbaum in [29], a file system is a collection of files and directories, plus operations on them(e.g., reading and writing files, creating and destroying directories, and moving files among directories).

On the other side, from a designer's point of view, a file system can look quite different. Designers need to be concerned with the implementation of storage allocation, directory structure, and the way that the system keeps track of the relationship between blocks and files. File systems vary in those aspects as well as in the methods they use to improve reliability, performance and scalability. Different file systems have been developed to meet various of requirements from hardware devices, operating systems, data access patterns and users. Here we categorize them into three groups: local file systems, network file systems and parallel file systems, and discuss each of them in the following sections.

1.1.1 Local File Systems

A local file system is a file system designed for the storage of files on a local data storage device, including but not limited to disk storage devices(e.g., floppy disk, hard disk drive, optical disk), tape storage devices and flash memory cards(e.g., multimedia card, USB flash drive, solid-state drive). Local file systems can trace their history back to the early days of computers when paper tape and punch cards were used to store information for automatic processing. Most of those file systems were built into the operating system that they came with and usually did not have a name. Many more advanced local file systems have been developed due to the invention of new storage devices and data access demands. Examples of local file systems include FAT(FAT12, FAT16, FAT32, exFAT), NTFS, HFS and HFS+, Ext(Ext2, Ext3, Ext4), and ZFS [29].

1.1.2 Network File Systems

As the computer network is playing an increasingly important role in modern computing, the corresponding growth of demand to access persistent storage over a network that supports the sharing of files, printers and other resources has impelled the development of network file systems. Most network file systems follow a client/server architecture, i.e., while distributed over the network, the clients and the servers cooperate to form a complete system. Figure 1.1 shows the example of a network file system. The 'Network File System'(NFS) created by Sun Microsystems in 1985 is the first widely used Internet Protocol based network file system. Other popular network file systems include Andrew File System(AFS) [24], NetWare Core Protocol(NCP), and Server Message Block(SMB). Some network protocol clients are also considered as file-system-like(e.g., FTP, WebDAV, SSH).

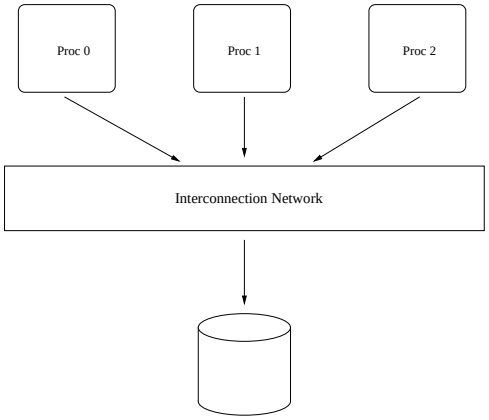

Figure 1.1: Network File System Overview

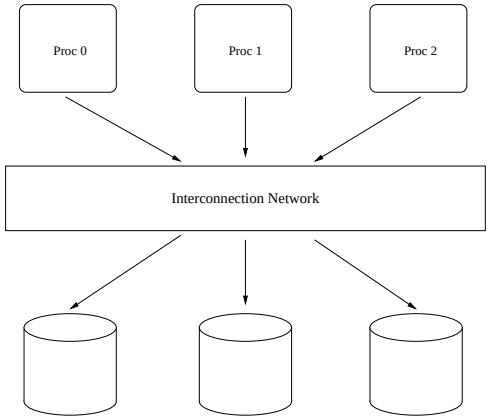

Figure 1.2: Parallel File System Overview

1.1.3 Parallel File Systems

The recent improvement of high-performance computing(HPC) systems has evolved into clusters containing more and more compute nodes which, with the advent of multi-core processors that can perform 10's of giga-flops, enables the executing of applications with tens of thousands of threads running in parallel. For example, the Roadrunner system from Los Alamos National Laboratory with 129600 cores has achieved 1.1 Peta flops in November, 2008 [18]. The growth in computing power imposes significant challenges for the storage systems–the ability to scale so as to handle the I/O requests generated by I/O intensive parallel applications that are commonly seen in both scientific and commercial High Performance Computing(HPC).

Many parallel file systems have been developed to help overcome the I/O bottleneck for parallel applications. By distributing data across multiple processing nodes, each with its own storage resources, a parallel file system can spread the I/O load balance across severale servers rather than bursting the I/O on a single server [4]. Figure 1.2 shows an example of a parallel file system. Not only does the distribution of resources allow the file system to leverage multiple independent storage devices, but also makes more effective use of the bandwidth of the interconnection network whereas the network throughput is not

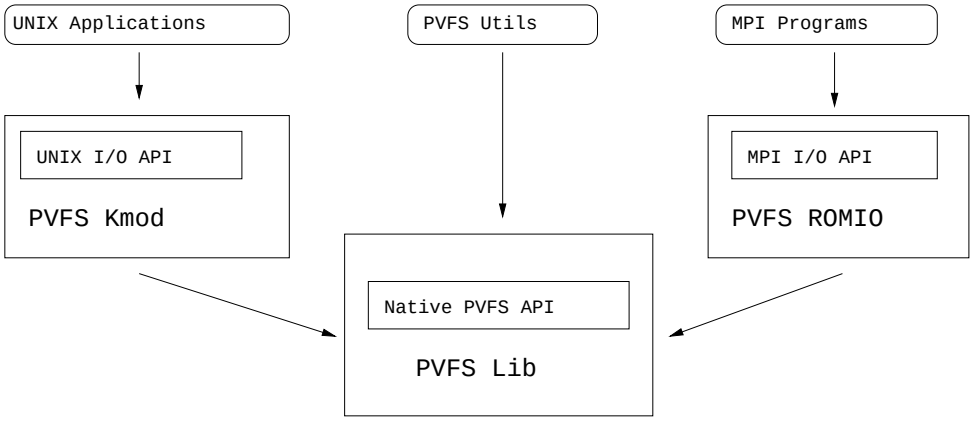

Figure 1.3: PVFS API

constrained by the limitation of the bandwidth of a single link. Examples of parallel file systems include Lustre [3], Google File System(GFS) [11], GPFS [25] and Parallel Virtual File System(PVFS) [5].

1.1.4 Parallel Virtual File System

The Parallel Virtual File System(PVFS) is a parallel cluster file system that is designed to provide high-speed access to file data for parallel applications. It provides a clusterwide consistent namespace, enables user-controlled striping of data across disks on different I/O nodes, and allow existing binaries to operate on PVFS files without the need for recompiling [5]. As is shown in Figure 1.3, PVFS supports multiple APIs: the native PVFS API, the UNIX/POSIX I/O API [1], as well as other APIs such as MPI-IO [12]. Designed as a client/server system, PVFS contains multiple servers that run on separate nodes in the cluster, called the I/O nodes. Each I/O node can perform as a metadata server that uses Berkeley DB [21] database to store metadata, and/or a data server that uses a local file system to store data files.

1.2 Directories

1.2.1 General Directories

To efficiently keep and organize files, a file system often uses directories(folders, catalogs or drawers) to contain files and other directories such that related files are stored in the same directory. The simplest form of a directory system is having one directory containing all the files [29], which is called the single-level directory. The advantage of a single-level directory is its simplicity and the ability to quickly locate a file. However, it is obviously inadequate for organizing thousands of files which is the common scenario for modern file systems. Consequently, a hierarchical directory system is used where a directory can be contained within another directory(called a subdirectory of the containing directory). Together, directories and subdirectories form a hierarchy, or a tree structure, which becomes a powerful tool to represent complicated logical relationships between files. For this reason, most modern file systems use a hierarchical directory system [29].

In many file systems, directories are themselves files. A directory contains references to the files and directories that are located within it. Such a reference is called a directory entry and usually contains the file or directory's name and a link to its content and other attributes or metadata. Though considered files, the operations allowed on directories are different from ordinary files. For example, the UNIX system supports the following directory operations [2]: mkdir, rmdir, opendir, closedir, readdir, rename, link and unlink. A directory can be created(mkdir), removed(rmdir), opened(opendir), closed(closedir) and renamed(rename) just like ordinary files. However, the reading and modifying of the content of a directory are dealt through operations on directory entries like readdir, link and unlink.

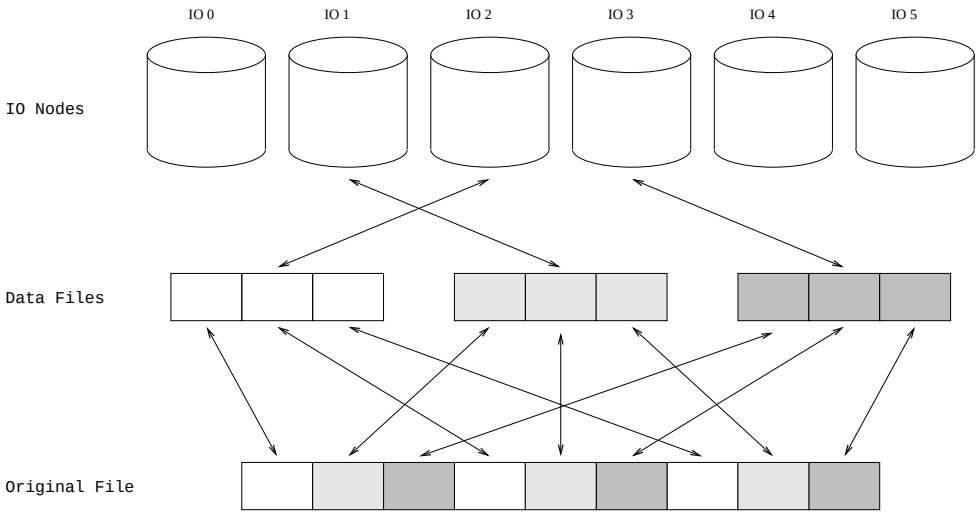

Figure 1.4: File Striping in PVFS

1.2.2 PVFS Directories

Internally, a PVFS server implements storage objects called "dataspaces" that can be combined to represent logical objects like files and directories[17]. For distinguishing them, each one of these objects is assigned a unique id that is called a handle in PVFS. The most important dataspace objects are: metafile object, datafile object, directory object and directory data object. As a virtual file system, PVFS runs on top of a local file system(usually Linux) and uses both Berkeley DB and the underlying file system to store these objects.

- **Metafile objects** represent logical files. Similar to the i-nodes in UNIX file system that contain block numbers of file data, the metafile objects in PVFS store all handles of the datafile objects associated with this particular file. In addition, several attributes stored for a metafile object are: POSIX metadata, datafile distribution, datafile count and datafile handles. Metafile objects are stored in the Berkeley DB databases on metadata servers.

- **Datafile objects** are used to store the raw data of files. Like many other parallel file systems, PVFS supports data distribution of files. Each PVFS file is divided into datafiles and striped across disks on the data servers to facilitate parallel access. Figure 1.4 shows the distribution of a file across multiple I/O nodes.

- **Directory objects** represent logical directories and are similar to metafile objects. They contain the handle of a directory data object which stores the handles of all files within the directory. Other attributes stored in the directory objects are: POSIX metadata, directory entry count and directory hints. Directory objects are stored on metadata servers.

- **Directory data objects** store the "file_name/metafile(directory)_object_handle" pairs to identify all files within the directory that the directory data object is associated with. Directory data objects are stored on metadata servers.

Figure 1.5 shows the location, relation and interaction of these objects in PVFS to provide the representation of logical files and directories. Metadata servers store directory objects, directory data objects and metafile objects in Berkeley DB database and data servers store datafile objects as files in the underlying file system. Each I/O node can be configured to run as a metadata server or a data server or both, across which all types of objects are properly distributed. However, each single object is indivisible thus must reside on a certain server. For example, each directory object has a unique directory data object associated with it and different directory data objects can be stored on different metadata servers. Yet for a certain directory 'foo', the directory data object representing it is located on one of the metadata servers. As a result, unlike logical files, directories can not be distributed in PVFS because of the "one-to-one" mapping between a logical directory and its directory data object design and the indivisible object rule.

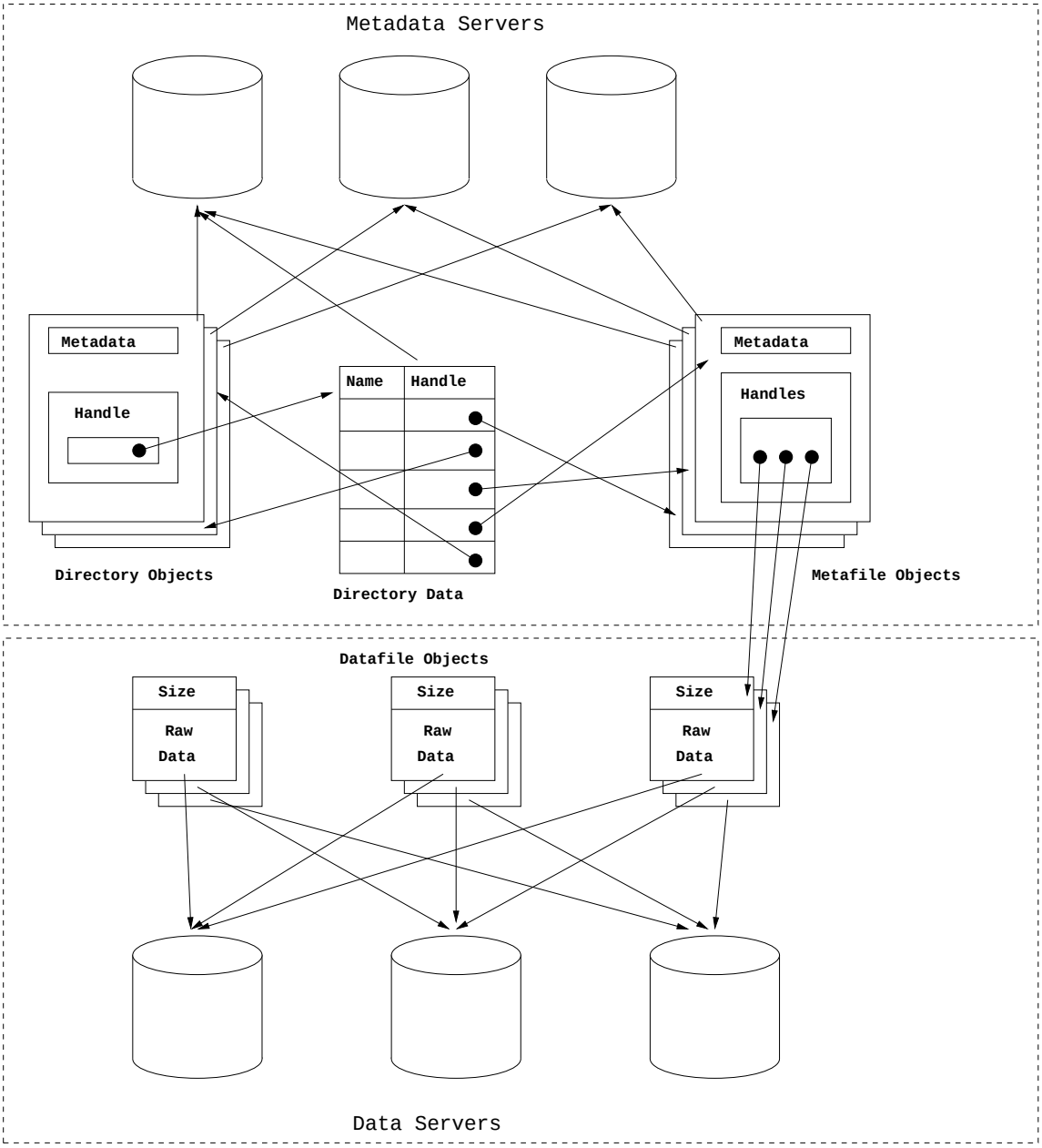

Figure 1.5: Directory in PVFS

1.3 Motivation

One of the challenges that the design of a parallel file system for HPC has to face today is maintaining the scalability to handle the I/O generated by parallel applications with tens of thousands of threads that involve accessing directories containing billions to trillions of entries and performing hundreds of thousands of operations per second in such directories. It is a typical scenario for database applications that do data mining and monitoring applications that deal with a large amount of transactions in real-time(e.g., those used by telecommunication companies to record phone calls or banks to collect financial information from the stock market) to create numerous files in the same directory every second. Other examples like per-process checkpointing in HPC clusters [23] create similar directory access scenario. As a result, it is important to build scalable directory services for parallel file systems to support efficient concurrent access to even larger directories in the future.

Traditional UNIX file systems and FAT use a flat, sequential, on-disk data structure for directory storage and thus have a lookup cost of $O(n)$–the larger the directories are, the worse the access time becomes. To improve performance, newer file systems introduce faster indexing structures like B-trees for $O(\log n)$ lookup cost and hash tables for $O(1)$. For example, XFS uses B+ tree for directory entry index [28] and Linux's Ext2/Ext3 uses hash tables [30]. However, being local file systems prevents them from scaling to more than one machines. Network file systems rely on the underlying local file system that runs on the server to provide directory access, and thus do not achieve extra scalability for handling larger directories.

Fast indexing structures are used in several parallel file systems to gain better performance in handling large directories, including the use of extensible hashing in GPFS [25] and B-link trees in Boxwood [20]. Nevertheless, since GPFS distributes directory entries

in disk block level and Boxwood relies on a global lock service for synchronized access to metadata, they lack the ability to effectively deal with the situation where a certain directory is accessed by many clients simultaneously, which thus becomes a "hotspot" and a potential bottleneck in the whole system.

In PVFS, a logical directory is represented as a directory object and a directory data object, both of which are stored in the Berkeley DB database on a single metadata server. Specifically, directory entries are stored in the directory data objects as key/value pairs, which are accessed by PVFS using the B+ tree method provided by Berkeley DB. Though providing low-cost lookup, insert, delete operations plus efficient sequential access by using B+ tree [8], the design of PVFS is inadequate to scale large directories beyond a single I/O node.

