RICE UNIVERSITY

A Storage Architecture for Data-Intensive Computing

by

Jeffrey Shafer

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Scott Rixner, Chair
Associate Professor of Computer Science
and Electrical and Computer Engineering

Alan L. Cox
Associate Professor of Computer Science
and Electrical and Computer Engineering

Peter J. Varman
Professor of Electrical and Computer
Engineering and Computer Science

HOUSTON, TEXAS

MAY 2010

Abstract

A Storage Architecture for Data-Intensive Computing

by

Jeffrey Shafer

The assimilation of computing into our daily lives is enabling the generation of data at unprecedented rates. In 2008, IDC estimated that the "digital universe" contained 486 exabytes of data [9]. The computing industry is being challenged to develop methods for the cost-effective processing of data at these large scales. The MapReduce programming model has emerged as a scalable way to perform data-intensive computations on commodity cluster computers. Hadoop is a popular open-source implementation of MapReduce. To manage storage resources across the cluster, Hadoop uses a distributed user-level filesystem. This filesystem — HDFS — is written in Java and designed for portability across heterogeneous hardware and software platforms. The efficiency of a Hadoop cluster depends heavily on the performance of this underlying storage system.

This thesis is the first to analyze the interactions between Hadoop and storage. It describes how the user-level Hadoop filesystem, instead of efficiently capturing the full performance potential of the underlying cluster hardware, actually degrades application performance significantly. Architectural bottlenecks in the Hadoop implementation result in inefficient HDFS usage due to delays in scheduling new MapReduce tasks. Further, HDFS implicitly makes assumptions about

how the underlying native platform manages storage resources, even though native filesystems and I/O schedulers vary widely in design and behavior. Methods to eliminate these bottlenecks in HDFS are proposed and evaluated both in terms of their application performance improvement and impact on the portability of the Hadoop framework.

In addition to improving the performance and efficiency of the Hadoop storage system, this thesis also focuses on improving its flexibility. The goal is to allow Hadoop to coexist in cluster computers shared with a variety of other applications through the use of virtualization technology. The introduction of virtualization breaks the traditional Hadoop storage architecture, where persistent HDFS data is stored on local disks installed directly in the computation nodes. To overcome this challenge, a new flexible network-based storage architecture is proposed, along with changes to the HDFS framework. Network-based storage enables Hadoop to operate efficiently in a dynamic virtualized environment and furthers the spread of the MapReduce parallel programming model to new applications.

Acknowledgments

I would like to thank the many people who contributed to this thesis and related works. First, I thank my committee members, including Dr. Scott Rixner for technical guidance in computer architecture, Dr. Alan L. Cox for his deep understanding of operating system internals, and Dr. Peter Varman for his insights and experience in storage systems. Second, I thank Michael Foss, with whom I collaborated and co-developed the Axon network device, which introduced me to datacenter technologies. Third, I thank the current and former graduate students in the Rice Computer Science systems group, including Thomas Barr, Beth Crompton, Hyong-Youb Kim, Kaushik Kumar Ram, Brent Stephens, and Paul Willmann for their feedback and advice at numerous practice talks and casual conversations throughout graduate school. Fourth, I thank my parents, Ken and Ruth Ann Shafer, for supporting my educational efforts and providing a caring and nurturing home environment. Finally, I thank Lysa for her enduring love, patience, and support through our many months apart. Thank you.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

New applications that store and analyze huge quantities of data are regularly emerging in the fields of commerce, science, and engineering. For example, consider scientific applications written for the Large Hadron Collider, an instrument expected to produce 15 petabytes of data per year during normal operation. Or, consider search and indexing applications for the Internet Archive, which currently stores 2 petabytes worth of archive data and is growing at a rate of 20 terabytes a month [43]. Data-intensive Computing (DC) applications such as these have significant value to consumers, scientists, governments, and corporations, and have motivated the development of programming techniques and cluster computer architectures to store, search, and manipulate these massive datasets.

DC applications generally are embarrassingly parallel, and can easily scale to hundreds or thousands of loosely synchronized processors. These applications exploit parallel programming models so that the work can be dynamically distributed to many computing elements, each of which are responsible for solving a small part of the entire problem [2, 3, 29, 32, 35, 63]. Because of the loose synchro-

nization requirements, application performance can be scaled almost linearly by increasing the available computation resources. This style of application programming has motivated the development of a new class of cluster computer. In these DC clusters, unlike traditional cluster supercomputers, it is more cost effective to increase the total number of compute nodes in the system than to increase the performance of each node. This encourages compute nodes to be constructed out of commodity components to reduce the per-node cost.

The MapReduce programming model, in particular, has emerged as a scalable way to perform data-intensive computations on a commodity cluster computer [35, 37]. It was developed at Google to support their web indexing and search applications, but has subsequently been used to support many other services. The success of the proprietary Google implementation of MapReduce has inspired the creation of Hadoop, a popular open-source alternative [2]. Written in Java for portability across heterogeneous hardware and software platforms, Hadoop is employed today by a wide range of commercial and academic users for backend data processing. A key component of Hadoop is the Hadoop Distributed File System (HDFS), which is used to store all input and output data for applications.

When designing storage systems for DC clusters, raw capacity is of utmost importance, as datasets can range in size from hundreds of terabytes to dozens of petabytes [43]. Given these massive data sets, a key premise in DC architecture design has been that there is insufficient network bandwidth to move the data to the

computation, and thus computation must move to the data instead. Based on the assumption that remote data can only be accessed with low bandwidth and high latency, current MapReduce architectures co-locate computation and storage in the same physical box. Although storage is shared across the network via a global file system that performs replication and load balancing, the goal of the task scheduler is to migrate computation to use data on locally-attached disks whenever possible.

The efficiency of the MapReduce model has been questioned in recent research contrasting it with the parallel database paradigm for large-scale data analysis. Typically, Hadoop is used as representative of the MapReduce model because proprietary (*e.g.*, Google-developed) implementations with potentially higher performance are not publicly available. In one study, Hadoop applications performed poorly when compared against applications using parallel databases, despite accomplishing the same tasks [67, 77]. For example, Figure 1.1 taken from the study uses a simple test application performing a data aggregation task, and compares the execution time of that application on two parallel databases and the Hadoop MapReduce framework. The application was tested at varying cluster sizes, with a constant amount of data per node, to evaluate scalability. As shown, Hadoop was at least twice as slow as the parallel databases at performing the same task. This gap was attributed to differences in the high-level programming model. However, this work did not perform the profiling necessary to distinguish the fundamental performance of the MapReduce programming model from a specific implementa-

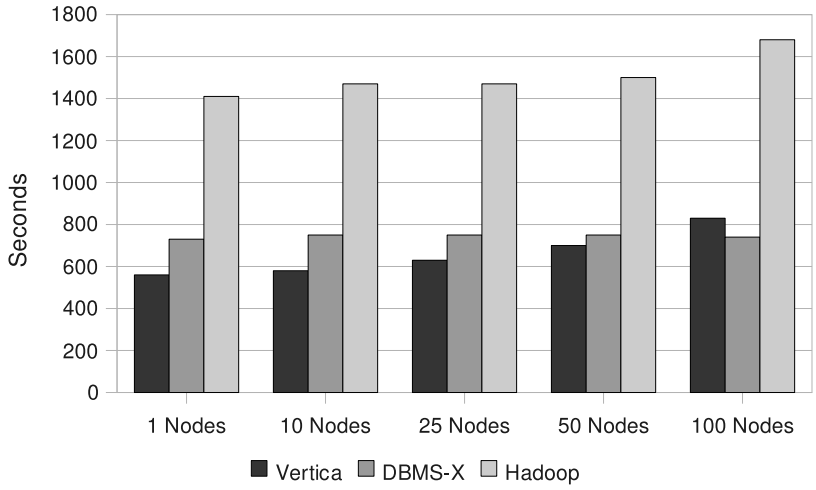


Figure 1.1: Aggregation Application Performance on Parallel Databases and Hadoop / MapReduce (Lower is Better) — Appeared as Figure 7 in [67]

tion, *i.e.*, Hadoop. A characterization of the Hadoop framework performed in this thesis shows that it is actually the implementation of the Hadoop storage system that degrades performance significantly.

The performance penalty incurred by HDFS can be easily demonstrated by comparing disk storage bandwidth in the native operating system against disk storage bandwidth inside Hadoop, using the same disk and host system. Table 1.1 shows the storage bandwidth of two simple test applications that write 10GB of data to disk with large, sequential accesses, and subsequently read it back. One application was run in the native operating system, and the other equivalent application was run in the Hadoop environment with and without data replication. While the test application running in the native operating system achieves full disk bandwidth, the application accessing data in HDFS without replication only achieves 70% of the original bandwidth. When replication is enabled, resulting

Environment	Replication	Bandwidth (MB/s)	
		Write	Read
Native Application	N/A	95	105
Hadoop Application	No	68	72
Hadoop Application	Yes	36	54

Table 1.1: Native versus Hadoop Storage Bandwidth (MB/s) for Synthetic Test Applications

in 3 copies of the data being saved to disk concurrently, the aggregate bandwidth (total of all 3 copies) degrades further, achieving only 37-50% of the native disk bandwidth. Thus, the Hadoop storage architecture imposes a substantial performance penalty and fails to provide the full performance of the underlying storage hardware to the application layer.

1.1 Contributions

This thesis contributes to the field of data-intensive computing in several ways. First, it focuses on improving the performance and efficiency of the Hadoop storage system using the traditional local disk architecture where storage and computation are co-located in the same box. Eliminating HDFS bottlenecks not only boosts application performance, but also improves overall cluster efficiency, thereby reducing power and cooling costs and allowing more computation to be accomplished with the same number of cluster nodes. Second, it explores the challenges in spreading the MapReduce programming paradigm to smaller or more intermittent jobs. Finally, it introduces a new architecture for persistent HDFS storage using networked disks that allows Hadoop to function effectively in a virtual-

ized and shared cluster computing environment. The specific contributions are as follows:

Characterization of Hadoop Performance — This thesis is the first to analyze the interactions between Hadoop and storage. It describes how the user-level Hadoop filesystem, instead of efficiently capturing the full performance potential of the underlying cluster hardware, actually degrades application performance significantly.