1.4 Thesis Statement

To address the insufficiency of handling highly concurrent access to large directories in the above file systems, we propose a scalable directory design for existing parallel file systems, specifically PVFS in our research. *In this thesis, we demonstrate a scalable distributed directory service and perform a thorough study of its use with a parallel file system so that it fulfills the following design goals:*

- **Maintain PVFS I/O semantics**: PVFS provides three APIs: the native PVFS API, UNIX/POSIX I/O API and MPI-IO. To ease the adoption of existing utilities, applications and MPI programs that are designed for PVFS, we maintain the PVFS I/O semantics so that the changes made in PVFS for the scalable directory service should be transparent to users.

- **Achieve high throughput and scalability**: Under the condition that highly con-

current access causes a large amount of various operations in large directories, the overall system performance should scale with the number of I/O nodes. It is also desired that hundreds of thousands of operations per second can be handed to the directories containing billions to trillions of entries.

- **Minimize bottlenecks and synchronization overheads**: We eliminate bottlenecks by avoiding using centralized locking, looking-up and management mechanism for directory access. Plus, we adopt lazy-invalidation protocols and client-side caching to minimize the "client/server" and "server/server" synchronization overheads.

1.5 Approach

The studies of scalable directory service is performed using a parallel file system simulator–the High-End Computing I/O Simulator(HECIOS). HECIOS [26] is a trace-driven parallel file system simulator built upon the discrete event simulation libraries provided by OMNeT++. It closely simulates the network messages and system calls used by the Parallel Virtual File System(PVFS) to perform file system tasks. During the simulation, HECIOS parses the trace files, processes the application requests into parallel file system requests and accordingly performs parallel file I/Os. Although the parallel file system models perform most of the simulation work, most of the simulated time accounting is performed by the physical device simulation including the network models, operating system models and disk models. In our study, those models are validated to accurately simulate the performance of PVFS on Palmetto–the HPC cluster in Clemson University.

Based on the simulation models in HECIOS, we have designed and implemented our scalable directory service for the parallel file system simulated by HECIOS. By distributing directory entries within large directories across multiple metadata servers in PVFS,

the workloads on such directories are balanced among servers. Two approaches of distributing directory entries are implemented in the scalable directory service: pre-distribution and dynamically splitting. Each of the approaches was tested and benchmarked with the I/O traces that we have collected by running MPI programs that perform directory entry operations in PVFS on Palmetto. The experiment results of both approaches were compared to demonstrate the impact of their differences on the system performance.

To analyze the performance of our scalable directory service including the throughput, scalability, directory growth rate and synchronization overheads, a series of experiments were performed by running MPI traces with various numbers of clients and servers, as well as different scalable directory settings. We also developed a set of tools to automate the experiments and to process and visualize the performance data.

1.6 Thesis Organization

In Chapter 2 we present an overview of the related research projects in the field of parallel file systems and scalable directory. Chapter 3 contains a detailed description of the research that we conducted in scalable directory design. Chapter 4 contains a thorough discussion of the simulator that we used for this study, its software component model and the implementation of scalable directory within the simulator, plus the methods we used to verify and validate the simulation model. In Chapter 5 we present the performance results of our scalable directory design including the throughput and scalability data. Finally, in Chapter 6 we briefly summarize our experimental results, describe the conclusions and implications of this study, and discuss some suggestions for future work based on our work.

Chapter 2

Related Work

2.1 File Systems

Much effort has been taken by various file system projects to address the I/O bottlenecks that are faced in HPC nowadays. To improve the performance and scalability of I/O operations(specifically large directory and concurrent access), researches are conducted in finding faster indexing data structure and file system architecture.

To avoid the linear scan of directory blocks in searching for a particular file, which is required by the directory structures used by many traditional file systems(e.g., Ext, FAT), file systems like Episode [7] and VxFS [27] speed up searching for entries within a directory block via hashing. Different schemes including in-memory and on-disk structures are also discussed [16]. Newer file systems like NTFS [10] and Cedar [13] start to use B trees to index the entries in the directory.

XFS [28] file system is a general purpose local file system for Silicon Graphics' IRIX operating system that focuses on scaling capacity and performance in supporting very large file systems, including large files, large number of files, large directories, and very high performance I/O. To support large directories, XFS uses an on-disk B+ tree structure

to store directories. To reduce the height for B+ trees, XFS hashes the file names in variable length to four byte values and keeps entries with duplicate hash value next to each other in the tree. The use of fixed size keys increases the breadth of the directory B+ tree thus improves the performance of lookup, create and remove operations in large directories.

Extensible hashing is used by GPFS [25] to support efficient file name lookup in large directories. In GPFS, each directory is represented by a various number of directory blocks, depending on the directory size. A hash function is applied to the name of the files within the directory and certain bits of the hashing value are used to decide the block number that the entry should reside in. If a block becomes full as new entries are added to different blocks, it gets split into two blocks and the hashing bits are updated accordingly. The extensible hashing method can make sure that operations in the directory only require the scanning of a single directory block so that the cost for operation is independent from directory size and directory is locally scalable. A more detailed description of extensible hashing and its use in our scalable directory design is given in Chapter 3.

2.2 HECIOS

The High-End Computing I/O Simulator(HECIOS) [22] is a trace-driven parallel file system simulator built upon the discrete event simulation libraries provided by OMNeT++ [9]. In this research we design and implement the scalable directory using the simulated model of a parallel file system(i.e., PVFS) in HECIOS that simulates the network messages and system calls required by PVFS to perform file system tasks. Figure 2.1 is a systematic overview of the major components in HECIOS. Above the INET network component, on the left are the client-side components and on the right are the server-side ones. By running trace files and simulating I/O activities in the simulator, the I/O performance statistics data are collected for further analysis.

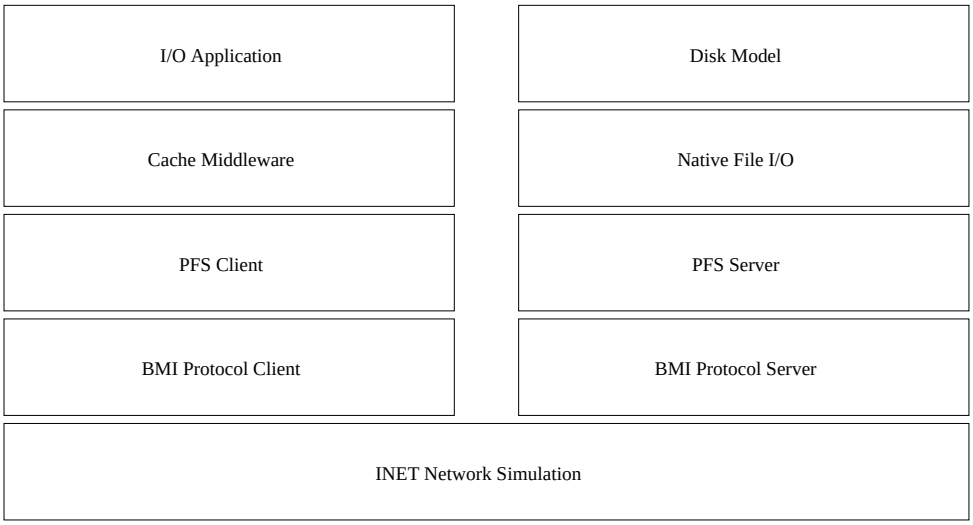

Figure 2.1: HECIOS Architecture

- **I/O Application** simulates the UNIX applications and MPI programs that generate I/O requests served by PVFS. It initiates I/O activities in the simulator by translating both UNIX system and MPI collective file I/O calls contained in the corresponding trace files into proper parallel file system messages and processing the responses accordingly.

- **Cache Middleware** is the module to provide client caching semantics. By storing an active subset of the file data and metadata in the compute node's main memory, it avoids contacting the parallel file system to access file and thus improves performance.

- **Native File I/O** is used for local file access since the data files in PVFS are stored as normal files in the local operating system that runs on the I/O nodes. It serves the local I/O requests by interacting with the disk model that simulates generic storage devices. The Berkeley DB operations required by metadata access are also simulated by performing certain local file I/Os.

- **PFS Client** simulates the PVFS client. It accepts PVFS I/O primitives and generates PVFS network messages that represent the primitives. Those messages are in turn sent to the PVFS servers.

- **PFS Server** simulates the PVFS server that provides both data and metadata access. It receives PVFS requests from PFS client and translates them into local file I/Os. Corresponding response is sent back to PFS client after serving each of the requests.

- **BMI Protocol Client and Server**, along with the INET Network Simulation component, simulate the network facility of PVFS. BMI is the interface used by both the clients and servers to communicate with each other in PVFS. Default flow protocol is also built on BMI.

- **INET Network Simulation** uses the INET network simulation framework that is developed for the OMNeT++ simulation packages to simulate the interconnection network connecting the compute nodes and I/O nodes in the cluster. It provides detailed network simulation models like TCP/IP and Ethernet.

- **Disk Model** uses a simplified drive geometry model that accounts for typical hard disk parameters such as head movement speeds, spindle speeds, and physical inertia times, but does not take into account more complicated parameters such as disk prefetching and speculative block accesses.

A set of tools are developed to convert the output from readily available parallel and serial application traces into the trace formats that follow the specifications defined by HECIOS. The raw trace files that are used in this research are generated by using the publicly available tracing tools from Los Alamos National Laboratory with certain parallel applications running on Palmetto, Clemson University's large cluster computer available for research use.

2.3 GIGA+

GIGA+ [23] is a scalable directory service that aims to scale and parallelize metadata operations in parallel file systems. GIGA+ divides large directories into a scalable number of fixed-size partitions that are distributed across multiple servers in the cluster by employing an extensible hashing scheme to achieve load balance across all servers. A single metadata server may hold one or more partitions of a directory. The maximum possible number of partitions equals to the number of meta data servers. A server list is used to record metadata servers that a directory splits to, which is unique for each directory. A Partition-to-Server(P2S) map is used to record the presence of a directory partition on a server where "1" indicates presence and "0" for absence. Along with the server list, a P2S map can be used to easily decide whether a directory is split to a certain server. Finally, a hash function is used to decide location of each directory entries among directory partitions.

Take lookup operation for example–when a client wants to access a certain file "bar" in the directory "foo", it follows this procedure to lookup the directory entry:

1. Get the P2S map for the directory "foo"
2. Let $i = Hash(\text{"bar"})$
3. While $P2S[i]! = 1$, let $i = \lfloor i/2 \rfloor$. Figure 2.2 describes this step.
4. Finally, $N = i$
5. Contact metadta server S_N for the directory entry "bar"

As new entries get added, a directory file is split when it becomes too large by moving half of the partitions that currently belong to it onto another server. The destination server number is determined by the following procedure.

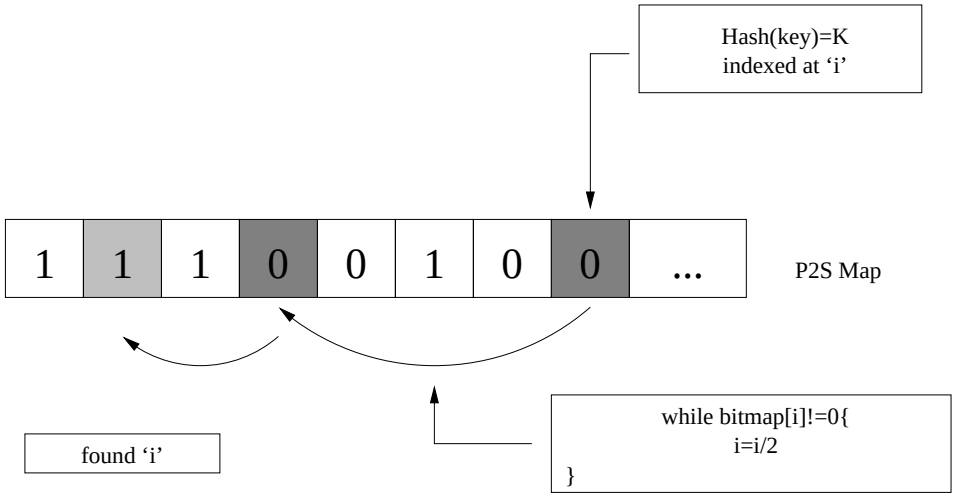

Figure 2.2: Lookup Directory Entry in GIGA+

1. Found the P2S map for the directory
2. Let k equals to the index of the leftmost non-zero bit of the source server number, so that $ID[k] = 1, ID[j] = 0 for j > k$
3. Let l equals to the number of splittings of this directory that took place on the source server
4. Let $ID[k + l + 1] = 1$, ID is the destination server number
5. If server ID is valid, perform splitting and update P2S map: $P2S[ID] = 1$

Figure 2.3 shows the above process in a splitting tree. Note that in the figure, splitting in the same level are independent from each other(e.g., "B" and "C" are independent), however splitting in a lower level can not happen until the splitting that leads to its parent in the tree finishes(e.g., "F" can not happen before "C"). Likewise, directory emerges in a reverse order when entries are removed.

In GIGA+ P2S maps are updated in an asynchronous manner for the clients and servers to minimize synchronization cost. As a result, it is likely that the P2S map on a

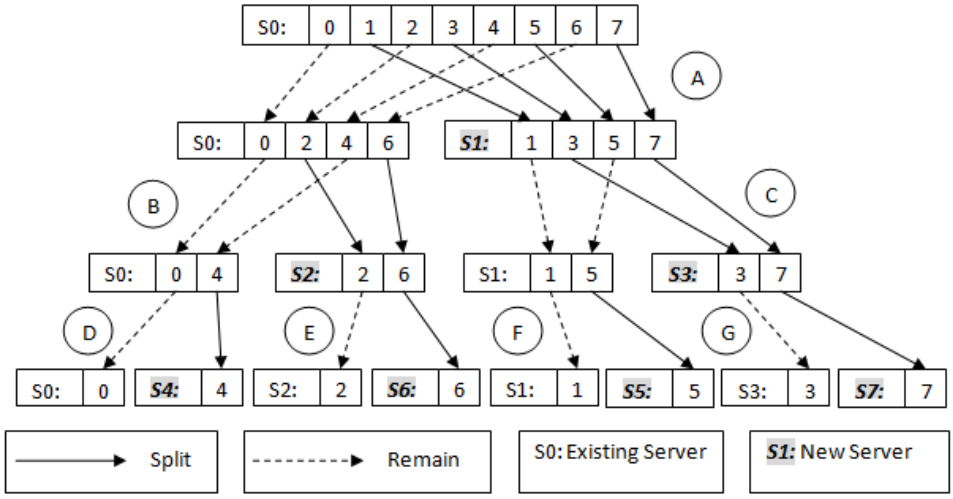

Figure 2.3: Split Directory in GIGA+

client is outdated in the sense that the client is not aware of a recent directory splitting and requests for a directory entry from the server that no longer hold it. In case this happens, the server forwards the request following the splitting history until the request reaches the server that hold the very directory entry in need. The worst case forward cost is logarithmic to the number of metadata servers.

A user level implementation of GIGA+ that adopts a "client/server" architecture is described in [15]. Built on the FUSE library, it supports scalable directory services including but not limited to splitting, nested huge directories, small directories, concurrency control and client caching. The design of scalable directory service for parallel file systems in this research is greatly inspired by the principle of GIGA+ in the use of extensible hashing scheme, P2S map, directory splitting and message forwarding.

Chapter 3

Research and Design

In this chapter, we will discuss the research and design issues regarding the scalable directory service of PVFS. The primary design goal is to efficiently distribute the directory files of large directories across multiple metadata servers to improve scalability and performance. Figure 3.1 shows the design of distributed directory in PVFS, in which a single directory data object that contains directory entries is divided and scattered onto a number of metadata servers.

This design shrinks the size of the directory data objects associated with large directories on a single server, hence leads to a shorter linear scan operation cost when looking up a directory entry within a directory data object. More importantly, it balances the load of the servers and makes better use of network bandwidth in case of highly concurrent access to a single directory, thus provides more parallelism. To achieve scalability, an extensible hashing scheme is used for the clients to locate a directory entry among the metadata servers. This hashing scheme is carefully designed to be flexible so that it accordingly adjusts itself to work with the metadata servers, the number of which can dynamically change in runtime due to the change of directory size. As a result, an $O(1)$ locating operation cost can be maintained as the number of metadata servers in use grows and therefore the size of

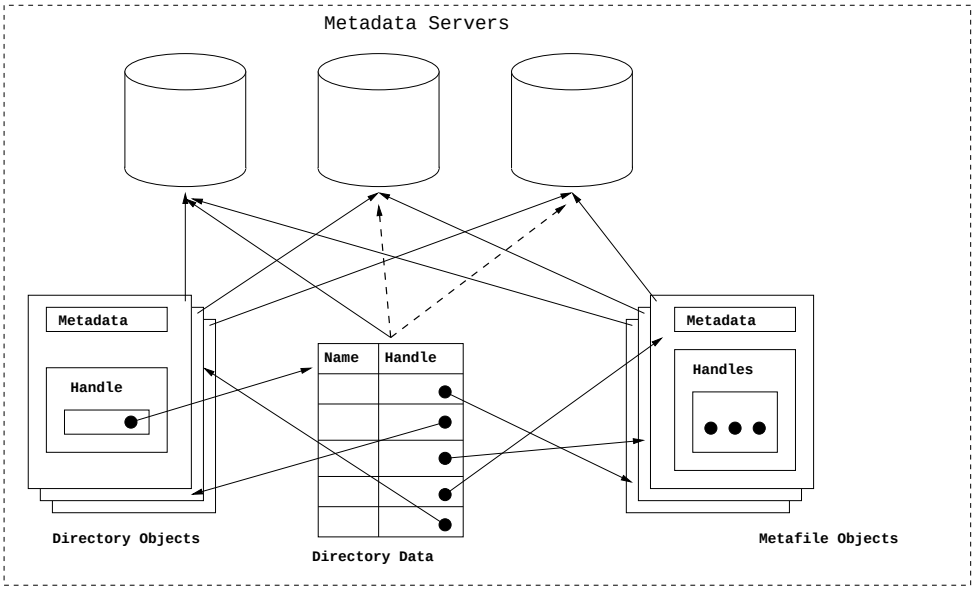

Figure 3.1: Distributed Directory in PVFS

directories scales with the number of metadata servers.