Increasing Disk Utilization by Identifying and Eliminating Software Architectural Bottlenecks — HDFS is not utilized to its full potential due to scheduling delays in the Hadoop architecture that result in cluster nodes waiting for new tasks. The impact of this is that the disk is utilized in a periodic, not continuous fashion, and sits idle for significant periods. A variety of techniques are applied to this problem to reduce the task scheduling latency and frequency at which new tasks need to be scheduled, thereby increasing disk utilization to near 100%.

Increasing Disk Efficiency and Identifying Tradeoffs Related to Portability and Performance — After increasing disk utilization, the disk efficiency is also examined. This is directly related to Hadoop's goal of providing a portable MapReduce framework. The classic notion of software portability is simple: does the application run on multiple platforms? But, a broader notion of portability is: does the application perform well on multiple platforms? While HDFS is strictly portable, its performance is highly dependent on the behavior of underly-

ing software layers, specifically the OS I/O scheduler and native filesystem allocation algorithm. These components of the native operating system are designed for general-purpose workloads, not data-intensive computing. As such, they produce excessive disk seeks and fragmentation, degrading storage bandwidth significantly. Further, some performance-enhancing features in the native filesystem are not available in Java in a platform-independent manner. This includes options such as bypassing the filesystem page cache and transferring data directly from disk into user buffers. As such, the HDFS implementation runs less efficiently and has higher processor usage than would otherwise be necessary. A variety of portable and non-portable methods are described and evaluated in order to increase the efficiency at which the underlying storage system is used.

Spreading the MapReduce Model — MapReduce was designed (by Google, Yahoo, and others) to marshal all the storage and computation resources of a dedicated cluster computer. Unfortunately, such a design limits this programming paradigm to only the largest users with the financial resources and application demand to justify deployment. Smaller users could benefit from the MapReduce programming model too, but need to run it on a cluster computer shared with other applications through the use of virtualization technologies. The traditional storage architecture for MapReduce that places persistent HDFS data on locally-attached disks is evaluated and deemed unsuitable for this new workload, motivating a fresh look at alternative architectures.

Evaluating Persistent Network-Based Storage for MapReduce — Network-based storage is proposed to allow MapReduce to coexist in a shared datacenter environment. Network-based storage offers advantages in terms of resource provisioning, load balancing, fault tolerance, and power management. Network-based storage is feasible for Hadoop for several reasons. First, data-intensive computing applications access storage using streaming access patterns and thus are bandwidth, not latency, sensitive. Second, network bandwidth historically has exceeded disk bandwidth for commodity technologies. Third, modern switches offer extremely high bandwidth and low latency to match that of the raw network links, unlocking fast connectivity to devices co-located in the same rack and connected to the same switch.

Design for Remote Storage Architecture — A design space analysis is performed for potential Hadoop network storage architectures. These architectures meet high-level constraints, such as using commodity hardware and a single network to lower installation and administration costs, and providing a scalable design without centralized bottlenecks. These designs are evaluated in terms of achieved storage bandwidth and processor overhead in order to determine the most efficient design that incurs the least overhead over the conventional Hadoop local storage architecture. The most efficient design takes advantage of the existing Hadoop network capabilities for data replication. Optimizations to the Hadoop filesystem scheduler are evaluated to reduce contention for network storage re-

sources and thereby improve performance. Network and storage bandwidth inside of a virtual machine is analyzed and optimized to ensure that the new remote storage architecture functions efficiently inside a virtualized environment.

1.2 Organization

This thesis is organized as follows. Chapter 2 provides a background into the Hadoop framework and global filesystem, and discusses related work to the traditional local storage architecture used in MapReduce computation. Chapter 3 characterizes the performance of Hadoop and its storage system utilizing locally-attached disks. In this chapter, bottlenecks are identified that degrade disk utilization and efficiency, thus slowing application performance. Chapter 4 proposes and evaluates architectural changes to Hadoop to improve storage system efficiency and performance. Chapter 5 discusses a broad history of network-based storage architectures. Next, Chapter 6 motivates a new storage architecture using networked disks to allow MapReduce to co-exist with other applications in a datacenter running a virtualization framework. Chapter 7 performs a design space analysis of several realizations for network storage, evaluates the architectures in terms of performance and computational efficiency, and examines modifications to the Hadoop framework to reduce resource contention and improve performance. Finally, Chapter 8 concludes this thesis.

CHAPTER 2

Hadoop Background

As the demand for data-intensive computing grows, the number and scale of DC clusters is increasing. One of the key elements of making such clusters both inexpensive and highly utilized is appropriate software frameworks and application programming models. The MapReduce programming model has emerged as an easy write to write scalable embarrassingly parallel applications that can exploit large commodity clusters for data-intensive computations [35]. MapReduce is designed to enable scalability by allowing each node to process its slice of the overall dataset with only loose coordination with other nodes. With this programming model, increasing the number of compute nodes increases the amount of parallelism that can be exploited and improves overall application performance.

Hadoop [2] is an open source framework that implements the MapReduce parallel programming model [35]. Hadoop was chosen for this thesis for several reasons. First, it is popular and in widespread use today by a number of leading Internet service companies, including Amazon, Facebook, Yahoo, and others. Second, it has a history of large-scale deployments, including a Yahoo cluster with

over 4000 nodes [6]. Third, it is designed for commodity hardware, significantly lowering the expense of building a research cluster. Google has shown in their web indexing framework that a specialized supercomputer is not needed for data-intensive computing, just a large number of commodity computers networked together. Fourth, Hadoop is open source, making it easier to obtain, profile, and modify when necessary. Fifth, the design philosophy used in Hadoop is similar to other DC frameworks [42, 68]. Thus, this research into the architecture of Hadoop and its filesystem should be applicable to similar systems.

The Hadoop framework is composed of a MapReduce engine and a user-level filesystem that manages storage resources across the cluster. For portability across a variety of platforms — Linux, FreeBSD, Mac OS/X, Solaris, and Windows — and ease of installation, both components are written in Java and only require commodity hardware. Here, the architecture of a Hadoop cluster is described, along with the operation of its various software services for computation and storage. Further, related work to Hadoop is presented, including the Google implementation of the MapReduce programming model, other file storage architectures, and databases and streaming media servers.

2.1 Hadoop Cluster Architecture

Given the highly parallelizable nature of the MapReduce computation model, it becomes relatively straightforward to exploit large clusters to increase application

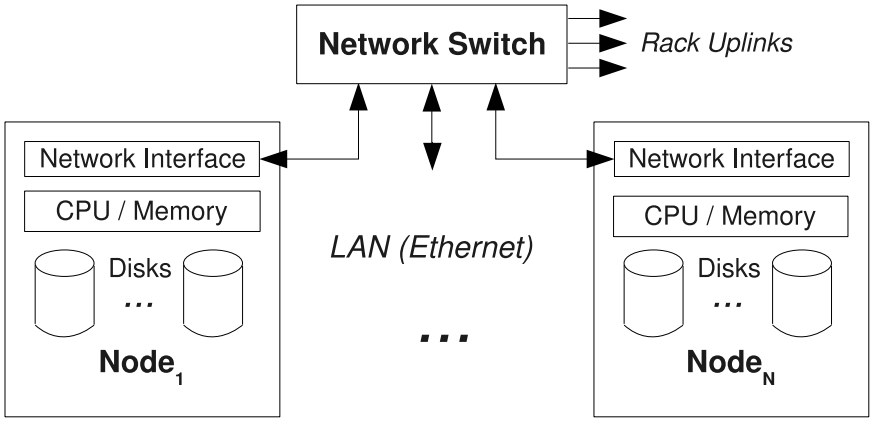


Figure 2.1: Hadoop Cluster Architecture (Single Rack)

throughput. Therefore, in a DC cluster running this type of application, the capability of each node is less important than the ability to scale the number of nodes in the cluster. Based on this philosophy, DC systems designed to run frameworks like Hadoop are built with the following commodity technologies in mind: x86-based processors, Ethernet networks, and Serial ATA (SATA) hard disks. Any other technology choice increases the per-node cost of the cluster and thus limits the number of nodes that can be economically purchased and utilized. For example, although Solid State Drives (SSDs) built on flash memory are expanding their presence in the storage marketplace thanks to impressive performance, SSDs are not suitable for use in DC clusters due to their expense. Flash-based storage will not match the capacity/dollar metric of disk-based storage for the foreseeable future [48, 61].

In such a cluster, storage bandwidth becomes a first order determinant of overall system performance [25, 32]. Each map or reduce task must have sufficient storage bandwidth available to efficiently complete its task on a given computa-

tion node. This requirement has led to a cluster architecture in which local disks in the computation nodes are utilized as part of a distributed file system to store file blocks. To efficiently exploit local disk bandwidth, Hadoop attempts to schedule tasks on nodes which store that task's input data.

An example of the Hadoop architecture in a single rack is shown in Figure 2.1. As the figure shows, each computation node is equipped with one or more disks and the nodes are interconnected with a commodity Ethernet network. A single rack is likely to be interconnected by a single, high-performance Ethernet switch which provides full bandwidth among all of the nodes within the rack. Each rack is then connected to the other racks through a hierarchy of Ethernet switches, with far less inter-rack bandwidth due to cost and cabling constraints. While any node can communicate with any other node, there is far more bandwidth available within a rack than across racks.

2.2 Hadoop MapReduce Engine

In the MapReduce model, computation is divided into a *map* stage and a *reduce* stage. In the map stage, the data to be processed is divided into many pieces and assigned (*i.e.*, mapped) to specific cluster nodes, each of which can work independently with minimal coordination. In the reduce stage, the output from the map stage on each node is read and combined to produce the final program output.

Both the map and reduce stages process data in the form of key/values pairs.

The map stage reads in input key/value pairs and produces one or more intermediate key/value pairs. This intermediary data is saved to memory, or spilled to local disk temporarily until the reduce stage executes. The reduce stage then takes these intermediate key/value pairs and merges all values corresponding to a single key. The map function can run independently on each key/value pair, exposing enormous amounts of parallelism. Similarly, the reduce function can run independently on each intermediate key value, also exposing significant parallelism.

In Hadoop, the MapReduce engine is implemented by two software services, the *JobTracker* and *TaskTracker*. The centralized JobTracker runs on a dedicated cluster node and is responsible for splitting the input data into pieces for processing by independent map and reduce tasks (by coordinating with the user-level filesystem), scheduling each task on a cluster node for execution, monitoring execution progress by receiving heartbeat signals from cluster nodes, and recovering from failures by re-running tasks. On each cluster node, an instance of the TaskTracker service accepts map and reduce tasks from the JobTracker. By default, when a new task is received, a new JVM instance will be spawned to execute it. Each TaskTracker will periodically contact the JobTracker via a heartbeat message to report task completion progress and request additional tasks when idle.

2.3 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) provides global access to files in the cluster [4, 78]. Map tasks can read input data from HDFS, and reduce tasks can save output data to HDFS. As previously mentioned, intermediary data between Map and Reduce tasks is not stored in HDFS, but instead resides on each local node in temporary storage. For maximum portability and ease of installation, HDFS is implemented as a user-level filesystem in Java which exploits the native filesystem on each node, such as ext3 or NTFS, to store data. Files in HDFS are divided into large blocks, typically 64MB, and each block is stored as a separate file in the local filesystem.

HDFS is implemented by two services: the *NameNode* and *DataNode*. The *NameNode* is responsible for maintaining the HDFS directory tree, and is a centralized service in the cluster operating on a single node. Clients contact the NameNode in order to perform common filesystem operations, such as open, close, rename, and delete. The NameNode does not store HDFS data itself, but rather maintains a mapping between HDFS file name, a list of blocks in the file, and the DataNode(s) on which those blocks are stored.

In addition to a centralized NameNode, all remaining cluster nodes provide the *DataNode* service. Each DataNode stores HDFS blocks on behalf of local or remote clients. Each block is saved as a separate file in the node's local filesystem. Because the DataNode abstracts away details of the local storage arrangement, all

nodes do not have to use the same local filesystem. Blocks are created or destroyed on DataNodes at the request of the NameNode, which validates and processes requests from clients. Although the NameNode manages the namespace, clients communicate directly with DataNodes in order to read or write data at the HDFS block level.

Hadoop MapReduce applications use storage in a manner that is different from general-purpose computing [42]. First, the data files accessed are large, typically tens to hundreds of gigabytes in size. Second, these files are manipulated with streaming access patterns typical of batch-processing workloads. When reading files, large data segments (several hundred kilobytes or more) are retrieved per operation, with successive requests from the same client iterating through a file region sequentially. Similarly, files are also written in a sequential manner.

This emphasis on streaming workloads is evident in the design of HDFS. First, a simple coherence model (write-once, read-many) is used that does not allow data to be modified once written. This is well suited to the streaming access pattern of target applications, and improves cluster scaling by simplifying synchronization requirements. Second, each file in HDFS is divided into large blocks for storage and access, typically 64MB in size. Portions of the file can be stored on different cluster nodes, balancing storage resources and demand. Manipulating data at this granularity is efficient because streaming-style applications are likely to read or write the entire block before moving on to the next. In addition, this design choice

improves performance by decreasing the amount of metadata that must be tracked in the filesystem, and allows access latency to be amortized over a large volume of data. Thus, the filesystem is optimized for high bandwidth instead of low latency. This allows non-interactive applications to process data at the fastest rate.

To read an HDFS file, client applications simply use a standard Java file input stream, as if the file was in the native filesystem. Behind the scenes, however, this stream is manipulated to retrieve data from HDFS instead. First, the NameNode is contacted to request access permission. If granted, the NameNode will translate the HDFS filename into a list of the HDFS block IDs comprising that file and a list of DataNodes that store each block, and return the lists to the client. Next, the client opens a connection to the closest DataNode and requests a specific block ID. That HDFS block is returned over the same connection, and the data delivered to the application. Ideally, the closest DataNode is the same node where the client application is already running, thus reducing the amount of network traffic in the cluster. If the data is not available locally, Hadoop falls back on its rack-awareness algorithm. Cluster administrators can configure Hadoop with knowledge of the arrangement of physical nodes in the cluster, specifically what nodes are physically adjacent in the same rack and connected to the same network switch. Because intra-rack network bandwidth is greater than inter-rack network bandwidth, due to limited uplink bandwidth in the hierarchical network, the HDFS framework will try to read data from a node within the same rack (connected to the same network

switch) whenever possible.

To write data to HDFS, client applications see the HDFS file as a standard output stream. This abstraction hides a great deal of complexity in the Hadoop framework, however. Three threads perform a variety of tasks related to writing HDFS data to disk. The first thread, the client-facing thread, first fragments the data stream into HDFS-sized blocks (64MB) and then into smaller packets (64kB). Each packet is enqueued into a FIFO that can hold up to 5MB of data, thus decoupling the client thread from storage system latency during normal operation. A second thread is responsible for dequeuing packets from the FIFO, coordinating with the NameNode to assign HDFS block IDs and destinations, and transmitting blocks to the DataNodes (either local or remote) for storage. A third thread manages acknowledgements from the DataNodes that data has been committed to disk.

For reliability, HDFS implements an automatic replication system. By default, two copies of each block are stored by different DataNodes in the same rack and a third copy is stored on a DataNode in a different rack (for greater reliability). Thus, in normal cluster operation, each DataNode is servicing both local and remote clients simultaneously. HDFS replication is transparent to the client application. When writing a block, a pipeline is established whereby the client only communicates with the first DataNode, which then echos the data to a second DataNode, and so on, until the desired number of replicas have been created. The write operation is only finished when all nodes in this replication pipeline have successfully

committed all data to disk. DataNodes periodically report a list of all blocks stored to the NameNode, which will verify that each file is sufficiently replicated and, in the case of failure, instruct DataNodes to make additional copies.

2.4 Related Work

This thesis focuses on the storage architecture of data-intensive computing clusters. As such, it builds upon prior work in a number of related areas. Topics related to the traditional Hadoop local storage architecture are discussed here, while topics related to the proposed network-based storage architecture are discussed later in Chapter 5.

Here, the original Google File System for data-intensive computing is first described and compared to the open-source HDFS implementation. The similarities between these systems are high, and many HDFS optimizations are equally applicable to the Google architecture. Second, the HDFS storage architecture is contrasted with the traditional file server model for storage. The need to scale to large cluster sizes makes the file server model impractical for MapReduce. Third, the design requirements for HDFS storage are compared with those for databases and streaming media servers. Here, discussion focuses on how those differences translate into the specific storage architecture used and techniques to accelerate performance. User-space filesystems are also discussed as one potential way to overcome the limitations of general-purpose filesystems for specific application

workloads, such as data-intensive computing.

2.4.1 Google File System

Hadoop and HDFS were conceived as open-source implementations of the Google MapReduce engine [35, 36, 37] and the Google File System (GFS) [42], respectively. The MapReduce programming model for data-intensive computing was developed at Google from earlier functional programming research, and it plays an important role in the operations of this Internet giant. The most recently published report indicates that, by 2008, Google was running over one hundred thousand MapReduce jobs per day and processing over 20 PB of data in the same period [36]. By 2010, Google had created over ten thousand distinct MapReduce programs performing a variety of functions, including large-scale graph processing, text processing, machine learning, and statistical machine translation [37]. In this section, the similarities and differences between HDFS and GFS are discussed. Overall, the differences are minor, meaning that many of the optimizations applied to HDFS in this thesis are equally applicable to the GFS architecture.

HDFS and GFS have common design goals. They are both targeted at data-intensive computing applications where massive data files are common. Both are optimized in favor of high sustained bandwidths instead of low latency, to better support batch-processing style workloads. Both run on clusters built with commodity hardware components where failures are common, motivating the inclu-

sion of built-in fault tolerance mechanisms through replication. Both filesystems provide applications with a write-once, read-many API that eschews full POSIX compliance in favor of design simplicity. Finally, both implementations provide no data caching. Clients experience little re-use because they either stream through a file, or have working sets that are too large.

By virtue of these common design goals, HDFS and GFS are also implemented in a similar manner. In both systems, the filesystem is implemented by user-level processes running on top of a standard operating system (in the case of GFS, Linux). A single GFS master server running on a dedicated node is used to coordinate storage resources and manage metadata. Multiple slave servers (*chunkservers* in Google parlance) are used in the cluster to store data in the form of large blocks (*chunks*), each identified with a 64-bit ID. Files are saved by the chunkservers on local disk as native Linux files, and accessed by chunk ID and offset within the chunk. Both HDFS and GFS use the same default chunk size (64MB) to reduce the amount of metadata needed to describe massive files, and to allows clients to interact less often with the single master. Finally, both use a similar replica placement policy that saves copies of data in many locations — locally, to the same rack, and to a remote rack — to provide fault tolerance and improve performance by reducing hot spots.

Although HDFS and GFS share many similarities, they are not exact clones of each other, and differ in a few ways. First, HDFS does not currently provide an

equivalent to the atomic file append functionality as available in GFS, although the implementation of this feature is on-going. Atomic file appends allow many concurrent writers to each append data to an otherwise immutable file without specifying the exact offset to write data to. GFS calculates the end of the file automatically while ensuring that the entire write operation is atomic. Second, HDFS does not yet provide an equivalent file or directory snapshot feature to HDFS. Snapshots can be used to make a copy of data without interfering with ongoing appends. GFS uses a copy-on-write framework to accomplish this. Finally, GFS does not provide the high level of portability provided in HDFS. GFS was reported to only run on Linux, and was not implemented in Java. From a corporate perspective, it is easier to standardize proprietary technology like GFS and Google MapReduce to run only on a specific OS (*e.g.,* a customized fork of the Linux kernel) rather than support a wide variety of host environments.

Optimizations proposed later in this thesis for Hadoop are equally applicable to the Google-developed MapReduce implementation that is not publicly available. In fact, they may already be present in some form. The optimizations described for the Google implementation include reducing disk seeks for writes by batching and sorting intermediate data, and reducing disk seeks for reads by smart scheduling of requests [37]. Further, it is reasonable to assume that Google also employs non-portable optimizations to improve performance further, such as tuning filesystem behavior to increase extents and thereby reduce fragmentation and disk seeks, and

bypassing the operating system page cache and transferring data directly from the disk into the user-space application buffer. These have not been described in public documents. Google has a history of extensive Linux kernel modifications, as most recently described at the Linux Kernel Summit in 2009 [7]. Storage-related improvements that have been made to the kernel — but not yet released or described in any detail — include proportional I/O scheduling, tracing of disk accesses for operations analysis, and lowering the system call overhead of *fadvise()* to provide caching hints to the operating system.