The synchronization overheads are minimized in this design in three ways: firstly, a client-side caching mechanism is used to cache the directory objects as well as other useful information to avoid repeated visits to the same metadata servers; secondly, the splitting of large directories follows a certain pattern so that metadata servers can keep track of a splitting history and thus directory entries can be easily located without heavy communications among metadata servers; finally, lazy-invalidation protocols are adopted where metadata servers defer the invalidation of the location information of directory data objects until clients explicitly access them and invalidation is necessary.

3.1 Server List

In PVFS, each logical directory has a directory object associated with it. When a directory is created, the new directory object will be put onto different metadata servers

in a round-robin manner so that directory objects are equally spread among the metadata servers in the cluster. The server where the directory object resides is called the "Home Server" of the corresponding logical directory, which is a very important concept for the design of scalable directory service. The directory data object that contains the directory entries within the logical directory is also located at the corresponding home server. Each of the objects in PVFS(e.g., directory objects, directory data objects, metafile objects, and datafile objects) has a handle(a 64 bits integer) that is unique in the system to identify itself. Accordingly, each server(metadata server and data server) in PVFS holds several ranges of objects within the system. As a result, we can simply look up the server range list to determine the location of a certain PVFS object, where Loc is the id of the server that holds it.

In the scalable directory service design, we construct a server list for each logical directory so that the extensible hashing scheme can refer to it for location. The server list of a directory is a linear array in which each element is the id of a metadata server and the index is its virtual id as regard to the very directory. The first element in the server list is always the home server of the directory, followed by the other metadata servers in clockwise order according to their ids.

For example, consider a cluster with eight I/O servers that runs PVFS, in which Server0, 1, 4, 5 and 7 are metadata servers as shown in Table 3.1.

I/O Server ID	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Metadata Server	x	x			x	x		x

Table 3.1: Example: A Cluster Running PVFS

Three directories "foo1", "foo2" and "foo3" respectively reside on Server0, 1 and 5. Table 3.2 shows the server list for each of those directories.

Note that though the server list of a directory can be easily generated from its home server with the knowledge of the metadata servers in cluster(by sorting the metadata servers

Directory		Server List				
Name	*Home Server*	*0*	*1*	*2*	*3*	*4*
foo1	0	0	1	4	5	7
foo2	1	1	4	5	7	0
foo3	5	5	7	0	1	4

Table 3.2: Example: Server List for the Directories in the above Cluster

in clockwise order starting from the home server), it is necessary to maintain a server list for every distributed directory so that changes in cluster configuration like the adding of new metadata servers can be handled seamlessly.

3.2 P2S Map and Extensible Hashing

To locate a directory entry among the metadata servers with small cost, a hashing scheme that combines the use of P2S map and extensible hashing is adopted in our scalable directory service design. Like in GIGA+, a P2S map stands for a Partition-to-Server map which stores its owner's knowledge of the distribution of a certain directory across the metadata servers. In our design, a P2S map is a bitmap in which each bit indicates the presence of a directory on the corresponding metadata server. Take the previous cluster configuration as an example–if the P2S map of directory "foo2" is "11010"(lower bit first), it is known that "foo2" is scattered onto Server1, 4, and 7 as is shown in Figure 3.2 by comparing the P2S map against the server list.

On the other side, the hash value of the directory entry name decides the location of a certain directory entry. After applying a pre-chosen hash function(e.g., MD5) to the considered directory entry name, the lower n bits($n = \lceil \log_2 N \rceil$, where N is the number of metadata servers) of the hash value is taken as the target virtual server id. This id can then be used with the P2S map and/or server list to perform various operations like lookup,

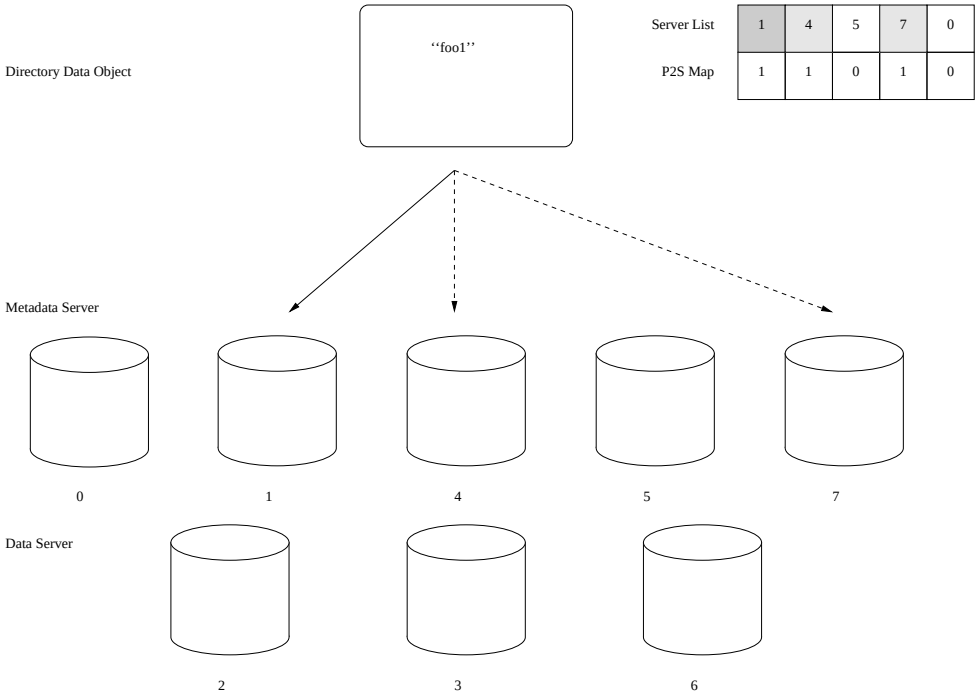

Figure 3.2: P2S Map and Distributed Directory

splitting, request forward and directory traversal.

$$Virtual\ ID = Hash(Entry\ Name) AND_{bitwise} (2^n - 1) \tag{3.1}$$

3.3 Client Lookup

As is described in Section 1.2, the design of PVFS directory consists of directory objects and directory data objects. The former contains the metadata of the directory that it refers to and the handle of the corresponding directory data object, which contains directory entries. To lookup a directory entry, the client firstly looks for the directory object of the root directory "/", the handle of which is well-known in the system. After obtaining the directory object of "/" and the contained handle of directory data object, the client requests

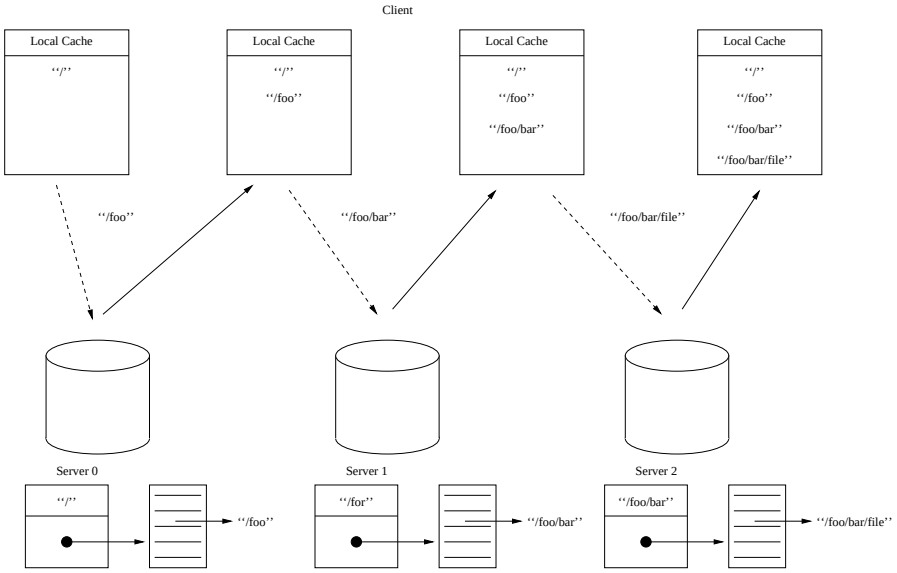

Figure 3.3: Lookup Operation in PVFS

for the corresponding directory data object and looks for the entry of the second level directory in the target path. Then the handle of the second level directory object is obtained, which may locate on another metadata server. Similarly, the client looks for the third level directory, the forth level, ..., until the desired path is reached.

Figure 3.3 shows the process of looking up the directory entry "/foo/bar/file"(i.e., to obtain the handle of its directory object) in PVFS, where the dashed arrows represent requests from client and the solid arrows represent responses from servers. The client determines which server to contact and requests for the entry name on that server. The server then responses with the handle of the directory entry to the client. Note that after the lookup operation, the handles of all the directories, each being part of the path, are cached by the client for future use.

When putting distributed directory into consideration, the target server can no longer be simply determined by looking up the the object handle in the server range list, which only indicates the home server for that directory. While the communication pattern is the

same as the lookup operation in the original PVFS design, it is required that the P2S map is used with virtual server id to determine the location of directory entries as in the following procedure before sending out requests from clients in the scalable directory design. Take the request for the directory entry "/foo/bar" as an example:

1. Now we have the handle and P2S map of the directory object "/foo": H_{foo} and $P2S_{foo}$.
2. The home server S_{home} for "/foo" is determined by looking up the server range list.
3. A server list can be constructed from S_{home}.
4. By using Equation 3.1, we can get the virtual server id for the directory entry "/foo/bar": id.
5. Check to see if the idth bit of the bitmap $P2S_{foo}$ is non-zero. If not, clear the highest non-zero bit of id and check again. Repeat this step until the idth bit of $P2S_{foo}$ is "1".
6. The idth element in the server list is the target server id.

The Step 5 in the above procedure involves repeatedly checking for non-zero bit in a P2S map when varying the virtual server id. Table 3.3 shows the process of varying virtual server id from 1 to 7. By clearing the highest non-zero bit(the underlined bit) at each round, it requires at most $\lfloor \log_2 N \rfloor$ rounds(where N is the number of metadata servers) to reach a non-zero bit in the P2S map(bit0 represents the home server and is thus always non-zero).

Virtual Server ID	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
*1*st round	**0(000)**	**0(000)**	1(001)	**0(000)**	1(001)	2(010)	3(011)
*2*nd round			**0(000)**		**0(000)**	**0(000)**	1(001)
*3*rd round							**0(000)**

Table 3.3: Example: Lookup Table for Virtual Server ID 1 to 7

3.4 Directory Splitting

Like GIGA+, we use a scalable number of fixed-size partitions to divide directory entries into multiple groups and assigns different partitions to different servers when splitting directories. The number of partitions equals to the number of metadata servers in the beginning, which simplifies the splitting operation so that it is as easy as to move partitions from one server to another. On the other hand, it should also be adaptive to changes of metadata server configuration such as the adding of new metadata servers into the cluster. That is to say, the directory service should make use of the newly added servers immediately and seamlessly in that case. This requirement is fulfilled by allowing scaling the number of partitions with the number of metadata servers.

Take the insertion of a new directory entry for example: when a client inserts a directory entry to a directory, the client firstly looks up the server to contact following the procedure in Section 3.3. On receiving the request for creating a new directory entry, the server calculates the virtual server id for the entry by using Equation 3.1. This id is then used as the partition number for the very entry. As a result, each directory entry has a partition number with it. It is not rare that multiple partitions that share the l identical lower bits(where l equals to the number of splittings of the directory starting at 0) reside on the same server. As the number of directory entries on a single server keeps accumulating, the server can initiate the splitting operation for better performance when that number reaches a pre-defined value. This is done by comparing the $l + 1$ lower bits of the partition number of every partition: the partitions with a value of $l + 1$ lower bits that is different from the

virtual server id of the current server are moved to a new server with a virtual server id of that value if this server exists. If such a server does not exist, the directory is flagged as "inseparable" and will not try to split itself again in the future.

For directories that are known to be large before creation, it is possible to pass a hint(a parameter passed from clients to servers that has a special meaning to the parallel file system if the servers support it) to PVFS so that those directories can be split when they are created. Depending on how far should the directory distribute, proper P2S map of the directory is set on both the client and the servers to hold the directory. Consequently, all following directory entry creations are aware of the directory distribution and thus the entries are automatically distributed among the servers in use. Moreover, already distributed directories can be further distributed to more servers if any are available. It is important that prior distribution follows the rules of the splitting tree(Figure 2.3) in Section 2.3.

When new metadata servers are added into the cluster(e.g., m servers are added to N servers), there are two cases:

- if $\lceil \log_2 (N+m) \rceil = \lceil \log_2 N \rceil$, the number of partitions stays the same, however, directories that are flagged as "inseparable" are tested again to see if newly added servers match their potential destinations. Thus valid splitting operations can be performed. Plus, newly created directories can choose these new servers as their home server. Therefore, new servers are made use of properly.

- if $\lceil \log_2 (N+m) \rceil > \lceil \log_2 N \rceil$, it is required that $n = \lceil \log_2 (N+m) \rceil$ gets plugged into Equation 3.1 and the partition id of each directory entry in large directories on each server is recalculated with the new parameter. After that, the checking for splitting can be performed to move partitions to the newly added servers and the other operations are similar to the above case.

In conclusion, the directory splitting operation is carefully design so that:

1. Small directories are not split.

2. Large directories are distributed automatically or dynamically.

3. The directory service is aware of metadata server number change and always adapts to it.

3.5 Request Forward and Response

Though P2S map plays a significant role in the scalable directory service, we update and synchronize P2S maps in a lazy manner to minimize communication and synchronization overheads. It is not plausible to actively update all the clients because broadcast is a costly network operation, especially when the client number is large(which is usually the case) and many of them are trying to broadcast concurrently, which leads to nothing but network congestion. Hence in our scalable directory service design, the updating of P2S map is deferred until it is necessary and convenient.

In PVFS, communications can be between a client and a server, or two clients, or two servers. These communication follow a "request/response" pattern, in which it is allowed for a client to send requests to a server while a server is not allowed to send requests to a client. In another word, servers can talk to clients only when they are being contacted by clients. The purpose of such design is to reduce the complexity of the communication layer thus to simplify the implementation. As a result, it is only convenient to provide the updated P2S map to clients in the response from servers. The updated P2S map is synchronized across the client that initiated the request and the servers involved in processing the request while the other clients and servers are not aware of this update thus their P2S maps are outdated.

Figure 3.4 shows this process in a cluster consists of six clients and six I/O servers,

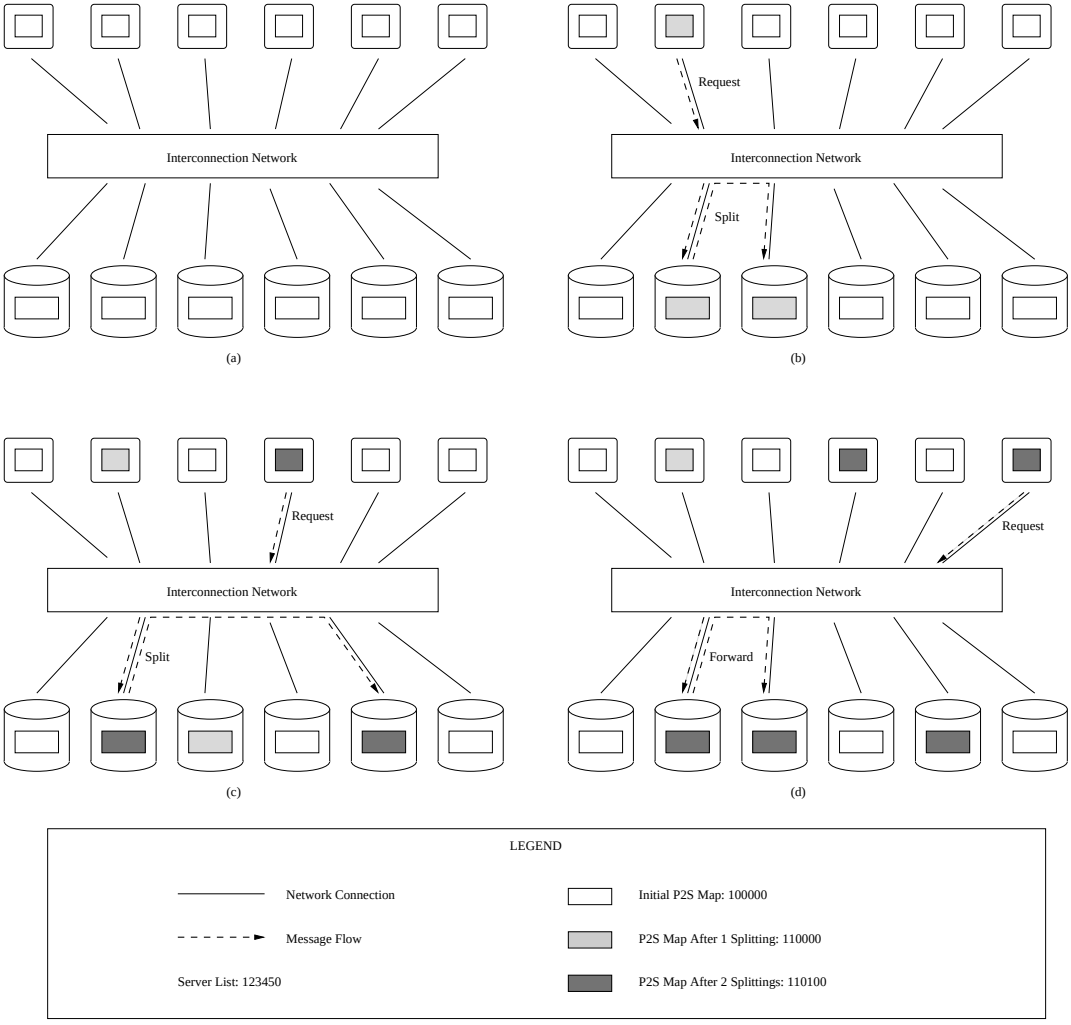

Figure 3.4: Lazy Update and Request Forward in PVFS

where the server list for a particular directory is "123450". In image (a), all the clients and servers hold the same P2S map(100000) which indicates that the directory exists in its home server–Server1. In image (b), a client sends a "create directory entry" request to Server1, which splits the directory to Server2 on completing the creation. Then an updated P2S map(110000) that indicates the presence of the directory on Server1 and 2 is shared between the two servers involved and sent to the client in the response. Note that during this time, the other clients and servers hold an outdated P2S map. Similarly, in image (c), another client sends a "create directory entry" request to Server1 and another splitting distributes the directory to Server4 and produces a new P2S map(110100). As a result, the client and Server1, 4 now hold the latest P2S map, leaving the maps on the other clients and servers unchanged.