2.4.2 File Server Model

In Hadoop, both MapReduce (*i.e.*, TaskTracker) and storage (*i.e.*, DataNode) services are typically executed on the same set of cluster nodes, allowing computation to access local storage resources at high bandwidth. This use of local storage in Hadoop runs counter to the prevalent file server model. In that model, a single file server, perhaps with a small number of backup servers for redundancy, provides access to a large number of clients across a network. Distributed storage systems [23, 39, 46, 49] are meant to alleviate the performance and reliability problems associated with a centralized file server, which is a single point of entry into the file system. Load balancing can be utilized on such distributed systems in order to distribute accesses to several servers and improve overall storage system performance [22, 83]. However, with the exception of [23], such distributed filesys-

tems do not typically include storage on the clients as a part of the file system.

The file server model is not compatible with the ever-increasing scale of Hadoop systems, particularly at Internet service companies. The cost of a server-based distributed file system that can scale to support the needs of a large Hadoop cluster are likely to be prohibitive. As an example of the scale of these systems, Yahoo announced its largest Hadoop cluster in September 2008, consisting of 4000 nodes, each with 2 quad-core x86 processors, 4 1TB SATA disks, 8GB of RAM, and a 1 gigabit Ethernet port. Each rack contained 40 compute nodes, and the rack switch had 4 gigabit Ethernet uplinks to the core network. Overall, the Hadoop cluster contained in excess of 30,000 processor cores and 16PB of raw disk capacity [6]. In this type of system, a conventional centralized or distributed file server would create a bandwidth bottleneck that would place severe limits on the peak performance of the cluster. Instead, the massive distributed storage within the compute nodes is incorporated directly into a serverless global file system provided by HDFS or the Google file system [4, 42].

2.4.3 Databases and Streaming Media Servers

HDFS servers (*i.e.*, DataNodes) and traditional streaming media servers are both used to support client applications that have access patterns characterized by long sequential reads and writes. As such, both systems are architected to favor high storage bandwidth over low access latency [70]. Beyond this, however,

there are key requirements that differentiate streaming media servers from HDFS servers. First, streaming media servers need to rate pace to ensure that the maximum number of concurrent clients receives the desired service level. In contrast, MapReduce clients running batch-processing non-interactive applications are latency insensitive, allowing the storage system to maximize overall bandwidth, and thus cluster cost-efficiency. Second, media servers often support differentiated service levels to different request streams, while in HDFS all clients have equal priority. Taken collectively, these requirements have motivated the design of a large number of disk scheduling algorithms for media servers [18, 30, 52, 71, 70, 76]. Each algorithm makes different tradeoffs in the goals of providing scheduler fairness, meeting hard or soft service deadlines, reducing memory buffer requirements, and minimizing drive seeks.

In addition to similarities with streaming media servers, HDFS servers also share similarities with databases in that both are used for data-intensive computing applications [67]. But, databases typically make different design choices that favor performance instead of portability. First, while Hadoop is written in Java for portability, databases are typically written in low-level application languages to maximize performance. Second, while Hadoop only uses Java native file I/O features, commercial databases exploit OS-specific calls to optimize filesystem performance for a particular platform by configuring or bypassing the kernel page cache, utilizing direct I/O, and manipulating file locking at the inode level [38, 53].

Third, while HDFS relies on the native filesystem for portability, many well-known databases can be configured to directly manage storage as raw disks at the application level, bypassing the filesystem entirely [1, 47, 62]. Using storage in this manner allows the filesystem page cache to be bypassed in favor of an application cache, which eliminates double-buffering of data. Further, circumventing the filesystem provides the application fine-grained control over disk scheduling and allocation to reduce fragmentation and seeks. Thus, databases show the performance that can be gained if portability is sacrificed or if additional implementation effort is exerted to support multiple platforms in different manners.

One particular aspect of database design — application-level I/O scheduling — exploits application access patterns to maximize storage bandwidth in a way that is not similarly exploitable by HDFS. Application-level I/O scheduling is frequently used to improve database performance by reducing seeks in systems with large numbers of concurrent queries. Because database workloads often have data re-use (for example, on common indexes), storage usage can be reduced by sharing data between active queries [26, 84]. Here, part or all of the disk is continuously scanned in a sequential manner. Clients join the scan stream in-flight, leave after they have received all necessary data (not necessarily in-order), and never interrupt the stream by triggering immediate seeks. In this way, the highest overall throughput can be maintained for all queries. This particular type of scheduling is only beneficial when multiple clients each access some portion of shared data,

which is not common in many HDFS workloads.

Based on work done in database and streaming media systems, Wang *et al.* proposed the concept of UCFS – User-space Customized Filesystems – to overcome application-specific bottlenecks in filesystems designed for general-purpose computing [81]. In such a system, the application or library would directly manage the raw disk, thus bypassing any OS buffering and filesystem limitations, and allowing the on-disk file layout to be extensively customized to the application requirements. As an example, they implemented a custom filesystem and caching scheme for a web proxy server. Particular attention was paid to implementing clustering, grouping, and prefetching algorithms to ensure that all disk accesses are done in large blocks, that each cluster contains all cache payload and metadata necessary to satisfy a request without further seeks, and future requests are likely to be satisfied by the same or adjacent clusters on disk. Such techniques could also be beneficial in improving HDFS performance by eliminating the local filesystem. As shown later in this thesis, however, the best-case improvement in storage bandwidth made possible by removing the general-purpose filesystem is relatively small, and any replacement system implemented in user-space must incur some overhead of its own for bookkeeping. Thus, it is questionable whether the performance improvement can justify the development time of a UCFS.

2.4.4 Efficient Cluster Architectures

Researchers have recently focused on new cluster architectures for data-intensive computing using application frameworks such as MapReduce [35] and Dryad [50] that allow efficient parallel computation over massive data sets. These architectures are in contrast to traditional clusters composed of high power, high performance servers with a small number of high power (and hot) disks in the same chassis. But, they still use locally-attached storage co-located with computation resources.

Caulfield *et al.* proposed an architecture called Gordon where myriads of compute nodes are constructed of low-power, inexpensive, efficient processors such as an Intel Atom coupled with solid-state flash storage instead of conventional hard drives [27]. The goal of this architecture is to increase both overall performance (by leveraging the improved bandwidth and latency of solid state storage) and power efficiency (by balancing the storage bandwidth required by the processor with the bandwidth that can be sustained by the storage system). The low-power requirements and compact design of both the processor and storage system allows high-density clusters to be constructed, with a standard rack in the near future predicted to hold 256 compute nodes with 64TB of aggregate storage. A MapReduce computation framework is used to scale the application across these large numbers of processors. Although flash storage has desirable power and performance characteristics, the current cost premium over conventional disks limits its potential for

data-intensive computing applications. In addition, this architecture provides an even tighter coupling between computation and storage than conventional servers. Instead of placing separate processors, motherboards, and hard drives in a case as discrete components, this design places processors and flash memory chips on the same circuit board, leaving almost no flexibility to vary the ratio between computation and storage.

A similar architecture has also been proposed by Vasudevan *et al.* in the *FAWN* (Fast Array of Wimpy Nodes) project [79]. This architecture couples low-power embedded processors with a variety of storage systems, including compact flash, solid-state disks, conventional hard drives, and DRAMs. These designs have been analyzed in the context of two different types of workloads: scan-oriented workloads (as exemplified by MapReduce) and seek-oriented random-access small read workloads (as exemplified by databases and web applications utilizing *memcached*), with the conclusion that the optimal storage system varies depending on application requirements.

CHAPTER 3

Hadoop Local Performance Characterization

The performance of the storage system is of utmost importance in a DC cluster. In this thesis, the Hadoop distributed filesystem configured with local disks is first evaluated in order to identify bottlenecks that degrade application performance. Because this architecture represents the common cluster design today, performance optimizations to the local disk architecture will have a broad impact. Further, many of the bottlenecks uncovered apply equally to disks accessed across the network as well, thus improving the performance of remote storage architectures, discussed later in this thesis.

In this section, the experimental cluster and test applications used for performance characterization will be described. Then, a representative hard drive will be profiled outside of the Hadoop environment in order to determine the best-case performance possible from the raw cluster hardware. Next, several types of bottlenecks will be identified in the Hadoop framework. These include architectural bottlenecks that result in an inefficient periodic utilization of cluster resources, unnecessary processor overhead incurred by portability choices in the Hadoop im-

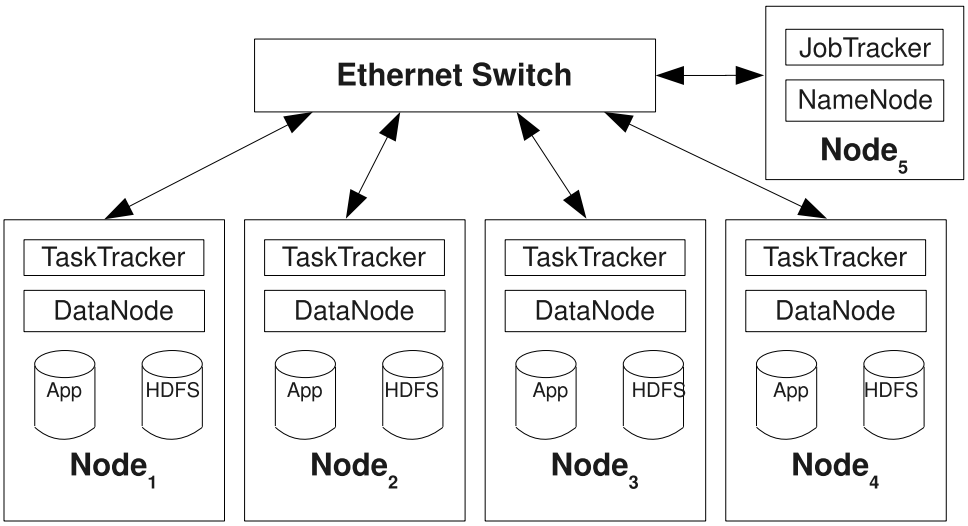


Figure 3.1: Cluster Setup

plementation, and excessive disk seeks and fragmentation caused by operating system components outside the direct control of Hadoop. Finally, the behavior of Hadoop running on top of other popular operating systems will be discussed to show that the performance problems identified here are widespread.

3.1 Experimental Setup

For performance characterization, a 5-node Hadoop cluster was configured, as shown in Figure 3.1. The first 4 nodes provided both computation (as MapReduce clients) and storage resources (as DataNode servers), and the 5th node served as both the MapReduce scheduler and NameNode storage manager. Each node was a 2-processor Opteron server running at 2.4 GHz or above with 4GB of RAM and a gigabit Ethernet NIC. All nodes used FreeBSD 7.2, Hadoop framework 0.20.0,

Code	Program	Data Size	Notes
S-Wr	Synthetic Write	10GB / node	Hadoop sequential write
S-Rd	Synthetic Read	10GB / node	Hadoop sequential read
Rnd-Text	Random Text Writer	10GB / node	Hadoop sequential write
Rnd-Bin	Random Binary Writer	10GB / node	Hadoop sequential write
Sort	Simple Sort	40GB / cluster	Hadoop sort of integer data
Search	Simple Search	40GB / cluster	Hadoop text search for rare string
AIO-Wr	Synthetic Write	10GB / node	Native C Program - Asynch. I/O
AIO-Rd	Synthetic Read	10GB / node	Native C program - Asynch. I/O

Table 3.1: Application Test Suite

and Java 1.6.0. The first four nodes were configured with two Seagate Barracuda 7200.11 500GB hard drives. One disk stored the operating system, Hadoop application, and application scratch space, while the second disk stored only HDFS data. All disks used the default UFS2 filesystem for FreeBSD with a 16kB block size and 2kB fragment size. An HP ProCurve 1800-24G Gigabit Ethernet switch was used to interconnect cluster nodes. Unless otherwise stated, Hadoop replication was disabled in order to focus on the efficiency with which Hadoop uses locally-attached (not network-attached) storage.

To characterize the Hadoop framework, a variety of test applications were installed as shown in Table 3.1. This test suite includes a simple HDFS synthetic writer and reader doing sequential streaming access, an HDFS writer that generates random binary numbers or text strings and writes them to the disk in a sequential fashion, a simple integer sort, and a simple search for a rare text pattern in a large file. Complex applications — like those used in industry — were not publicly available for use in this characterization. Hadoop is still a young platform

Model Number	ST3500320AS
Capacity	500GB
Rotation Speed	7200rpm
Interface	SATA
Features	Native Command Queuing
Seek time	8.5ms
Price in 2009	$70

Table 3.2: Commodity Hard Drive - Seagate Barracuda 7200.11

and thus an ecosystem of open-source applications or benchmarks has not yet developed. For comparison purposes, a program written in C was used to perform asynchronous I/O (AIO) on the raw disk to determine the best-case performance, independent of any Hadoop, Java, or filesystem-specific overheads.

Next, the latency and bandwidth characteristics of the hard drives used in the experimental cluster are profiled, in order to place an upper-bound on the performance of applications running inside the Hadoop framework.

3.2 Raw Disk Performance

A modern disk used for DC applications has a sequential bandwidth in excess of 100MB/s (0.8 Gb/s) and a seek time under 9ms. All cluster nodes were outfitted with the representative commodity hard drive shown in Table 3.2. To place an upper bound on Hadoop performance, the raw bandwidth of the commodity hard drive used in the cluster was measured independent of OS filesystem and cache effects.

To quantify the performance of the hard drive for sequential reads and writes

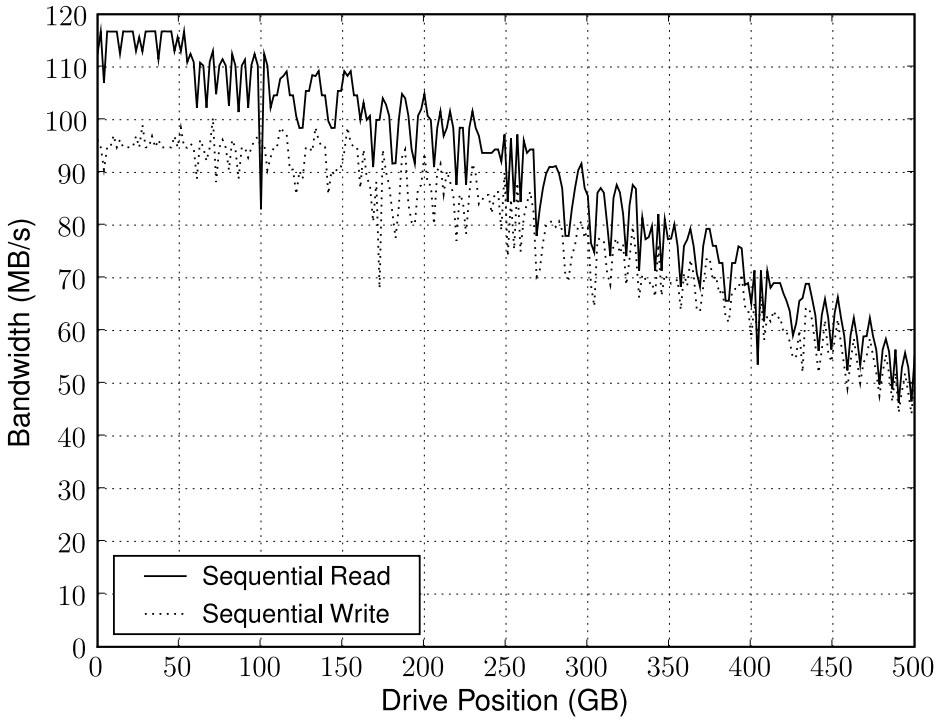


Figure 3.2: Hard Drive Bandwidth by Position (Large Sequential Accesses)

at various regions of the disk, the HDTach synthetic utility [11] was employed. A sequential test of peak I/O bandwidth matches the expected best-case streaming access patterns of DC workloads. The results of the drive test are shown in Figure 3.2.

By convention, logical block addresses on the drive are numbered such that the lowest address is placed at the outside edge of the drive, and the highest address is placed at the inside edge of the drive. Sectors stored on outer regions of the drive have the highest I/O bandwidth because more data is stored in the outer tracks while the angular velocity of the entire drive remains constant. This technique is called Zone Bit Recording or Zone Constant Angular Velocity. The commodity

hard drive tested has a sequential read speed in excess of 110MB/s and a sequential write speed approaching 100MB/s, and is able to sustain this I/O bandwidth over at least the first third of the drive. In all hard drives, write bandwidth is typically lower than read bandwidth due to the extra time needed to read back the data from the platter after writing and verify its correctness. The raw performance of the drive measured here represents the peak storage bandwidth that should be possible for any given Hadoop cluster node under ideal conditions.

In addition to position, seeks also directly impact the performance of hard drives. Hard drives are optimized for streaming access patterns. Randomly accessed blocks force the drive heads to seek to a new location on disk, incurring latency and degrading the achievable I/O throughput. To quantify the performance impact of seeks, the AIO program (running on a raw disk and bypassing the Hadoop filesystem, OS-provided file cache, and native filesystem) was configured to perform long duration sequential reads and writes, with a seek to a random aligned location every n megabytes. This represents the best-case Hadoop behavior where a large HDFS block of n megabytes is streamed from disk, and then the drive seeks to a different location to retrieve another large block. The outer regions of the drive (identified by low logical addresses) were used to obtain peak bandwidth. As shown in Figure 3.3, the drive performance approaches its peak bandwidth when seeks occur less often than once every 32MB of sequential data accessed. Thus, the HDFS design decision to use large 64MB blocks is quite rea-

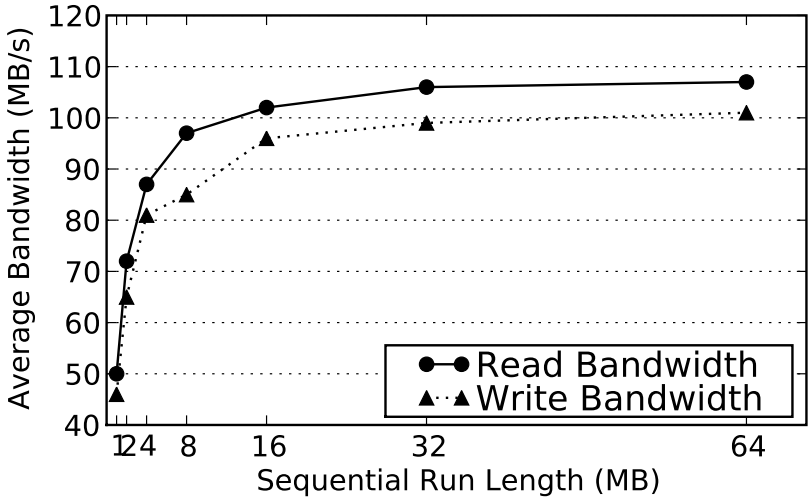


Figure 3.3: Hard Drive Read and Write Bandwidth from AIO Test With Random Seek Every *n* Megabytes

sonable and, assuming that the filesystem maintains file contiguity, should enable high disk bandwidth.

In the following sections, data-intensive application performance is evaluated using the previously described cluster in order to uncover performance bottlenecks in the Hadoop storage architecture.

3.3 Software Architectural Bottlenecks

Software architectural bottlenecks degrade the performance of Hadoop applications by interfering with the desired disk access pattern. Ideally, MapReduce applications should manipulate the disk using streaming access patterns. The application framework should allow for data to be read or written to the disk continuously, and overlap computation with I/O. Many simple applications with low

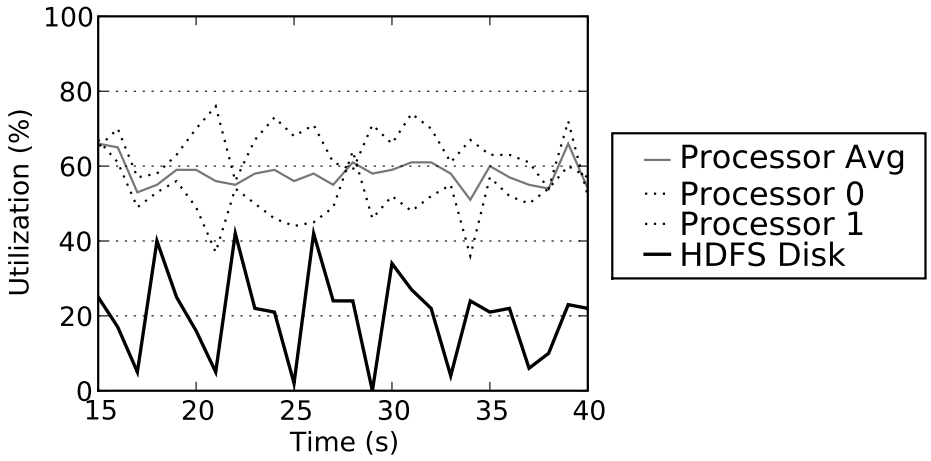


Figure 3.4: Simple Search Processor and Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

computation requirements do not achieve this ideal operating mode. Instead, they utilize the disk in a periodic fashion, decreasing performance.