It may appear to be inconsistent for different clients and servers to hold different versions of P2S map of the same directory, many of which are outdated. However, this actually will not lead to incorrectness or inconsistency from the file system's point of view due to the well-defined directory splitting path and the request forwarding mechanism. First of all, each of the metadata servers maintains a splitting history of itself and it is ensured that of the servers in the splitting history either is holding a particular directory entry or once held it and knows where it goes to(one server in its splitting history). This guarantees that by recursively contacting the most likely destination, the clients and the servers can eventually locate a directory entry. Second of all, upon locating the target metadata server that holds the directory entry, the request from the client is forwarded to that server as well, which is then served by that server. We favor server-to-server message forwarding over multiple client-to-server "request/response" sessions during the searching for a directory entry because in many cases, the server-to-server communications in clusters are carried out by specific hardware devices and thus are much faster and more efficient than client-to-server communications.

In image (d) of Figure 3.4, a client that holds the outdated P2S map(has no knowledge of Server2) sends a request to Server1 to visit a directory entry that is actually located on Server2 now. On receiving the request, Server1 calculates the virtual id the directory entry and decides that Server2 might hold it(construction of splitting history). So Server1 forward the request to Server2 and Server2 repeats what Server1 has done. If Server2 holds the entry, which is the case in Figure 3.4, it will generate response and send it back to Server 1, which will in turn send it back to the client with an updated P2S map. If Server2 has split the directory and does not hold the entry, it will forward the request to another server and so on until the entry is found(or all possible servers are contacted and the entry doesn't exist). Finally, the client and the servers involved will have a P2S map that is the most updated to their knowledge.

In conclusion, in our scalable directory service design, the P2S maps in the cluster can be modified and updated concurrently in a distributed way. Consequently the communication and synchronization overheads are significantly reduced while the correctness can be maintained.

3.6 Directory Traversal

Many times, clients want to read the content of a directory as whole, namely all the entries within the directory. For small directories that reside on a single server, this simply involves the client requesting the directory and the server that holds it responding with a list of entries of that directory. However, things become more complicated for large directories that are distributed across multiple metadata servers. An intuitive way of doing this is for the clients to contact every metadata server that is in use as is indicated by the P2S map. However, certain drawbacks prevent it from being practical. First of all, the information retrieved by using this method could be incomplete because the P2S map on the client can

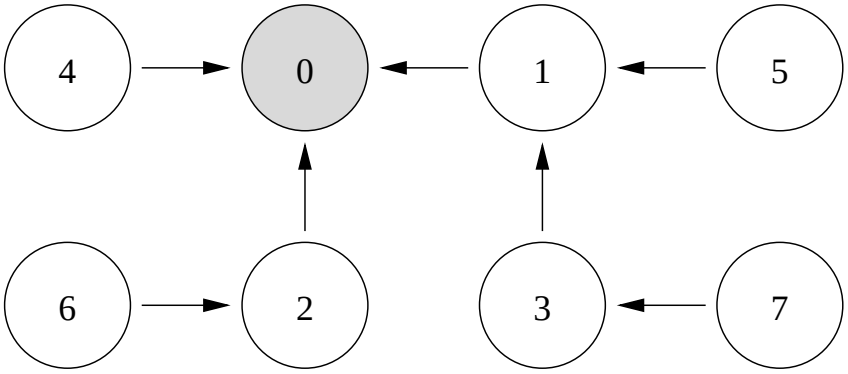

Figure 3.5: Directory Entry Collecting in PVFS

be outdated and thus does not contains all the servers where the directory resides. Second of all, client-to-server communication is not favored because it can be slower than server-to-server communication in many cases. So in our design, directory traversal is performed on server-side.

Whenever a client is to initiate a "read directory" operation, it sends the request to the home server of that directory, which will then take charge of the directory traversal. Similar to the "request forwarding" operation, the home server will perform an exhaustive traversal in the directory splitting tree–sending requests to the servers in the P2S map and collecting directory entries from them and the servers in their P2S maps. Figure 3.5 shows the process of gathering directory entries to the home server of that directory, forming a "hypercube" communication pattern. The communication pattern is like this assuming that the P2S map on each server is not synchronized with others so that every server only has knowledge of its direct children and contacts them. However, improvement of the communication pattern can made based on the network topology of the cluster and the information from updated P2S map. After gathered all the directory entries, the home server can generate the response that contains the entries and the latest P2S map and send it back to the client. As a side-effect, every server involved will end up with a P2S map that

contains the latest updates from its children.

Chapter 4

Implementation and Methodology

In this chapter, we will describe the implementation of our scalable directory service design in the parallel file system simulator–HECIOS, which simulates the system architecture and the behavior of each component in PVFS. We want the scalable directory service to be seamlessly integrated into HECIOS and to properly cooperate with other components in HECIOS. More importantly, the changes made to PVFS should be transparent to users so that PVFS I/O semantics can be maintained. A description of the methodology used for benchmarking and validating the simulation model is also included.

4.1 HECIOS

4.1.1 Software Layers

As a parallel file system simulator specifically developed to simulate PVFS, the software layers of HECIOS are designed to be similar to that of PVFS. As is shown in Figure 4.1, each layer implements the functionality of its corresponding PVFS software layer in HECIOS as well as the interfaces to interact used with other layers. Built on OMNeT++, HECIOS adopts OMNeT++ models, messages and finite state machines to

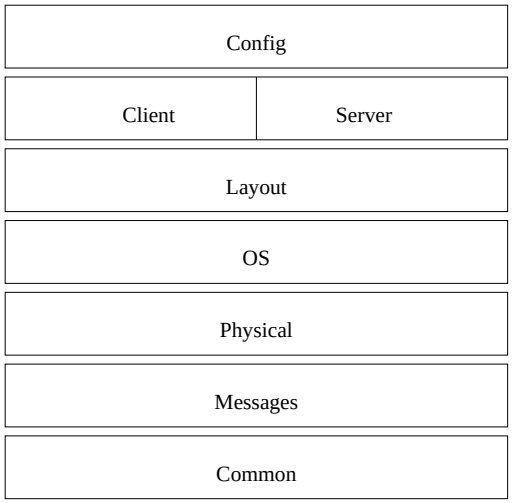

Figure 4.1: HECIOS Software Layers

construct discrete event networks that represent complicated systems, namely PVFS. In implementing our design, the scalable directory service is distributed into different layers for the purpose of seamless integration.

- **Config** layer contains the models that represent the configuration of the PVFS system being simulated, including but not limited to the models of computing nodes and I/O nodes, networking environment, operating system, MPI software and storage devices. Along with the parameters given in configuration files, these models can then be used to represent various PVFS systems.

- **Client and Server** layer contains the models that represent the client and server components of PVFS, as well as the finite state machines used to describe the operations supported by them. Driven by different messages within the system, these state machines can describe complicated operations and are adaptive to changes.

- **Layout** layer contains functions and data types that deal with data striping in PVFS, supporting various distribution schemes and data types. A helper class(File Builder)

that eases the management of files in HECIOS is also included.

- **OS** layer contains models and finite state machines that simulate the operating system that hosts the server components on I/O nodes. Major supported operations are system calls that are involved in performing I/Os and interacting with storage devices. A local file system is also simulated by OS layer.

- **Physical** layer contains models of physical devices(e.g., hard disks) and networking interfaces that deal with the network protocols in BMI and MPI.

- **Messages** layer defines the message prototypes that are used by different components to interact with each other. Included prototypes are: BMI message prototype, cache message prototype, MPI middleware message prototype, MPI message prototype, network message prototype, operating system message prototype and PVFS message prototype. All the activities occur in HECIOS are represented by certain types of messages.

- **Common** layer contains basic functions and data types that are shared by other layers. Such functions include but are not limited to accessing trace files, processing MPI data types and managing collective communications.

4.1.2 Components

There are two types of components in HECIOS: simple component and module component. While the former represents simple objects that could not be further divided, the latter represents compound objects that consist of a group of simple components and module components. The whole system is a special module component that forms a network, which contains other simple components and module components and thus forms an

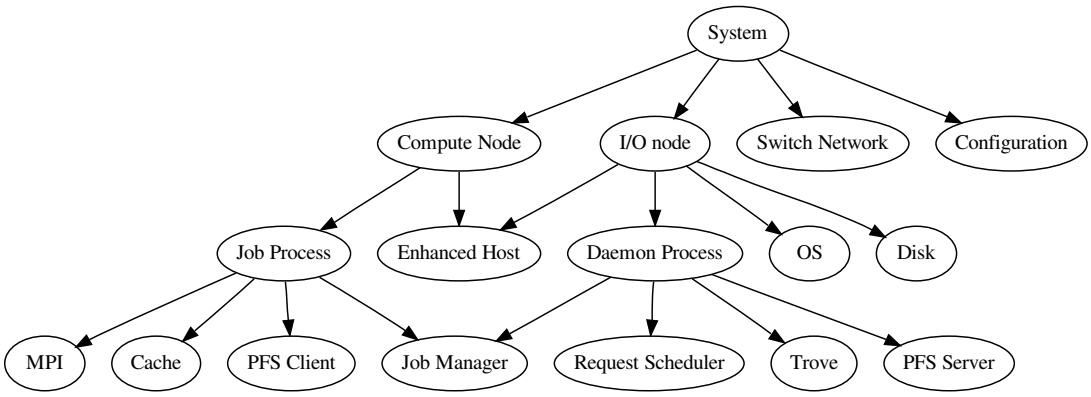

Figure 4.2: HECIOS Objects Hierarchy

objects hierarchy as is shown in Figure 4.2. We can see from the figure that client components run on compute nodes and server components run on I/O nodes. Both the compute node and server node have an enhanced host that deals with network communication related issues including IP address, routing, sockets, TCP/IP protocols and etc. The daemon process has a trove component, which is a unified interface for PVFS to access disk files and database in the same way.

Figure 4.3 shows a configuration of HECIOS that simulates a PVFS system which contains 8 compute nodes and 4 I/O nodes connected by a switch network running in the simulator GUI. The arrows in the figure represent links between components that deliver messages(e.g., API messages, network messages). Operations like system calls and network communication can be simulated by passing specific messages across components. There are also 3 isolated components that hold configurations for parallel file system, MPI and network, respectively.

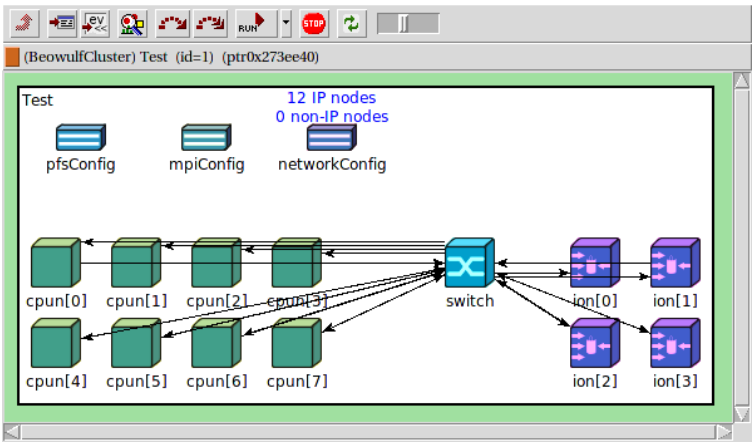

Figure 4.3: HECIOS Network

4.2 Scalable Directory

Our scalable directory service is implemented by mainly modifying three components: the configuration component, client component and server component. New data structures, functions and finite state machines are added into these components to integrate scalable directory service into HECIOS. The service is also designed to be optional so that it can be easily enabled or disabled.

4.2.1 Configurations

The configuration component contains parameters that control the behavior of the parallel file system, MPI and network. We add new options to the parallel file system configuration for the scalable directory service as is shown in Table 4.1. New data types that hold P2S maps are also added to the directory metadata structure. The trace file format is changed to allow the definition of the distribution of directory entries as well.

Module	Option	Description
PFS Config	useDistDEnt	Globally enable scalable directory service
PFS Config	dEntCapacity	Number of directory triggers directory splitting
PFS Client	useDistDEnt	Enable scalable directory service on compute nodes
PFS Server	useDistDEnt	Enable scalable directory service on I/O nodes

Table 4.1: Scalable Directory Options

4.2.2 Operations

Some of the PVFS operations are related to the scalable directory service. We make sure that each of those operations are properly modified so that they can correctly work with and without scalable directory service.

A list of scalable directory related operations on client-side(other operations like Read, Write and Close Operation are not related):

- Create Directory Operation
- Delete Operation
- Open Operation
- Read Directory Operation
- Stat Operation

A list of scalable directory related operations on server-side:

- Lookup Operation
- Change Directory Entry Operation
- Remove Directory Entry Operation
- Create Directory Entry Operation

- Read Directory Operation
- Split Directory Operation

All of these operations are implemented using different finite state machines, those which are related to scalable directories have two versions: one to work with scalable directory service and one to work without the service. We will discuss each of them both on the client-side and server-side.

4.2.3 Client

Table 4.2 contains the information of the finite state machines used by different operations.

	Lookup	GetAttribute	CreateDir	Remove	Create	ReadDir
Create Dir Op	x	x	x			
Delete Op	x			x		
Open Op	x	x			x	
Read Dir Op		x				x
Stat Op	x	x				
Service Related	x		x	x	x	

Table 4.2: Client Operation State Machines

Note that the Lookup, CreateDir, Remove and Create state machines are related to the scalable directory service.

- **Lookup**: as is shown in Figure 4.4, the state machine contains three states, in which the "LOOKUP_HANDLE" state implements the recursive searching for path segments that we mentioned in Section 3.3. The change we make to the version that works with scalable directory is that instead of sending a request to the home server of the directory being accessed, a P2S map is used to find the location where the directory entry resides.

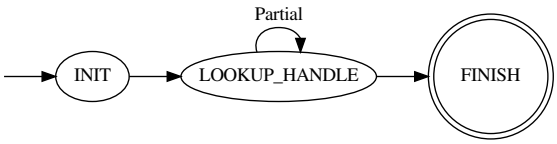

Figure 4.4: Lookup Name State Machine

- **CreateDir**: as is shown in Figure 4.5, the state machine contains six states, in which the "WR_DENT" state implements the operation of writing a directory entry onto a metadata server. The change we make is using the P2S map to locate the target metadata server before sending the request.

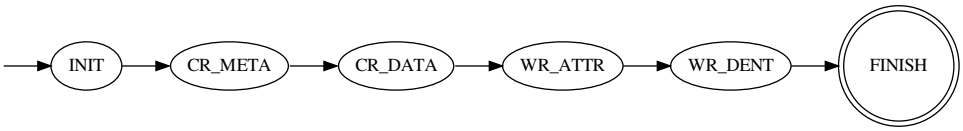

Figure 4.5: Create Directory State Machine

- **Remove**: as is shown in Figure 4.6, the state machine contains six states, in which the "RM_DENT" state implements the operation of removing directory entry from metadata server. Similarly, we make changes so that the P2S map is used to locate the target metadata server.

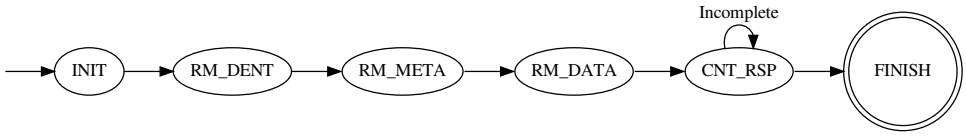

Figure 4.6: Remove Directory Entry State Machine

- **Create**: as is shown in Figure 4.7, the state machine contains seven states, in which the "WR_DENT" state implements the operation of writing directory entry onto metadata server. We make changes so that not only the P2S map is used to locate

the target metadata server, but also the P2S map is updated when the metadata server responds, in case that the directory is split on the server.

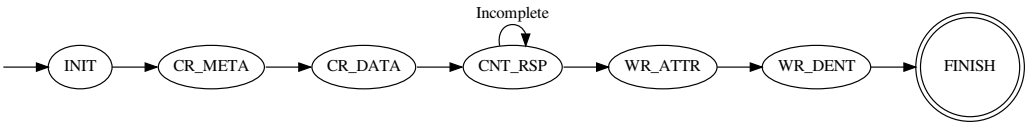

Figure 4.7: Create Directory Entry State Machine

To sum up, the implementation of scalable directory service in HECIOS on client-side is accomplished mainly by integrating the use of the P2S map into various state machines.

4.2.4 Server

Unlike the client-side, each of the operations on the server-side is implemented by a single state machine. We discuss those scalable directory service related state machines by comparing the version that works with the scalable directory service to the version that does not.

- **Lookup**: Figure 4.8 shows the lookup state machines that work with(bottom) and without(top) scalable directory service. To implement the scalable directory, a "LOCATE_DENT" state is added to look up the P2S map and determine if the directory entry of interest is located locally. The request is forwarded to another metadata server in case that the directory entry is not here. The "FORWARD" state implements the operation of forwarding request. Note that the target metadata server of the forwarded request will inherit the state of the state machine from the sender. The dashed arrow in the figure shows the behavior of the target metadata server. As for the sender, it will wait until the response is received from the target server. Its P2S map is subject to update if necessary during the process.

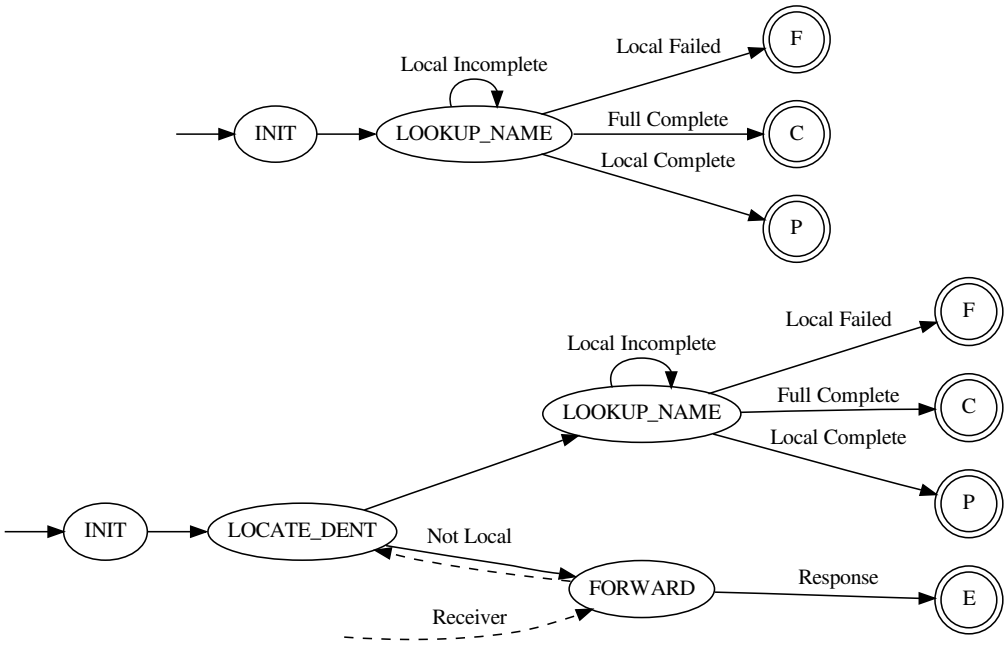

Figure 4.8: Server-side Lookup State Machine

- **Change and Remove Directory Entry**: Figure 4.9 shows the access directory entry state machines that work with(bottom) and without(top) scalable directory service, including the changing and removing of directory entries. Those two operations can be modeled the same way because the differences among them only lie in the implementations of the "WR_DENT" state.

  Similar to the lookup state machine, a "LOCATE_DENT" state is added to work with the P2S map. For requests of directory entries that are not locally located, the metadata server forwards them to other metadata servers and waits for responses. The state of the state machine from the sender server is inherited by the receiver as is shown by the dashed arrow in the figure. The servers involved in the process update their P2S maps accordingly.

- **Create Directory**: Figure 4.10 shows the create directory entry state machines that

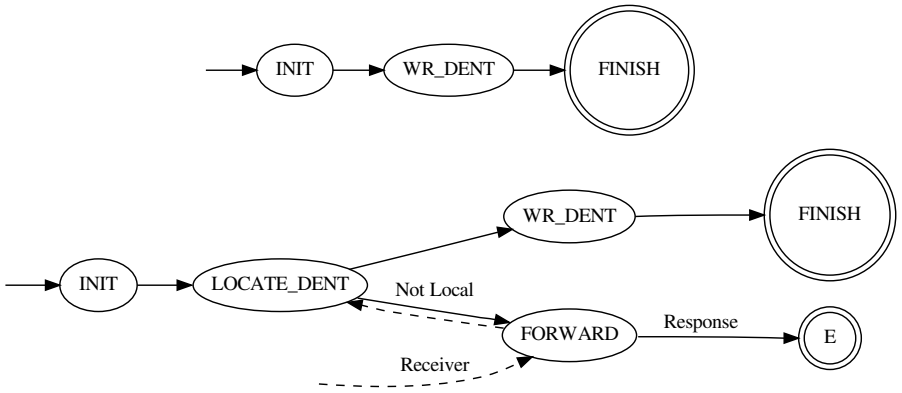

Figure 4.9: Server-side Access Directory Entry State Machine

work with(bottom) and without(top) the scalable directory service. We have made changes so that the state machine that works with the scalable directory service supports both the use of the P2S map and directory splitting. Similar to the above state machines, the "LOCATE_DENT" and "FORWARD" states are implemented for the P2S map lookup and request forwarding. Moreover, a "CNT_DENT" state is implemented to decide whether a particular directory has exceeded its capacity on the server and whether there are servers available for directory splitting. If both conditions are satisfied, a "SPLIT_DENT" state is used to perform the splitting operation. A separate split directory state machine is developed for the target metadata server to perform the operation on its own side. Again, the P2S maps are subject to update for any involved servers.

- **Read Directory**: Figure 4.11 shows the read directory state machines that work with(bottom) and without(top) scalable directory service. We make changes so that distributed directory entries are properly collected and sent to the client. A "CHK_SPLIT" state is implemented to check if request forwarding is necessary–when a metadata server is known to hold directory entries and is not contacted by the request sender.

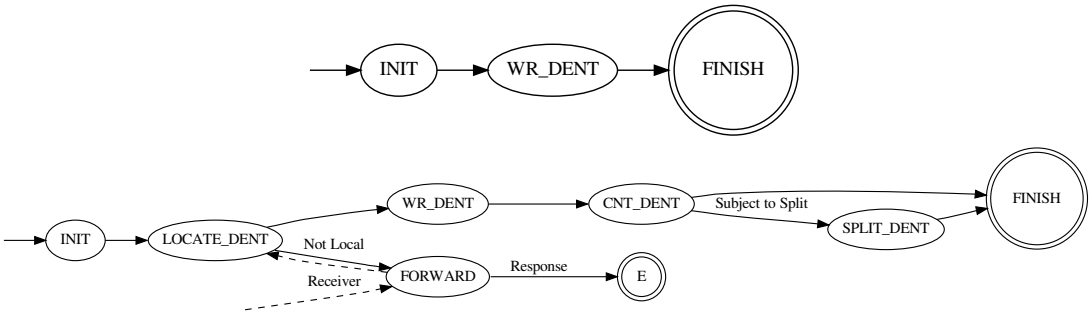

Figure 4.10: Server-side Create Directory Entry State Machine

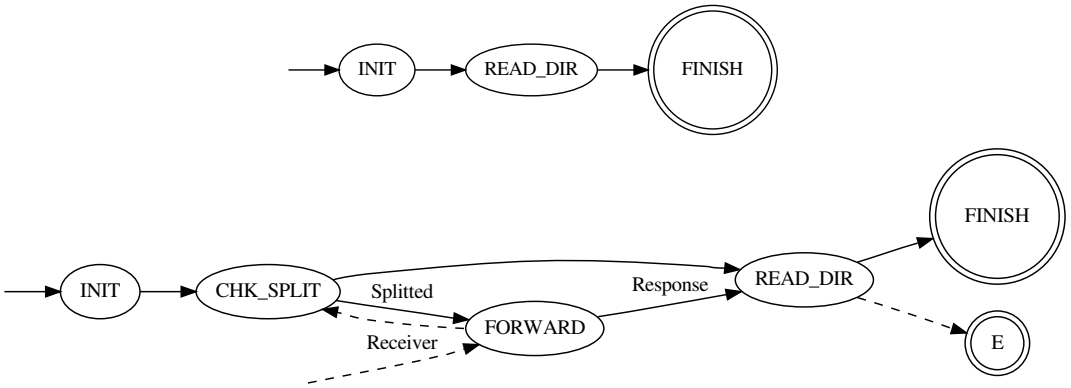

Figure 4.11: Server-side Read Directory State Machine

This ensures any metadata servers that hold a portion of the directory are contacted and none of them will be contacted more than once. The dashed arrow from "READ_DIR" state to "E" state indicates responding to a sender server other than to the client.

- **Split Directory**: Figure 4.12 shows the split directory state machine that works with the scalable directory service. This operation is triggered when a metadata server receives a split request from another server. Upon receiving the request, the target metadata server stores the directory entries locally and updates its P2S map accordingly.

Figure 4.12: Server-side Split Directory State Machine

In conclusion, the implementation of scalable directory service on server-side mainly focuses on request forwarding and directory splitting.

4.3 Modeling

One of the goals of designing a simulator is to make sure that the experiment results are good approximation of the real world performance data. Thus it is important to carefully design the simulation model and properly validate it. In our case, the simulation model of HECIOS is designed to accurately simulate the performance of the Palmetto Cluster in Clemson University. Moreover, we develop mathematical models to model the performance of I/O operations on both the client-side and server-side.

4.3.1 Architecture

The Palmetto Cluster is the fastest supercomputer in Clemson University. It is a collection of 772 high-end computational nodes as is shown in Table 4.3. Each compute node has a dual processor and each processor has four cores(quad-core) for a total of 6176 cores for computation. It is demonstrated that a cluster performance of 46 TFlops has been reached on Palmetto by the Clemson Computing & Information Technology(CCIT) staff in [6].

Two independent interconnection networks are provided by Palmetto for each compute node: a Gigabit Ethernet network interface and a Myrinet Myri-10G network interface.

Model	Processor	Count	L2 Cache (MB)	Cores	Memory (GB)
Dell PE 1950	Intel Xeon E5345 @2.33GHz x2	258	4	8	12
Dell PE 1950	Intel Xeon E5345 @2.33GHz x2	258	6	8	12
Sun X2200 M2 x64	AMD Opteron 2356 @2.3GHz x2	256	4	8	16

Table 4.3: Palmetto Compute Nodes Architecture

While the former is used by both the application and cluster management network, the latter is dedicated for application use, which provides low-latency message passing and 1.2 GB/s of sustained network bandwidth.

On Palmetto, local storage is provided with each compute node and longer term storage is provided via a 120 TByte StroageTek Model SL8500 storage system, on which SAM-QFS Storage Management Software is used for long term storage(e.g., /home, /projsmall, /projlarge) and PVFS is used for fast storage(e.g., /pvfs2).

In HECIOS, the networking environment is modeled by using the INET network simulation package, which constructs a TCP/IP network transport over a switched Ethernet local area network. In [26], Settlemyer described a detailed process of validating and benchmarking the simulation model of HECIOS according to the performance of Palmetto, including but not limited to the performance of PVFS, Gigabit Ethernet and Myri-10G by using MPI and Flash I/O benchmarks. The result shows that HECIOS was able to provide an adequate simulation of how the proposed modifications would effect a diverse metadata intensive workload.

4.3.2 Operations

Before running any simulations on HECIOS, we firstly run MPI programs in a cluster against the tracing tools provided by LANL to retrieve trace files that contain informa-

tion regarding the runtime, system and MPI calls, parameters of those MPI programs. The trace files can then be translated by tools that are part of HECIOS into the formats readable by HECIOS. Using these readable formats(HECIOS trace formats) as inputs, HECIOS can simulate the execution of the MPI codes in a configured simulation environment. Since HECIOS trace formats contains the timing and parameters of each MPI and system calls, HECIOS can provide accurate simulation of MPI codes.

Figure 4.13 is a sequence diagram that shows how the simulation on HECIOS is driven by trace files. A sequence of instructions is retrieved from the trace files and fed to client. The client then executes each of the instructions either locally or by contacting servers until all instructions from the traces files are served. To accurately time every MPI call, a special "CPU_PHASE" instruction is used to simulate the executing of CPU instructions that are not related to I/O, which causes the client to pause for a certain amount of time that is specified in the trace files.

ID	Instruction	Start Time(s)	Duration(s)	Return	Parameters
0	MPI_COMM_RANK	0.013379	0.000225	0	0x44000000...
1	CPU_PHASE	0.013604	0.005907	0	
2	MPI_FILE_OPEN	0.019511	0.003594	0	0x44000001...
3	CPU_PHASE	0.023105	5.2e-05	0	
...	...	...	...	...	...

Table 4.4: Example: HECIOS Trace Format

Table 4.4 shows the content of a sample HECIOS trace file, where there is a CPU PHASE instruction in every two instructions. By pausing the client for the amount of time that is equals to the duration for each CPU PHASE instruction, a pretty close approximation of the real world CPU computational power can be reached in the simulation.

The execution of each MPI operation can be divided into local computation and network communication on client-side and server-side. Network latency applies to the "Request/Response" communication shown in Figure 4.13, which is handled by the INET

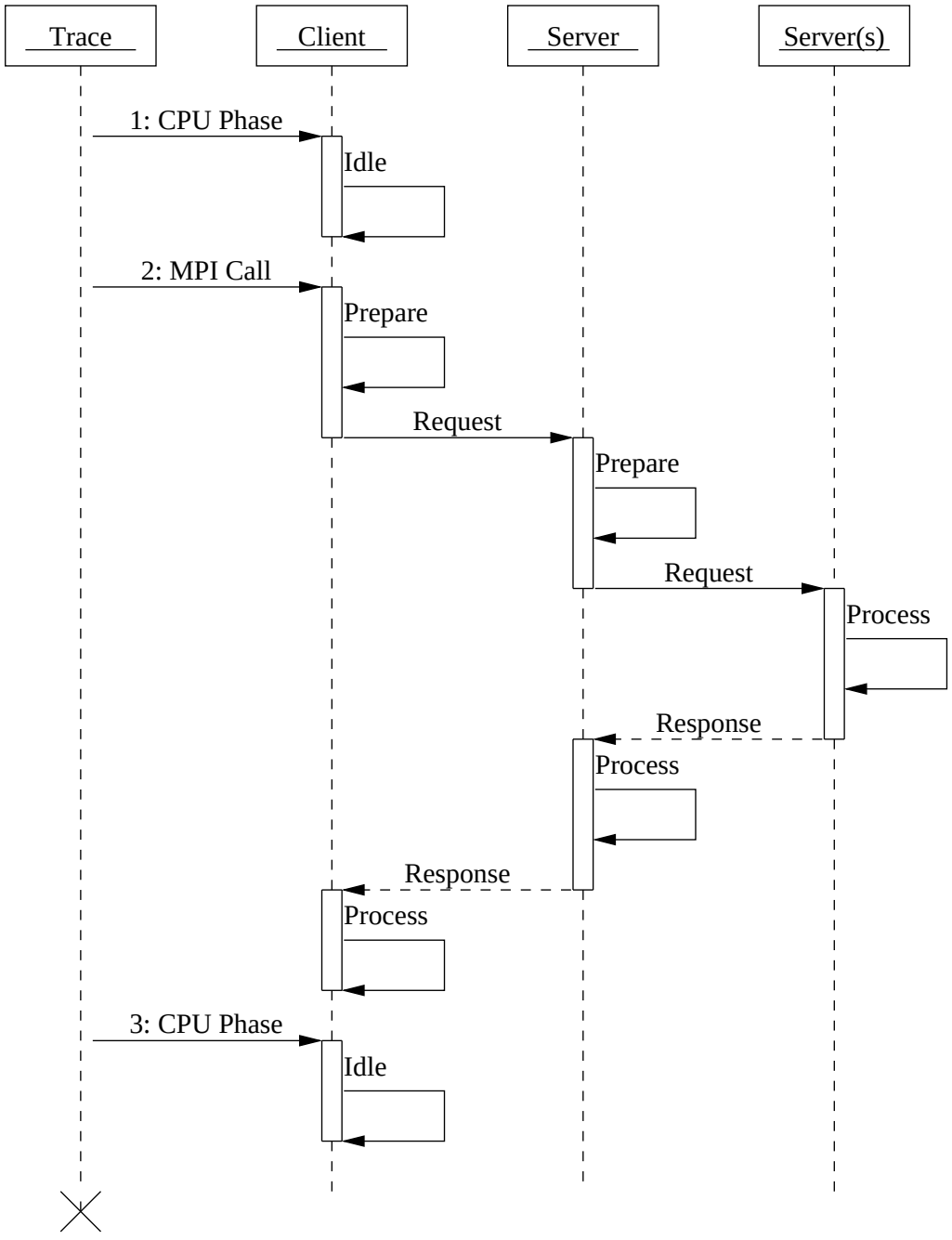

Figure 4.13: HECIOS Simulation Sequence Diagram

network simulation package. Since the computation time on client-side is simulated by CPU Phase, here we specifically model the local computation time of the following operation on server-side that are related to our scalable directory service by benchmarking the server module of the PVFS file system that runs on Palmetto.

The server module of PVFS is implemented using a set of finite state machines, each of which implements an operation. By adding timing code into the state machines that we are interested in, we are able to benchmark the performance of the server module of PVFS on Palmetto. In our configuration, we use 1 compute node and 2 I/O nodes, each of which takes 8 processes per node to avoid interference from other jobs.