To determine if Hadoop is using the disk to its full potential, several FreeBSD utilities including *iostat* and *vmstat* were used to profile the system. Data samples were taken every second for the duration of application execution to measure processor and disk utilization. Processor utilization was measured on a per-core basis, and disk utilization was measured as the percentage of time that the disk had at least one I/O request outstanding. As such, this profiling did not measure the relative efficiency of disk accesses (which is done later in this chapter), but simply examined whether or not the disk was kept sufficiently busy with outstanding service requests.

The behavior of the disk and processor utilization over time for the simple search benchmark is shown in Figure 3.4. Here, the system is not accessing the

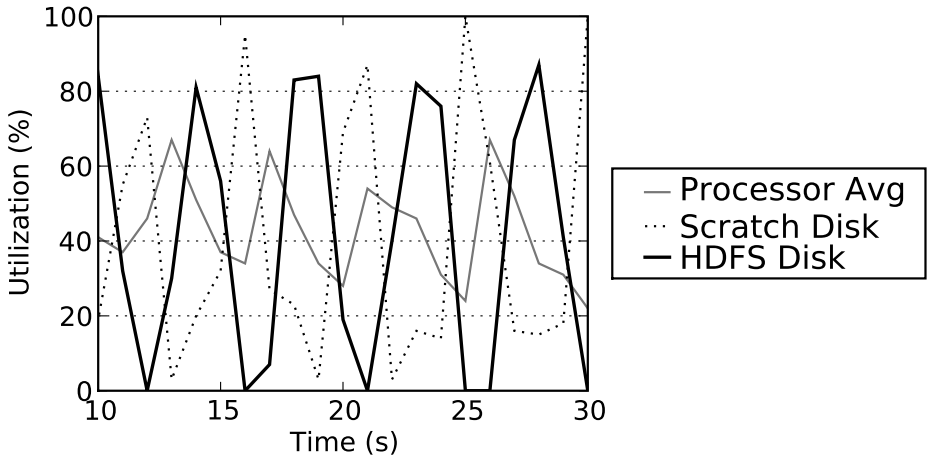


Figure 3.5: Simple Sort Processor and Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

disk in a continuous streaming fashion as desired, even though there are ample processor resources still available. Rather, the system is reading data in bursts, processing it (by searching for a short text string in each input line), and then fetching more data in a periodic manner. This behavior is also evident in other applications such as the sort benchmark, shown in Figure 3.5. Note that the sort benchmark also uses the scratch disk in a periodic fashion to spill temporary key/value pairs that are too large to store in memory.

The overall system impact of this periodic behavior is shown in Figure 3.6, which presents the average HDFS disk and processor utilization for each application in the test suite. The AIO test programs (running as native applications, not in Hadoop) kept the disk saturated with I/O requests nearly all the time (97.5%) with very low processor utilization (under 3.5%). Some Hadoop programs (such as S-Wr and Rnd-Bin) also kept the disk equivalently busy, albeit at much higher

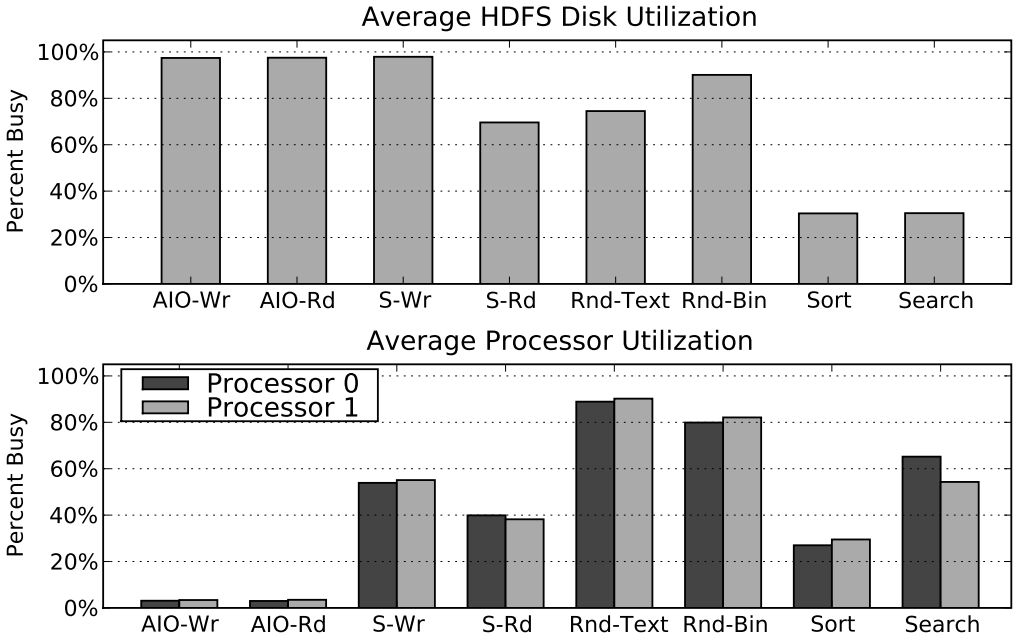


Figure 3.6: Average Processor and HDFS Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

processor usage due to Hadoop and Java virtual machine overheads. In contrast, the remaining programs have poor resource utilization. For instance, the search program accesses the disk less than 40% of the time, and uses the processors less than 60% of the time.

This poor efficiency is a result of the way applications are scheduled in Hadoop, and is not a bottleneck caused by HDFS. By default, the test applications like search and sort were divided into hundreds of map tasks that each process only a single HDFS block or less before exiting. This can speed recovery from node failure (by reducing the amount of work lost) and simplify cluster scheduling. It is easy to take a map task that accesses a single HDFS block and assign it to the node that contains the data. Scheduling becomes more difficult, however, when map tasks

access a region of multiple HDFS blocks, each of which could reside on different nodes. Unfortunately, the benefits of using a large number of small tasks come with a performance price that is particularly high for applications like the search test that complete tasks quickly. When a map task completes, the node can be idle for several seconds until the TaskTracker polls the JobTracker for more tasks. By default, the minimum polling interval is 3 seconds for a small cluster, and increases with cluster size. Then, the JobTracker runs a scheduling algorithm and returns the next task to the TaskTracker. Finally, a new Java virtual machine (JVM) is started, after which the node can resume application processing.

This bottleneck is not caused by the filesystem, but does affect how the filesystem is used. Increasing the HDFS block size to 128MB, 256MB, or higher — a commonly-proposed optimization [64, 67] — indirectly improves performance not because it alleviates any inefficiency in HDFS but because it reduces the frequency at which a node is idle and awaiting scheduling. Another option, over-subscribing the cluster by assigning many more Map and Reduce tasks than there are processors and disks in the cluster nodes, may also mitigate this problem by overlapping computation and I/O from different tasks. But, this technique risks degrading performance in a different manner by increasing I/O contention from multiple clients, a problem discussed further in Section 3.5. More direct methods to attack this performance bottleneck are described and evaluated in Section 4.1.

Even when tasks are available for processing and each task is operating over

large HDFS blocks located on the same node, a bottleneck still exists because the HDFS client implementation is highly serialized for data reads. As discussed in Chapter 2, there is no pipelining to overlap application computation with I/O. The application must wait on the I/O system to contact the NameNode, contact the DataNode, and transfer data before processing. This latency is greater on large clusters with busy NameNodes, or in cases where the data being accessed is not on the same node. Similarly, the I/O system must wait for the application to complete processing before receiving another request. Beyond the lack of pipelining, there is also no data prefetching in the system, despite the fact that MapReduce applications access data in a predictable streaming fashion. Only metadata is prefetched, specifically the mapping between HDFS filename and block IDs. Rather than contact the NameNode each time a new block ID is required, the client caches the next 10 blocks in the file with each read request.

In addition to suffering from software architectural bottlenecks that interfere with efficient storage access, Hadoop applications are also slowed by processor overhead imposed by the HDFS framework.

3.4 Portability Limitations

The Hadoop framework and filesystem impose a significant processor overhead on the cluster. While some of this overhead is inherent in providing necessary functionality, other overhead is incurred due to the design goal of creating a

portable MapReduce implementation. As such, they are referred to here as *Portability Limitations*.

An example of the total overhead incurred is shown in Figure 3.6. The asynchronous I/O write (AIO-Wr) test program — written in C and accessing the raw disk independent of the filesystem — takes less than 10% of the processor during operation. But, the synthetic writer (S-Wr) test program — written in Java and running in Hadoop — takes over 50% of the processor to write data to disk in a similar fashion with equivalent bandwidth. That overhead comes from four places: Java, HDFS implementation, the local filesystem, and the filesystem page cache. While the first two overheads are inherent in the Hadoop implementation, the last two are not.

As discussed in Chapter 2, the Hadoop DataNode uses a local filesystem to store data, and each HDFS block exists as a separate file in the native filesystem. While this method makes Hadoop simple to install and portable, it imposes a computation overhead that is present regardless of the specific filesystem used. The filesystem takes processor time to make data allocation and placement decisions. Similarly, the filesystem page cache consumes both memory resources and processor time to manage cache allocation, deallocation, and copying of data into user buffers due to alignment restrictions. This overhead is not necessary for Hadoop, which already provides its own filesystem (HDFS) and whose streaming access pattern is unlikely to benefit from OS-provided caching.

To quantify the processor resources consumed by the filesystem and cache, a synthetic Java program was used to read and write 10GB files to disk in a streaming fashion using 128kB buffered blocks. This program is similar to the *dd* utility in UNIX, but does not access other file descriptors such as */dev/zero* or */dev/null* that make profile interpretation difficult. The test program incurs file access overheads imposed by Java but not any Hadoop-specific overheads. It was executed both on a raw disk and on a large file in the filesystem in order to compare the overhead of both approaches. The *pmcstat* utility was used to obtain callgraph profiles of the FreeBSD kernel during application execution. The UFS filesystem overhead is computed by examining the gathered kernel profiles and summing the cycles consumed by the *ffs_read()* / *ffs_write()* functions (for the read test and write test, respectively) as well as the cycles consumed by all of their descendents in the callgraph.

As shown in Table 3.3, using a filesystem has a low processor overhead. When reading, 4.4% of the processor time was spent managing filesystem and file cache related functions, and while writing, 7.2% of the processor time was spent on the same kernel tasks. This overhead would be lower if additional or faster processors had been used for the experimental cluster, and higher if additional or faster disks were added to the cluster.

A third class of performance bottlenecks is described next. These are caused by Hadoop's lack of control of the underlying operating system behavior. Instead of

Metric	Read		Write	
	Raw	**Filesystem**	**Raw**	**Filesystem**
Bandwidth (MB/s)	99.9	98.4	98.1	94.9
Processor (total)	7.4%	13.8%	6.0%	15.6%
Processor (FS+cache)	N/A	4.4%	N/A	7.2%

Table 3.3: Processor Overhead of Disk as Raw Device versus Disk with Filesystem and Page Cache (FS+cache)

causing excessive processor utilization, these problems cause excessive disk seeks and on-disk fragmentation.

3.5 Portability Assumptions

A final class of performance bottlenecks exists in the Hadoop filesystem that we refer to as *Portability Assumptions*. Specifically, these bottlenecks exist because the HDFS implementation makes implicit assumptions that the underlying OS and filesystem will behave in an optimal manner for Hadoop. Unfortunately, I/O schedulers can cause excessive seeks under concurrent workloads, and disk allocation algorithms can cause excessive fragmentation, both of which degrade HDFS performance significantly. These agents are outside the direct control of HDFS, which runs inside a Java virtual machine and manages storage as a user-level application.

To identify disk seeks and fragmentation effects at the lowest levels of the system, the *ad_strategy()* function in the FreeBSD kernel was instrumented to gather a block-level I/O trace of all storage requests issued. Several parameters of the I/O request were logged, including the transfer size, access type (read or write),

destination disk to distinguish between the system and HDFS disks, and logical block address (LBA) of the request. Each LBA number in the trace corresponds to a 512 byte sector on disk. Traces are saved in a 64k-entry kernel memory buffer, and then later dumped to disk via a control utility. After profiling a desired application, the traces are analyzed with a Python script to determine the amount of sequential versus random disk access, the run length of all sequential accesses, and the number of I/O operations per second. Sequential seeks are defined as the block strictly following the last request (accounting for its length, which can vary). Random seeks are classified as any seek that is not strictly sequential from the previous access. The run length is simply the total length of all sequential accesses in a series terminating with a random seek.

3.5.1 Scheduling

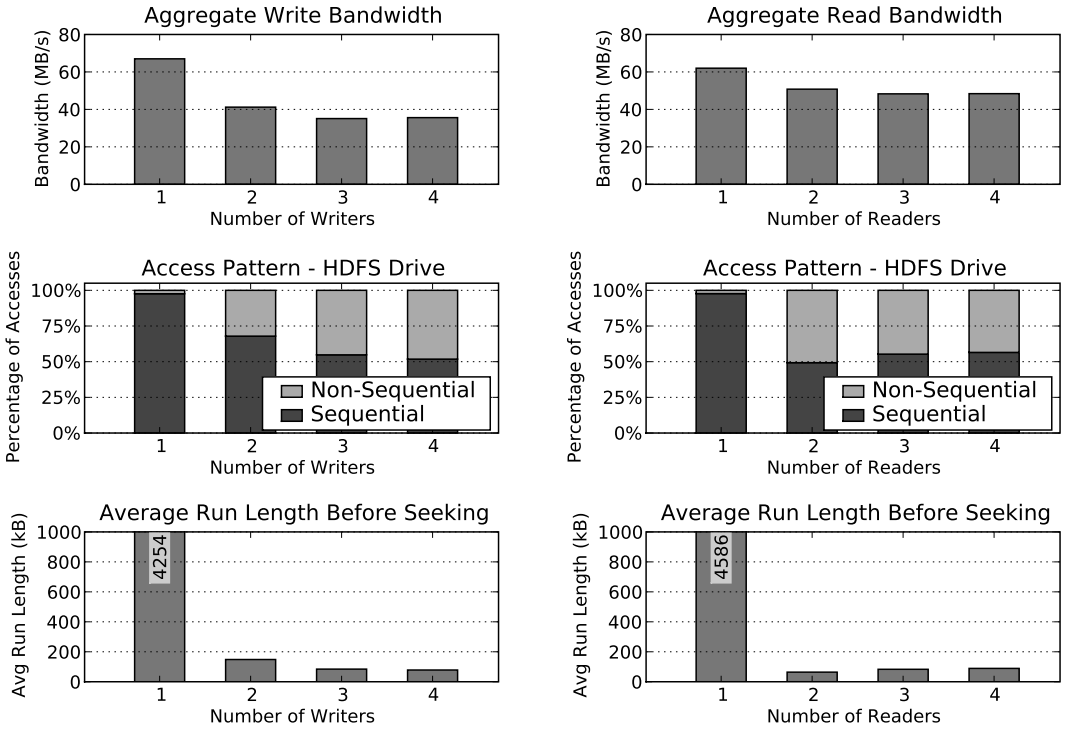
HDFS performance degrades whenever the disk is shared between concurrent writers or readers. Excessive disk seeks occur that are counter-productive to the goal of maximizing overall disk bandwidth. This is a fundamental problem that affects HDFS running on all platforms. Existing I/O schedulers are designed for general-purpose workloads and attempt to share resources fairly between competing processes. In such workloads, storage latency is of equal importance to storage bandwidth; thus, fine-grained fairness is provided at a small granularity (a few hundred kilobytes or less). In contrast, MapReduce applications are almost

entirely latency insensitive, and thus should be scheduled to maximize disk bandwidth by handling requests at a large granularity (dozens of megabytes or more).

To examine the impact of OS disk scheduling, a synthetic test program in Hadoop was used to write 10GB of HDFS data to disk in a sequential streaming manner using 64MB blocks. 1-4 copies of this application were run concurrently on each cluster node. Each instance writes data to a separate HDFS file, thus forcing the system to share limited I/O resources. The aggregate bandwidth achieved by all writers on a node was recorded, as shown in Figure 3.7(a). Aggregate bandwidth dropped by 38% when moving from 1 writer to 2 concurrent writers, and dropped by an additional 9% when a third writer was added.

This performance degradation occurs because the number of seeks increases as the number of writers increases and the disk is forced to move between distinct data streams. Eventually, non-sequential requests account for up to 50% of disk accesses, despite the fact that, at the application level, data is being accessed in a streaming fashion that should facilitate large HDFS-sized block accesses (*e.g.*, 64MB). Because of these seeks, the average sequential run length decreases dramatically as the number of writers increases. What was originally a 4MB average run length decreases to less than 200kB with the addition of a second concurrent writer, and eventually degrades further to approximately 80kB. Such short sequential runs directly impact overall disk I/O bandwidth, as seen in Figure 3.3.

A similar performance issue occurs when HDFS is sharing the disk between



(a) Concurrent Writers (b) Concurrent Readers

Figure 3.7: Impact of Concurrent Synthetic Writers and Readers on HDFS Drive Access Patterns

concurrent readers. To demonstrate this, the same synthetic test program was used. First, a single writer was used per node to write 4 separate 10GB HDFS files. A single writer process creates data that is highly contiguous on disk, as shown by the negligible percentage of seeks in the previous 1-writer test. Then, 1-4 concurrent synthetic reader applications were used per node to each read back a different file from disk.

In this test, the aggregate bandwidth for all readers on a particular node was recorded, as shown in Figure 3.7(b). The aggregate bandwidth dropped by 18% when moving from 1 reader to 2 readers. This is because the number of seeks

increased as the number of readers increased, reaching up to 50% of total disk accesses. This also impacted the average run length before seeking, which dropped from over 4MB to well under 200kB as the number of concurrent readers increased.

By default, the FreeBSD systems used for testing employed a simple elevator I/O scheduler. More sophisticated schedulers are available that aim to minimize seeks, such as the Anticipatory Scheduler. The Anticipatory Scheduler attempts to reduce seeks by waiting a short period after each request to see if further sequential requests are forthcoming [51]. If they are, the requests can be serviced without extra seeks; if not, the disk seeks to service a different client.

To determine the effect of a more sophisticated scheduler on disk seeks, an anticipatory scheduler for FreeBSD was configured and tested using concurrent instances of the Hadoop synthetic writer and reader application. The new scheduler had no impact on the I/O bandwidth of the test programs. Profiling revealed that, for the read workload, the scheduler did improve the access characteristics of the drive. A high degree of sequential accesses (over 95%) and a large sequential run length (over 1.5MB) were maintained when moving from 1 to 4 concurrent readers. But, because the drive was often idle waiting on new read requests from the synchronous HDFS implementation, overall application bandwidth did not improve. Profiling also showed that the scheduler had no impact on the access characteristics of write workloads. This is expected because the filesystem block allocator is making decisions before the I/O scheduler. Thus, even if the anticipatory sched-

uler waits for the next client request, it is often not contiguous in this filesystem and thus not preferred over any other pending requests.

3.5.2 Fragmentation

In addition to poor I/O scheduling, HDFS also suffers from file fragmentation when sharing a disk between multiple writers. The maximum possible file contiguity — the size of an HDFS block — is not preserved by the general-purpose filesystem when making disk allocation decisions.

File fragmentation can be characterized by examining the on-disk metadata associated with each file and retrieving the exact disk placement. For the purposes of this work, however, the precise details of file fragmentation are less important than the overall impact of fragmentation on application-level disk bandwidth and disk seek rates. Both of these metrics can be measured using the infrastructure previously used to characterize concurrent disk accesses.