The MPI program we used to benchmark the performance of directory entry operations in PVFS on Palmetto firstly creates 10000 files within a single directory and deletes them one by one in the end. The process involves operations like creating, removing and looking up directory entries.

4.3.3 Create Directory Entry Processing

As is shown in Figure 4.14, the create directory entry processing time on server-side in PVFS on Palmetto roughly forms a normal distribution, in which the probability density function is given by formula 4.1. The distribution is decided by both μ which is the mean value of x(processing time in our case) and σ which describes the range and shape of the probability density curve.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{4.1}$$

So we model the server-side create directory entry processing time as a normal distribution and we use the cumulative distribution function(cdf), which is given by formula 4.2, to determine the proper μ and σ to use.

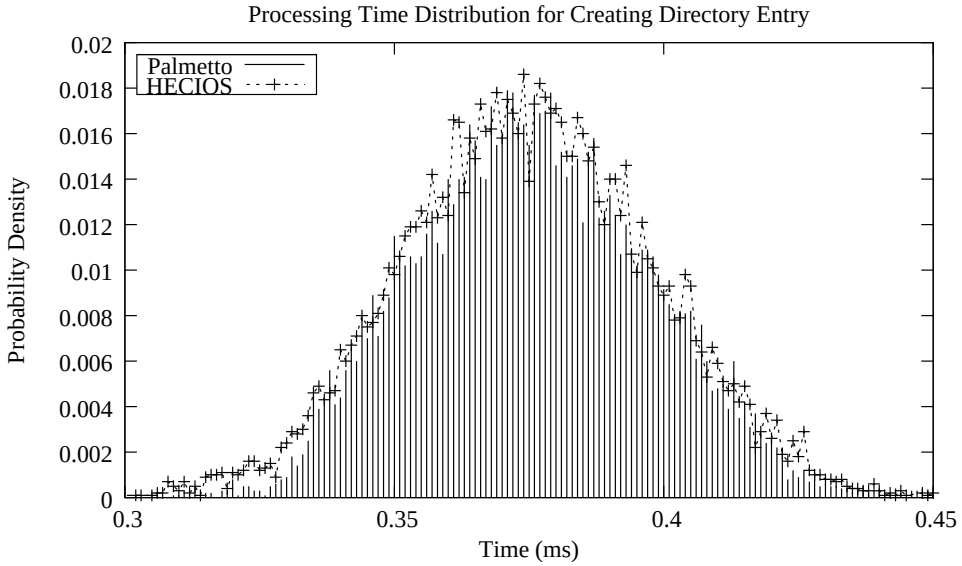

Figure 4.14: Create Directory Entry Processing Time Distribution

$$\Phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}(\frac{x-\mu}{\sigma\sqrt{2}})\right) \tag{4.2}$$

where $\mathrm{erf}()$ is called the Gauss error function that is given by formula 4.3.

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t \tag{4.3}$$

Figure 4.15 shows the curves of the cdf with different pairs of μ and σ, from which it is determined that the pair $\mu = 0.378$ and $\sigma = 0.025$ matches best the performance data. Thus the server-side create directory entry processing time is modeled as a normal distribution with the probability equation 4.4, where t is the processing time.

$$p(t) = \frac{1}{0.025 \times \sqrt{2\pi}} \exp\left(-\frac{(t-0.378)^2}{2 \times 0.025^2}\right) \tag{4.4}$$

The HECIOS curve in Figure 4.14 represents the performance data collected by running the benchmark in HECIOS with the above create directory entry processing time

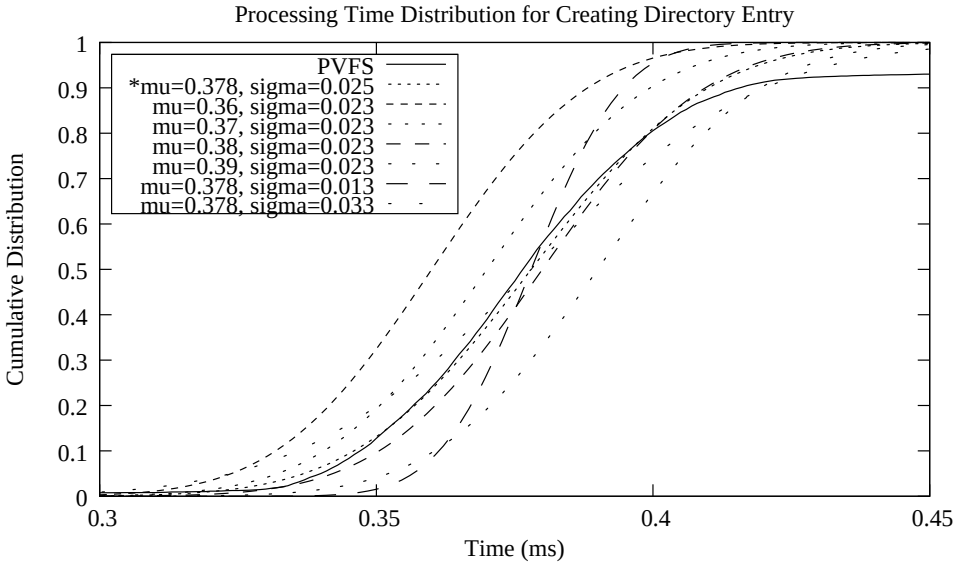

Figure 4.15: Create Directory Entry Processing Time Cdf

model. It can observed that the simulation data matches the experiment data pretty well.

4.3.4 Remove Directory Entry Processing

As is shown in Figure 4.16, the remove directory entry processing time also roughly forms a normal distribution. Similar to the create directory entry processing time, we model the remove directory entry processing time as a normal distribution as well. The curves of cdf with different pairs of μ and σ are shown in Figure 4.17, where the pair $\mu = 0.375$ and $\sigma = 0.023$ matches best the performance data. Thus we model the remove directory entry processing time as a normal distribution with the probability equation 4.5.

$$p(t) = \frac{1}{0.023 \times \sqrt{2\pi}} \exp\left(-\frac{(t-0.375)^2}{2 \times 0.023^2}\right) \tag{4.5}$$

The HECIOS curve in Figure 4.16 validates this model.

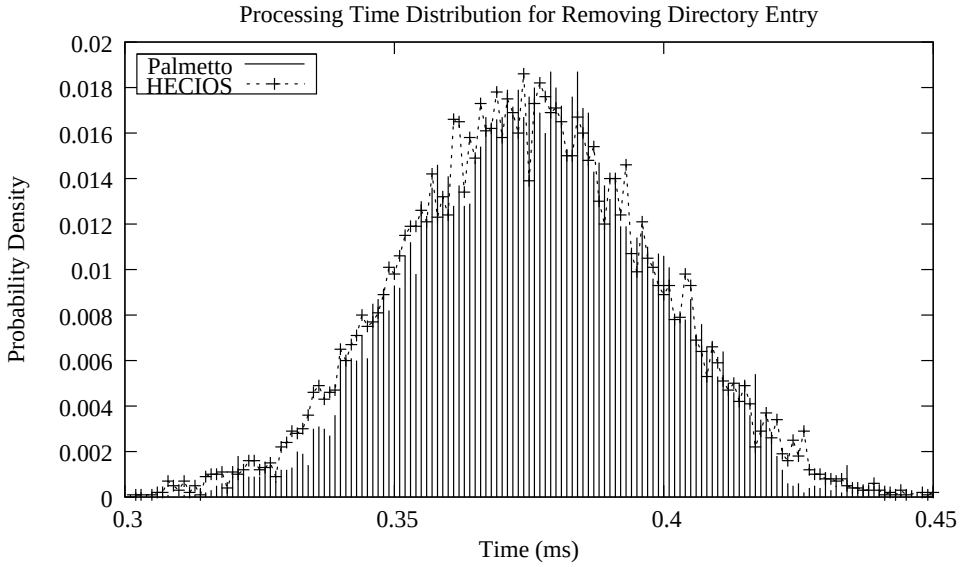

Figure 4.16: Remove Directory Entry Processing Time Distribution

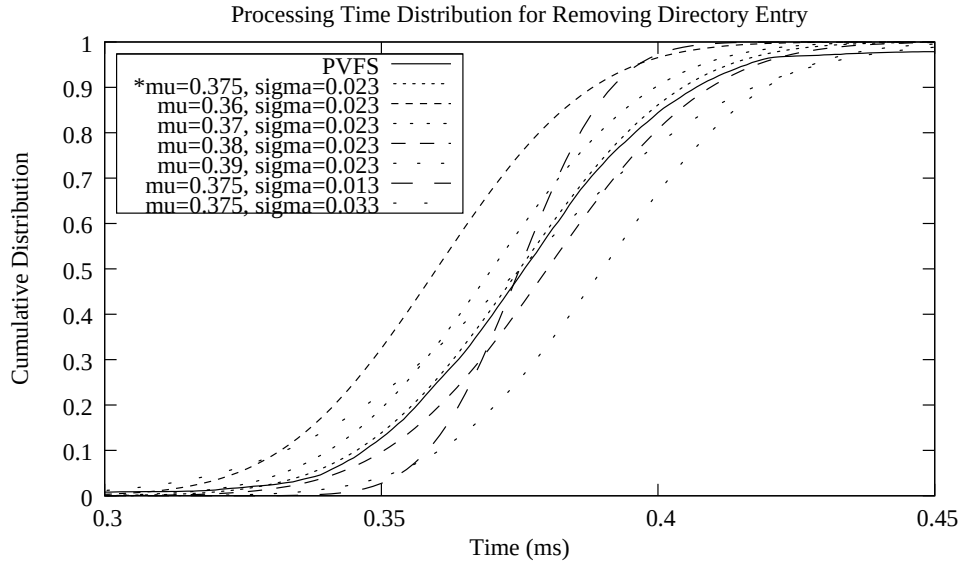

Figure 4.17: Remove Directory Entry Processing Time Cdf

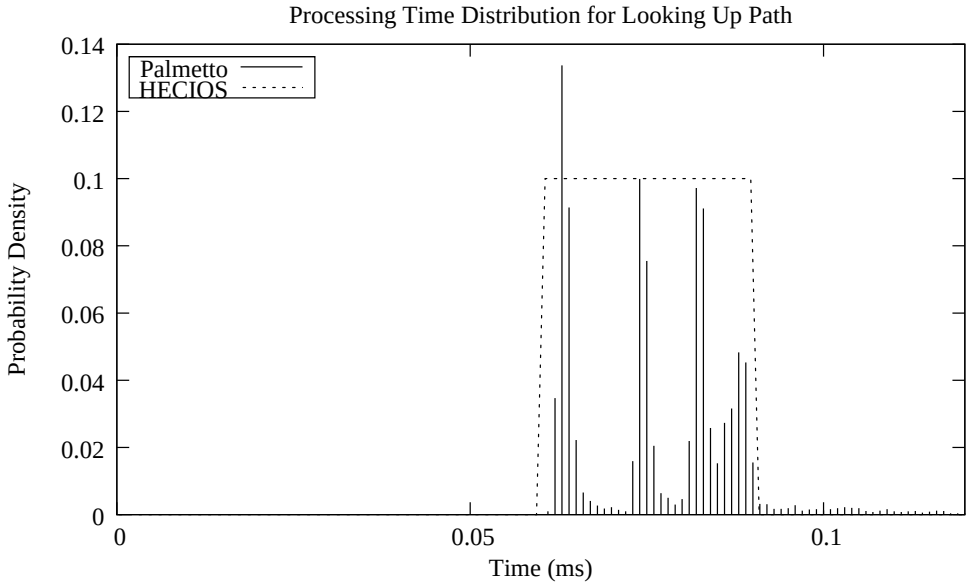

Figure 4.18: Lookup Path Processing Time Distribution

4.3.5 Lookup Path Processing

As is shown in Figure 4.18, the server-side lookup path processing time does not form a well-known distribution pattern, nor does it show obvious relation with the directory size as is shown in Figure 4.19. It is observed that over 90% of the values range from 0.06 to 0.09. Thus we model the lookup path processing time as a uniform distribution within the range $[0.06, 0.09]$ as is given by probability function 4.6. The cdf is shown in Figure 4.20.

$$p(t) = \begin{cases} \frac{1}{0.09-0.06} & 0.06 \leq t \leq 0.09 \\ 0 & otherwise \end{cases} \tag{4.6}$$

The HECIOS curve in Figure 4.18 represents probability density function of the above lookup path processing time model. Though it inaccurately approximates the experiment data, the impact of this inaccuracy in our model is limited in this research because: firstly, in our experiment clients cache directory info locally, so lookup path operations only

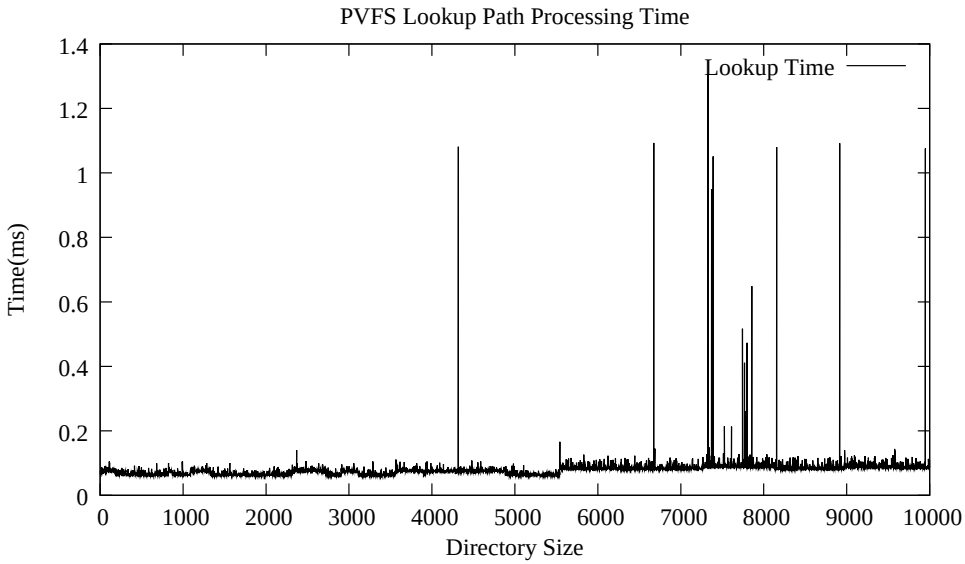

Figure 4.19: Lookup Path Processing Time

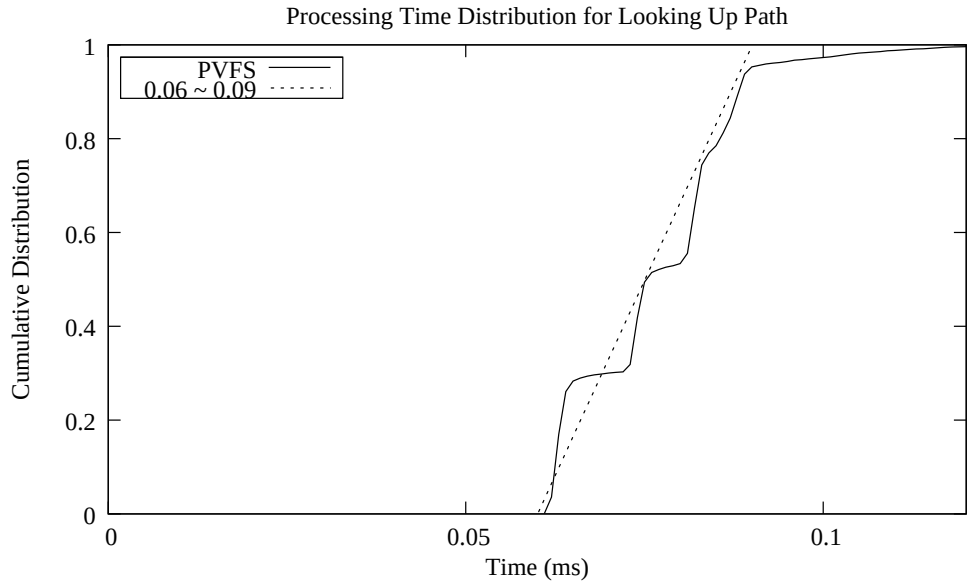

Figure 4.20: Lookup Path Processing Time Cdf

occur in the beginning; secondly, since our experiment operates within a single directory, the number of lookup path operations is small.

4.3.6 P2S Map Processing

Many of the server-side directory operations that work with the scalable directory service involve processing the P2S map. Since the P2S map is a bitmap in which each bit represents a metadata server, the size of P2S map is as small as

$$S = \frac{N}{8} \tag{4.7}$$

where S is the size in bytes and N is the number of metadata servers. That is to say, the size of the P2S map of a directory on a cluster with 1024 metadata server is 128 bytes(< 0.2KB). Thus the processing time for P2S map operation on modern computers is too small to put into consideration compared to I/O operations and network operations. Further, most operations on the P2S map are logarithmic. So we model the P2S map processing time as 0.

Chapter 5

Results

In this chapter, we will describe a series of experiments that we conducted in studying the throughput and scalability, directory growth rate and synchronization overhead of our scalable directory service design, in which we ran benchmark programs on Palmetto with PVFS and collected the MPI I/O traces. All trace files were then transformed into HECIOS trace format and executed in HECIOS with the simulation models that we developed in previous chapters. An analysis of the simulated performance data is also included.

5.1 Mathematical Model

First, we develop a mathematical model to help understanding the whole system and analyzing the performance data. We use the queuing model[14] that is widely used for network performance analysis to describe our system. Normally a queuing system consists of:

- One or more servers that provide service to arriving customers
- A waiting queue with space for zero or more customers to await access to a server

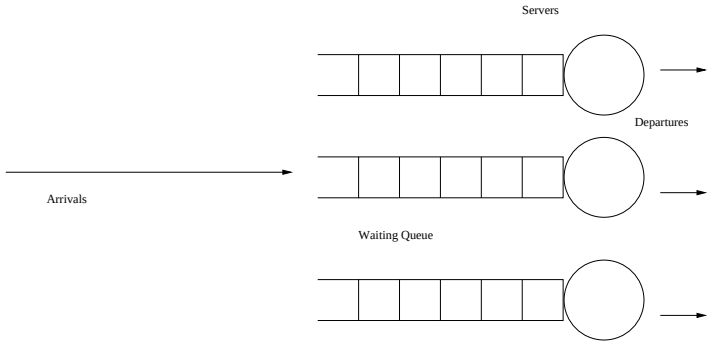

Figure 5.1: Directory Entry Creation Queuing Model

In our experiment, the servers refer to metadata servers that handle directory entry creation requests, which are referred as the customers in the model. Each of the servers maintains a waiting queue to ensure exclusive access to the directory data file by a certain request.

5.1.1 Queuing Model

Figure 5.1 shows the queuing model of our experiment configuration, which is considered as a single source multiple server queuing system. According to Kendall's notation for queuing systems [14], it can be categorized as a $G/M/n$ queuing system, where G stands for the generic arrival process, M the exponentially distributed service time and n the number of parallel servers. Such queuing system can be described by:

- λ: The arrival rate
- μ: The service rate
- n: The number of parallel servers

and the performance of which can be determined with:

- Mean delay: The mean time a customer spends in system

- Mean queue length: The mean number of customers in the waiting queue
- Server utilization: The fraction of time that a server is busy
- Throughput: The throughput of the queuing system

5.1.2 Little's Law

According to Little's Law[19], in a queuing system, let N represent for the mean number of customers in system, λ the arrival rate and T the mean time that a customer spends in system, then,

$$N = \lambda \times T \tag{5.1}$$

In addition, let N_q denote for the mean number of customers in waiting queue and W the mean time a customer spends in the waiting queue,

$$N_q = \lambda \times W \tag{5.2}$$

Thus, we have

$$T = W + \frac{1}{\mu} \longrightarrow N = N_q + \frac{\lambda}{\mu} = N_q + \rho \tag{5.3}$$

where ρ is the server utilization. For our system, since the clients wait for the completion of the previous request to initiate a new one, the request arrival rate equals the departure rate. Consequently our system is a stable system in which the total number of requests in the queues on all servers(N_q) is a constant that equals to the number of compute nodes. Thus if the servers are fully utilized so that the departure rate equals to the service rate, then $\lambda = \mu$ and thus $\rho = 1$ in our system.

5.2 Throughput and Scalability

To benchmark the throughput and scalability of our scalable directory service design, we use a MPI program that continuously creates files within a single directory. We are able to collect throughput data by varying the number of I/O nodes from 1 to 32 with a fixed number of compute nodes. Similarly, we collect scalability data by running a fixed number of I/O nodes with various number of compute nodes($1 \sim 512$). In the experiment, we configure the directory to be split during creation time so that I/O traffic can be balanced and the throughput and scalability can be easily observed. As a comparison, the performance data of a dynamically splitting directory is also collected.

OMNeT++ provides two data recording tools: a vector class that records vector data and a scalar class that records scalar data. We use these tools to collect performance data including server directory entry create processing time and client directory entry create wait time.

5.2.1 Throughput

Figure 5.2 shows the directory entry creation throughput in a pre-distributed directory of running various number of compute nodes that continuously create entries while scaling the number of I/O nodes from 1 to 32. It is observed that with a larger number of compute nodes, which introduce higher creation request rate, the throughput scales better in a sense that it can reach a higher value and keep to scale until a larger number of I/O nodes is used. On the other hand with a smaller number of compute nodes(e.g., 32) the throughput stops scaling and even goes down beyond 6 I/O nodes. This is because 32 compute nodes can not provide the creation request arrival rate that is high enough to keep the I/O nodes busy and highly utilized.

It is also noticeable that when the number of I/O nodes is relatively small($<= 6$),

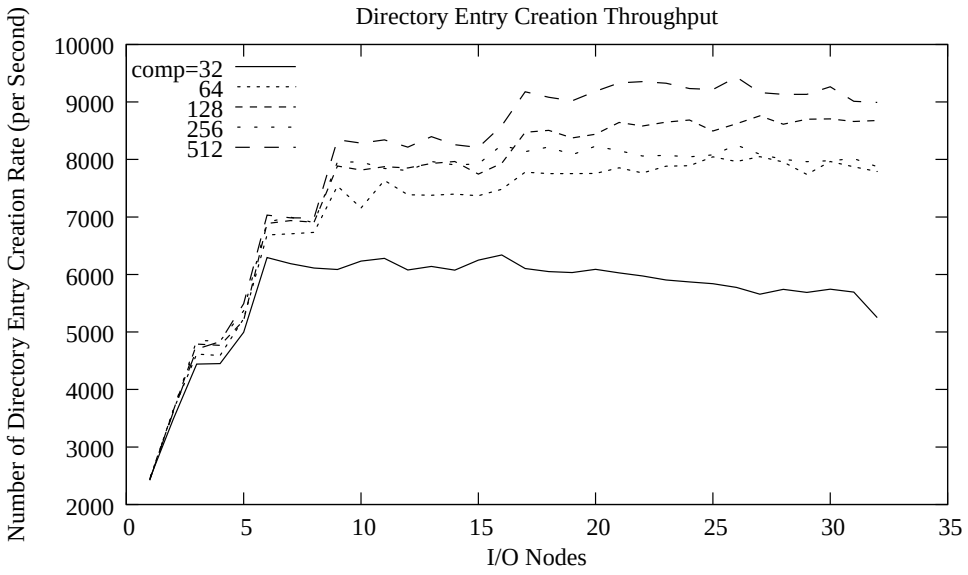

Figure 5.2: Directory Entry Creation Throughput for Pre-Distributed Directory

increasing the number of compute nodes will not dramatically improves the throughput. This is because the potential arrival rate of requests from 32 compute nodes has already exceeded the overall service rate, and the servers are 100% utilized. Similarly, it can be observed that 9 servers are fully utilized roughly by 64 compute nodes.

Figure 5.3 shows the queue size during the directory entry creation process. It is observed that a smaller number of compute nodes results a shorter waiting queue. By referencing it to Figure 5.2, we can observe noticeable throughput increase when the queue length dramatically decreases. This is because a smaller waiting time in queue(W in Equation 5.2) which leads to a shorter queue also indicates a higher departure rate.

We also experiment with a directory that is not pre-distributed and the throughput is shown in Figure 5.4. It is observed that not only is the throughput lower than that of a pre-distributed directory, but also the improvement gained by increasing the number of compute nodes is less noticeable. Since the directory is dynamically splitting, the number

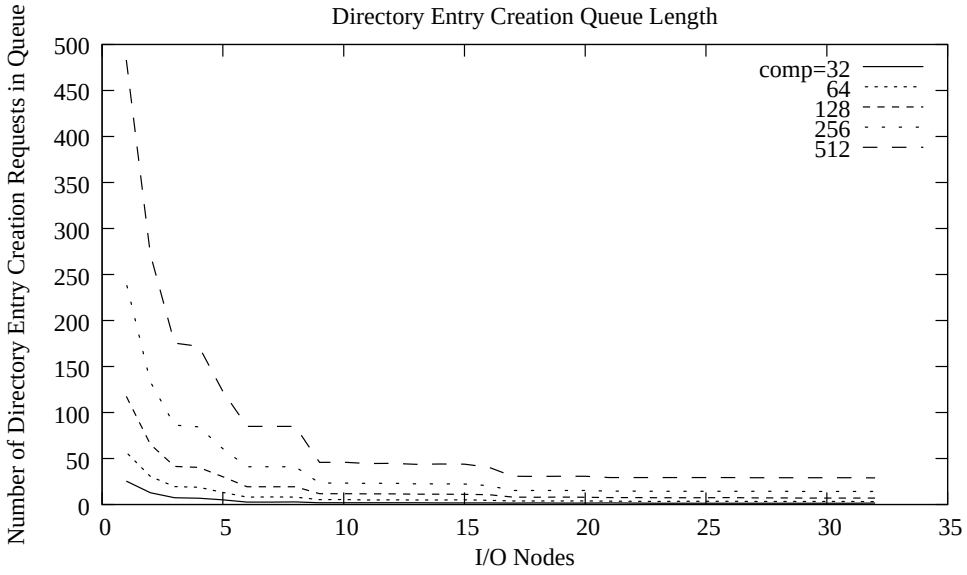

Figure 5.3: Directory Entry Creation Queue Size

of metadata servers that handle the requests for the directory is small in the beginning. As the directory gradually splits to more metadata servers, the departure rate begins to grow. Thus the overall departure rate is lower than that of a pre-distributed directory. Besides, the splitting operation and P2S map synchronization that occur during the process introduce communication overhead that decreases the performance.

In summary, the throughput of directory entry creation in our scalable directory service is closely related to the request arrival rate(number of compute nodes), the service rate(number of I/O nodes) and the server utilization. It is also desirable to pre-distribute large directories on creation to gain performance improvement.

5.2.2 Scalability

Figure 5.5 shows the mean waiting queue size during the process of directory entry creation in a pre-distributed directory on various number of I/O nodes when scaling the

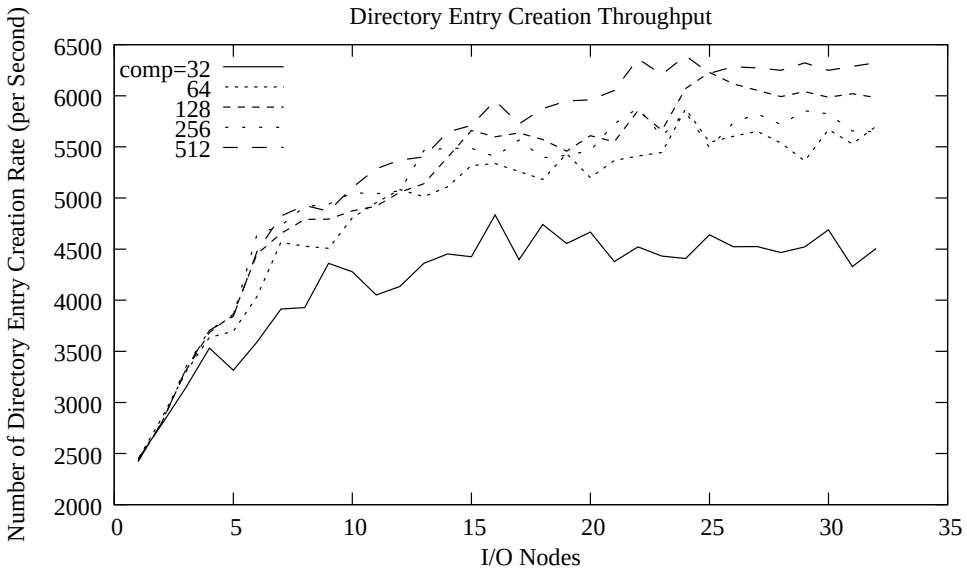

Figure 5.4: Directory Entry Creation Throughput in a Dynamically Splitting Directory

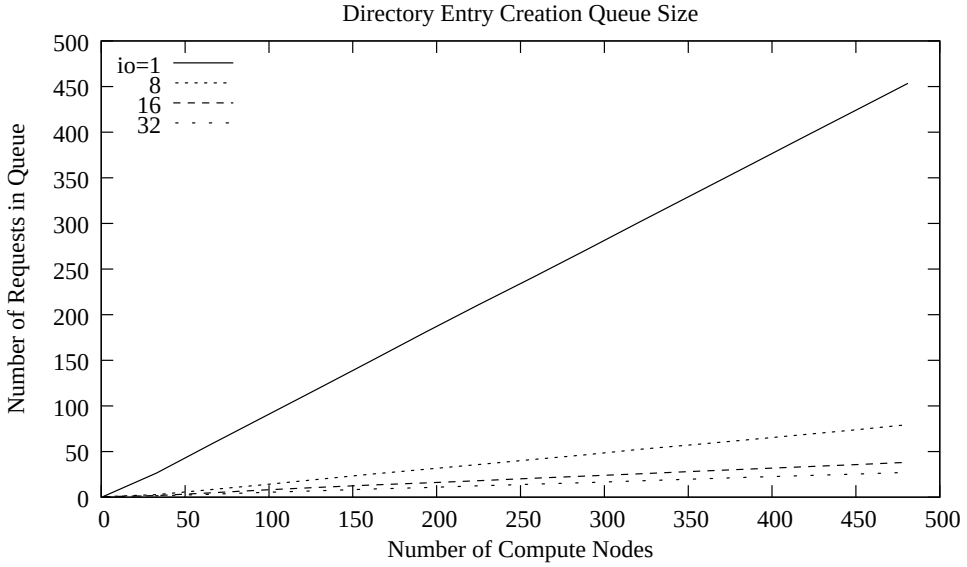

Figure 5.5: Directory Entry Creation Queue Size for a Pre-Distributed Directory

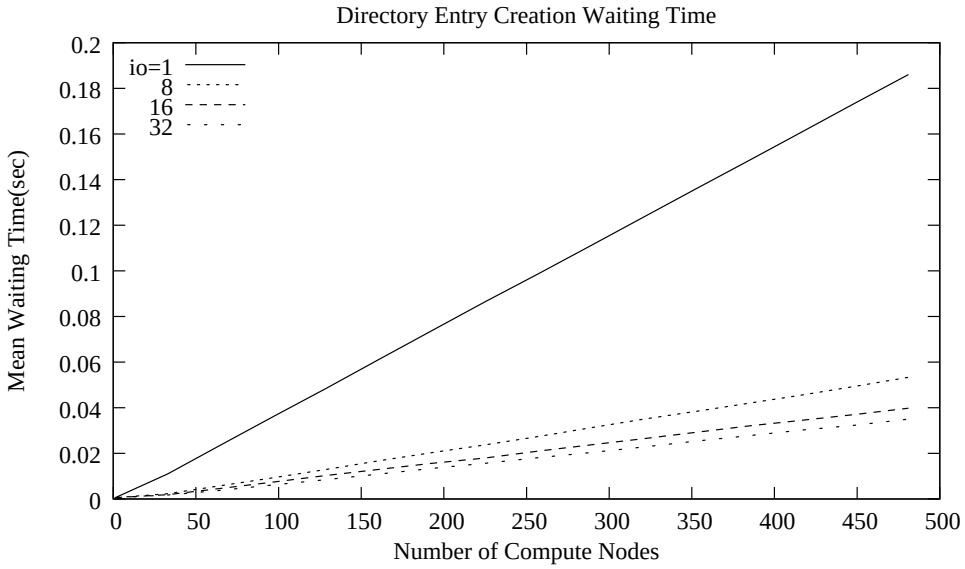

Figure 5.6: Directory Entry Creation Waiting Time for a Pre-Distributed Directory

number of compute nodes. Let n_{cp} be the number of compute nodes and n_{io} be the number of I/O nodes. It is noticeable that the average queue size on each server equals to the number of compute nodes divided by the number of I/O nodes.

$$N_q = \frac{n_{cp}}{n_{io}} \tag{5.4}$$

Figure 5.6 shows the directory entry creation waiting time during the process in a pre-distributed directory. It is observed that the mean waiting time increases linearly with the number of compute nodes and the higher the number of I/O nodes is, the lower the slope is.

According to Equation 5.2 and Equation 5.3, we have,

$$\overline{T} = \frac{1}{\mu} + \overline{W} = \frac{1}{\mu} + \frac{\overline{N_q}}{\overline{\lambda}} \tag{5.5}$$

Plug Equation 5.4 into Equation 5.5, we have,

$$\overline{T} = \frac{1}{\mu} + \frac{n_{cp}}{n_{io} \times \overline{\lambda}} \tag{5.6}$$

in which for each server, μ is a constant and $\lambda = \rho \times \mu$, thus we have,

$$\overline{T} \sim n_{cp} \tag{5.7}$$

and

$$\overline{T} \sim \frac{1}{n_{io}} \tag{5.8}$$

As a comparison, Figure 5.7 shows the mean waiting time of creating directory entries in a dynamically splitting directory. It is noticeable that the waiting time shows a similar relation with n_{cp} and n_{io} in this case. It is also observed that the waiting time is the same with that in a pre-distributed directory for 1 I/O node and is larger for other numbers of I/O nodes. This is because the directory splitting and P2S map synchronization overhead decrease the performance.

In summary, for our scalable directory service, the directory entry creation waiting time scales well with the number of I/O nodes so that more I/O nodes bring better performance for a large number of compute nodes. Again, it is shown that to pre-distribute large directories and thus to avoid splitting and synchronization overhead can lead to performance improvement.

5.3 Directory Growth

Figure 5.8 shows how the directory size grows on each of the 32 servers for a pre-distributed directory when 512 clients continuously creating directory entries in it. Note

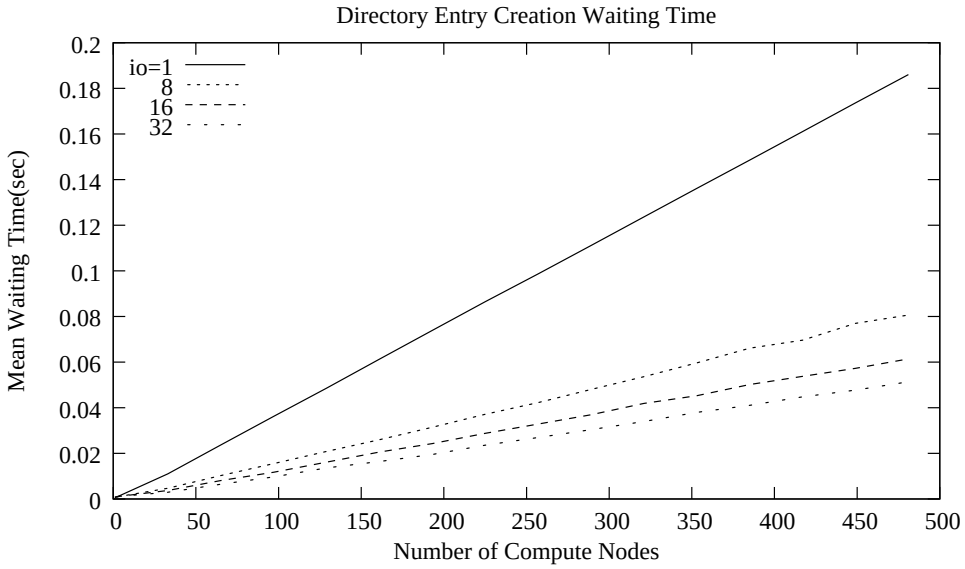

Figure 5.7: Directory Entry Creation Waiting Time for a Dynamically Splitting Directory

that during this process, each of the metadata servers is put into use since the very beginning and no directory splitting operation is involved. Consequently it is observed that the directory size on each server keeps increasing as more entries are added. For our scalable directory service, the location of a certain directory entry is decided by the hash value of its file name. In this experiment we randomly generate strings as the file names. The figure also shows that the directory entries are not equally distributed among the metadata servers. This is because the hash function is not ideal.

The same process for a dynamically splitting directory is shown in Figure 5.9. It is noticeable that the directory size on some servers decreases during the process, which indicates directory splitting operation. For dynamically splitting directories, the number of splitting operations is decided by both the directory entry creation pattern of the clients and the splitting threshold settings on the metadata servers. In our experiment, the splitting threshold equals to the total number of directory entries to be created divided by the number of I/O nodes.

For large directories, pre-distribution is favored because it distributes the directory

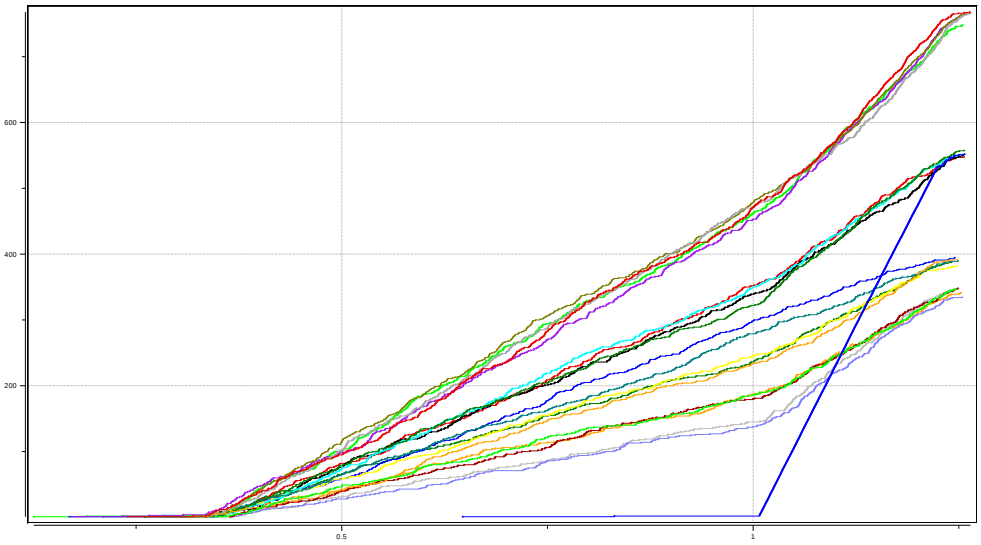

Figure 5.8: Directory Growth in Pre-Distributed Directory

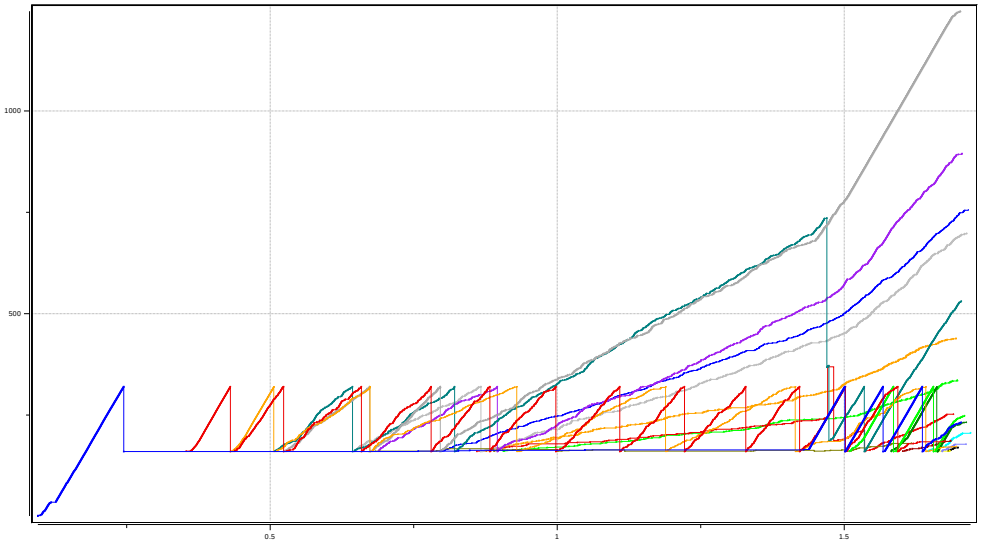

Figure 5.9: Directory Growth in Dynamically Splitting Directory

entries among the metadata servers without introducing the splitting overhead. However, if the directory size is unknown or known to be small, dynamically splitting is preferred because:

1. The number of metadata servers involved in operations like readdir can be limited.

2. It is adaptive to the growth of directory size and can benefit from directory distribution when needed.

5.4 Synchronization

In our scalable directory service, we introduce request forwarding on server-side to avoid unnecessary client/server communication overhead for synchronizing the P2S map. For a pre-distributed directory, the P2S maps on server-side are not changed during the process of creating directory entries and the P2S map on client-side only needs to update once for each client. Thus request forwarding operations happen to each client at most once. For a dynamically splitting directory, the P2S maps keep changing as the directory get split to more servers. As a result, more request forwarding operations are expected in this case.

In this experiment, we record the number of request forwarding operations during the process of continuously creating directory entries by a large number of clients in a pre-distributed directory and a dynamically splitting directory.

Figure 5.10 shows the number of request forwarding operations involved during the directory entry creation process in a pre-distributed directory when scaling the number of I/O nodes from 1 to 32. It is observed that the number of forwarding operations never exceeds the number of compute nodes. This is because the servers send responses to the clients with an updated P2S map that properly indicates the directory distribution. Thus

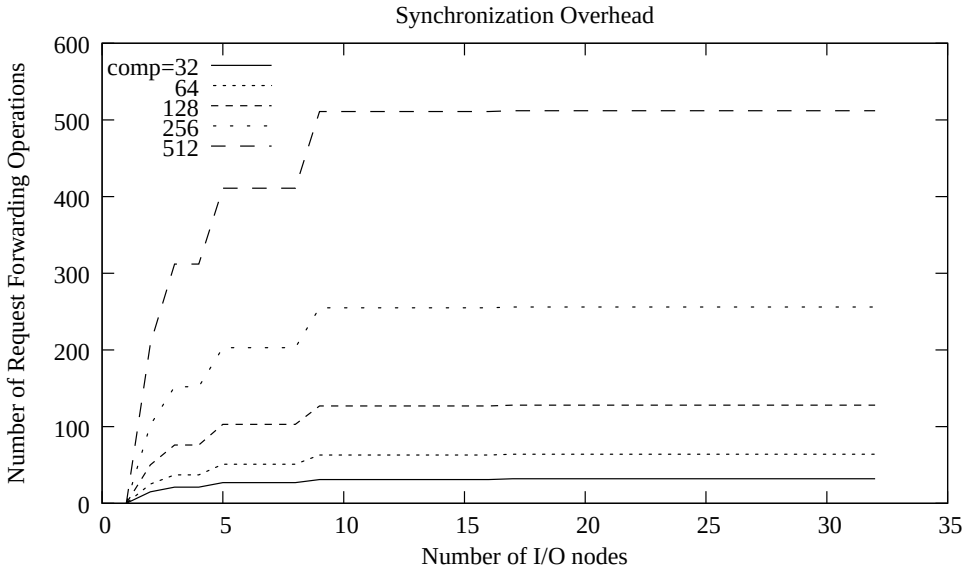

Figure 5.10: Synchronization Overhead in Pre-Distributed Directory

the clients will not send requests to the wrong server for a second time. In this case, the synchronization overhead is as small as a one-time cost for each client.

Figure 5.11 shows the number of request forwarding operations during the same process in a dynamically splitting directory. The number is noticeably higher than that in a pre-distributed directory, which indicates more significant synchronization overhead in this case. As a larger number of splitting operations are involved in the process, each of which changes the P2S map, the outdated P2S map on clients are more likely to cause metadata server miss hits and in turn more request forwarding operations. However, since the splitting operations are closely related to the file creation pattern and the splitting threshold settings, such synchronization overhead will not dramatically increase when adding more I/O nodes into the system. Thus it does not prevent our scalable directory service from scaling with the number of I/O nodes.

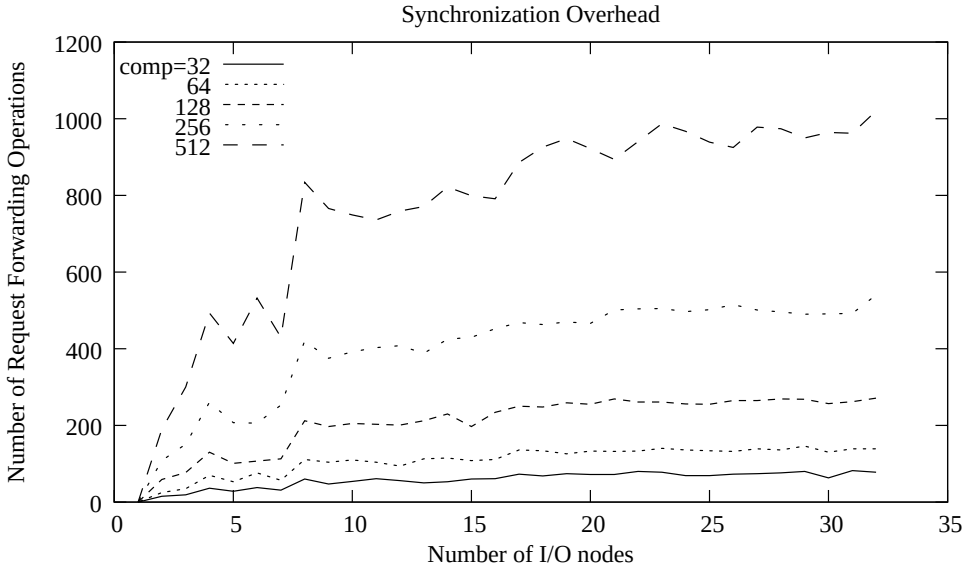

Figure 5.11: Synchronization Overhead in Dynamically Splitting Directory

Chapter 6

Conclusions

In this thesis we mainly focused on addressing the insufficiency of handling highly concurrent access to large directories in existing parallel file systems. In Chapter 1 we introduce several file systems with their directory services and propose a scalable directory service design for parallel file systems. Chapter 2 presents an overview of the related research projects including various file systems, HECIOS and GIGA+. In Chapter 3, we discuss the research that we conducted in scalable directory service and the issues of designing it within PVFS. Chapter 4 describes in detail the implementation of our scalable directory service for PVFS in HECIOS and the methods we used to verify and validate the simulation model. In Chapter 5, we demonstrate the performance data that we collected from experimenting with the scalable directory service in HECIOS, in which a mathematical model is also developed to analyze those data. In this Chapter, we summarize our experimental results and describe some future works.

6.1 Throughput

In our experiment, we found that the throughput of creating directory entries with our scalable directory service is decided by:

- The request arrival rate(related to the number of compute nodes)
- The service rate(related to the number of I/O nodes)
- The server utilization

Properly configured number of compute and I/O nodes can increase the overall request arrival rate and service rate, while maintaining the server utilization and thus yielding higher throughput.

6.2 Scalability

Our scalable directory service makes use of available I/O nodes in a cluster and provides performance improvement by decreasing the overall client waiting time for certain directory operations. We present experimental results and mathematical analysis to study and demonstrate the high scalability of our scalable directory service. In addition, we compare the performance of directory operations in a pre-distributed directory and a dynamically splitting directory to show further performance improvement with the former method by avoiding splitting and synchronization overhead.

6.3 Directory Growth

With the directory growth data that we collected from the experiments, we are able to demonstrate how pre-distributed directories gain better performance by avoiding

the directory splitting operations. We also show that dynamically splitting directories can adaptively adjust the number of metadata servers that they distribute to, so as to deal with different file access patterns.

6.4 Synchronization

We are able to show that the synchronization overhead is minimized by design and thus does not affect the scalability of our scalable directory service. We also demonstrate further synchronization overhead reduction by using pre-distribution method with large directories.

6.5 Summary

In summary, the scalable directory service fulfills the following design goals:

- **Maintain PVFS I/O semantics**: In our experiments, we collect trace files by running MPI programs with PVFS on Palmetto. The same trace files are used by HECIOS with and without the scalable directory service. Thus the changes we made to PVFS in HECIOS is transparent to user programs and PVFS I/O semantics are maintained.

- **Achieve high throughput and scalability**: Our scalable directory service shows good scalability and high throughput under highly concurrent access to large directories in PVFS on HECIOS. We are able to achieve 9000 operations per second by using 512 compute nodes and 32 I/O nodes. The performance can be further improved by scaling the number of compute and I/O nodes.

- **Minimize bottlenecks and synchronization overheads**: We are able to remove directory operation bottlenecks by distributing directory entries among multiple meta-

data servers. Synchronization overhead as small as one-time cost is demonstrated in pre-distributed directories. Plus, synchronization overhead is minimized be design so that it does not prevent the system performance from scaling with the number of I/O nodes.

6.6 Future Works

Based on our research, some of the possible future works can be carried out in the following fields:

- Since the directory entry distribution is greatly decided by the hash function that is used, it is desirable to study the impact of using different hash functions with different file name and file access patterns on system performance.
- The directory splitting is controlled by the splitting threshold settings on the metadata servers. It will be interesting to further explore the effect of different setting on system performance.
- This work is finished in a parallel file system simulator. Thus it will be good progress to validate our design by implementing it in real file systems like PVFS.

Bibliography

[1] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[2] AT&T. *UNIX Programmer's Manual System Calls and Library Routines*, volume 2. Holt, Rinehart, and Winston, New York, NY, USA, 1986.

[3] P. J. Braam. The lustre storage architecture, 2002.

[4] Philip H. Carns. *Achieving scalability in parallel file systems*. PhD thesis, Clemson University, Clemson, SC, USA, 2005. Adviser-Ligon,III, Walter B.

[5] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: a parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[6] Clemson University CCIT. The palmetto architecture. http://citi.clemson.edu/palm_arch.

[7] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992.

[8] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[9] OMNeT++ Community. Omnet++. http://www.omnetpp.org.

[10] Helen Custer. *Inside Windows NT*. Microsoft Press, Bellevue, WA, USA, 1993. The authoritative technical reference on Windows NT.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03: proceedings of the 19th ACM Symposium on Operating Systems Principles: the Sagamore, Bolton Landing, Lake George, New York, USA, October 19–22, 2003*, pages 29–43, 2003.

[12] W. Gropp, E. Lusk, R. Ross, and R. Thakur. Using mpi-2: Advanced features of the message passing interface. *Cluster Computing, IEEE International Conference on*, 0:xix, 2003.

[13] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, TX, USA, November 8–11 1987.

[14] Joseph L. Hammond and Peter J.P. O'Reilly. *Performance analysis of local computer networks*. Addison-Wesley Pub. Co., 1986.

[15] Sanket Hase, Aditya Jayaraman, Vinay K. Perneti, Sundararaman Sridharan, Swapnil V. Patil, Milo Polte, and Garth A. Gibson. User level implementation of scalable directories (giga+). Technical Report CMU-PDL-08-107, Carnegie Mellon University, May 2008.

[16] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, January 17–21 1994.

[17] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Directory-based metadata optimizations for small files in pvfs. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 90–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Los Alamos National Laboratory. Roadrunner, world's first petaflop/s system, remains #1 on the top500 and #7 on the green500. Internet.

[19] John D. C. Little. A proof for the queuing formula: L= Îż w. *Operations Research*, 9(3):383–387, 1961.

[20] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[21] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[22] Clemson University PARL, the Parallel Architecture Research Labortary. The high-end computing i/o simulator. http://www.parl.clemson.edu/hecios.

[23] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 26–29, New York, NY, USA, 2007. ACM.

[24] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, May 1990.

[25] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[26] Bradles Settlemyer. *A study of client-based caching for parallel I/O*. PhD thesis, Clemson University, August 2009.

[27] Vertas Software. Vxfs. http://www.veritas.com.

[28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, January 22–26 1996.

[29] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, Upper Saddle River, NJ 07458, USA, third edition, 2008.

[30] Theodore Y. Ts'o and Stephen Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–243, Berkeley, CA, USA, 2002. USENIX Association.