To measure the impact of file fragmentation on a freshly formatted disk, 1-4 synthetic writer applications were used per node to each create 10GB files, written concurrently. Next, a single synthetic reader application was used to read back one of the 1-4 files initially created. If the data on disk is contiguous, the single reader will be able to access it with a minimum of seeks and maintain high read bandwidth. As fragmentation increases, however, the amount of disk seeks will increase and the application bandwidth will decrease.

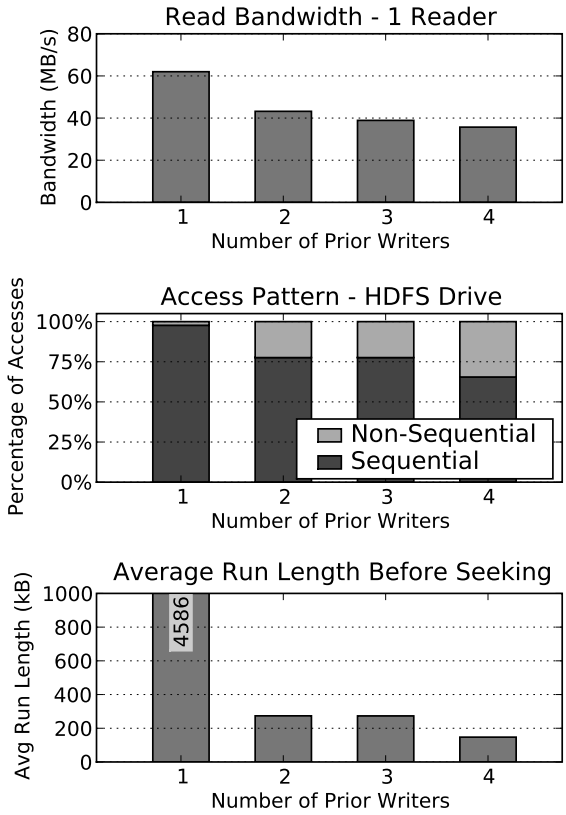


Figure 3.8: One Hadoop Synthetic Reader Program Accessing Data From One Synthetic Writer. (Data was Previously Generated With 1-4 Concurrent Writers)

The results from this experiment are shown in Figure 3.8. Here, file fragmentation occurs whenever multiple writers use the disk concurrently. When the single reader accesses data written when only one writer was active, it receives high bandwidth thanks to a negligible percentage of random seeks, showing that the data was written to the disk in large contiguous blocks. However, when the reader accesses data written when 2 writers were active, read bandwidth drops by 30%. The cause of this drop is an increase in the number of random seeks, and a corresponding decrease in the average sequential run length from over 4MB to ap-

proximately 250kB. This trend continues when 3-4 concurrent writers were used, showing that files suffer from increasing fragmentation as the number of concurrent writers is increased. The level of fragmentation here was produced by using a freshly formatted disk for each experiment. In a Hadoop cluster running for many months or years, the real-world disk fragmentation would likely be greater.

The average run lengths shown in Figure 3.8 for the fragmentation test are almost twice as long as the multiple writers test shown in Figure 3.7(a). This demonstrates that after a disk does a seek to service a different writer, it will sometimes jump back to the previous location to finish writing out a contiguous cluster. Unfortunately, the filesystem used only attempts to maintain small clusters (128kB). As such, the overall level of on-disk file contiguity is still very low compared to what would be optimal for HDFS.

3.6 Discussion

As shown previously, concurrent readers and writers degrade the performance of the Hadoop filesystem. This effect is not a rare occurrence in cluster operation that can be disregarded. Concurrent disk access is found in normal operation because of two key elements: multiple map/reduce processes and data replication.

MapReduce is designed to allow computation tasks to be easily distributed across a large computer cluster. This same parallelization technique also allows the exploitation of multiple processor cores. In the cluster used for experimenta-

tion, each node had 2 processors, and thus was configured to run 2 MapReduce processes concurrently. While 2 processes allowed the test suite to use more computation resources, the concurrent reads and writes created slowed the overall application execution time. Although it might be reasonable in this configuration to either install a second HDFS disk or run only 1 application process per node, this "solution" is not scalable when cluster nodes are constructed with processors containing 4, 8, 16, or more cores. It is unreasonable to either install one disk per core or leave those cores idle — abandoning the parallelization benefits made possible by the MapReduce programming style — to bypass performance problems caused by concurrent disk access. Further, Hadoop installations often deliberately oversubscribe the cluster by running more Map or Reduce tasks than there are processors or disks. This is done in order to reduce system idle time caused by high latency in scheduling and initiating new tasks as identified in Chapter 3.3.

In addition to multiple computation processes, concurrent disk access can also arise due to HDFS data replication. As previously mentioned, clusters typically operate with a replication factor of 3 for redundancy, meaning that one copy of the data is saved locally, one copy is saved on another node in the same rack, and a third copy is saved on a node in a distant rack. But, writing data to disk from both local and remote programs causes concurrent disk accesses.

The effect of a cluster replication factor of 2 on disk access patterns was tested. The results in Table 3.4 show that replication is a trivial way to produce concurrent

Metric	Synthetic Write	Synthetic Read
Sequential %	77.9%	70.3%
Non-Sequential %	22.1%	29.7%
Avg. Sequential Run Length	275.2kB	176.8kB

Table 3.4: Disk Access Characteristics for Synthetic Write and Read Applications with Replication Enabled

access. The behavior of the synthetic writer with replication enabled is highly similar to the behavior of 2 concurrent writers, previously shown in Figure 3.7(a). The mix of sequential and random disk accesses is similar, as is the very small average run length before seeking. Similar observations for the read test can be made against the behavior of 2 concurrent readers, previously shown in Figure 3.7(b). Thus, the performance degradation from concurrent HDFS access is present in every Hadoop cluster using replication. The final section in this chapter shows how these same problems are present in other platforms beyond FreeBSD.

3.7 Other Platforms – Linux and Windows

The primary results shown in this thesis used HDFS on FreeBSD 7.2 with the UFS2 filesystem. For comparison purposes, HDFS was also tested on Linux 2.6.31 using the ext4 and XFS filesystems and Windows 7 using the NTFS filesystem. Here, multiple synthetic writers and readers were used to repeat the same tests described in Section 3.5.1 and Section 3.5.2.

HDFS on Linux suffers from the same type of performance problems as on FreeBSD, although the degree varies by filesystem and test. A summary of test

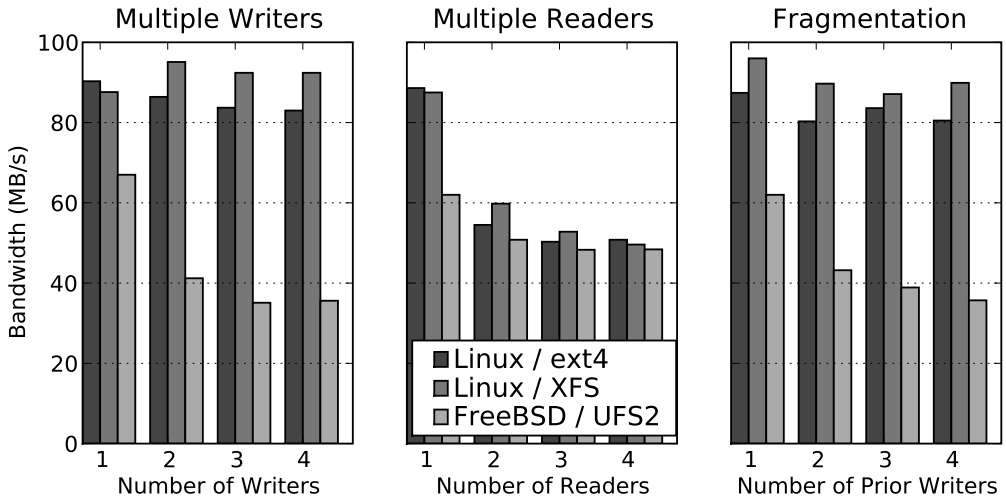


Figure 3.9: Aggregate Bandwidth in Linux with ext4 and XFS Filesystems under Multiple Writer, Multiple Reader, and Fragmentation tests.) FreeBSD results shown for comparison.

results is shown in Figure 3.9 for both the ext4 and XFS filesystems. Previously-reported results for FreeBSD using the UFS2 filesystem are also included for comparison. The most important thing to observe with regards to the raw performance numbers is the higher disk bandwidth in Linux compared to FreeBSD. This is due solely to placement decisions made by the filesystem, as confirmed by instrumenting the operating system. By default, the Linux filesystems start writing at the outer edge of the empty disk, yielding the highest bandwidth from the device as seen in Figure 3.2. In contrast, FreeBSD starts writing at the center of the disk, a region that has lower bandwidth. Both of these placement decisions are reasonable, because as the disk eventually fills with data, the long-term performance average will be identical. Thus, what is important to observe in this filesystem comparison is not the absolute performance, but the change in performance as the number of

multiple writers and readers increases.

Concurrent writes on Linux exhibited better performance characteristics than FreeBSD. For example, the ext4 filesystem showed a 8% degradation moving between 1 and 4 concurrent writers, while the XFS filesystem showed no degradation. This compares to a 47% drop in FreeBSD as originally shown in Figure 3.7(a). In contrast, HDFS on Linux had worse performance for concurrent reads than FreeBSD. The ext4 filesystem degraded by 42% moving from 1 to 4 concurrent readers, and XFS degraded by 43%, compared to 21% on FreeBSD as originally shown in Figure 3.7(b). Finally, fragmentation was reduced on Linux, as the ext4 filesystem degraded by 8% and the XFS filesystem by 6% when a single reader accessed files created by 1 to 4 concurrent writers. This compares to a 42% degradation in FreeBSD, as originally shown in Figure 3.8.

Hadoop in Windows 7 relies on the Cygwin Unix emulation layer to function. Disk write bandwidth was acceptable (approximately 60MB/s), but read bandwidth was very low (under 10MB/s) despite high disk utilization exceeding 90%. Although the cause of this performance degradation was not investigated closely, it is consistent with small disk I/O requests (2-4kB) instead of large requests (64kB and up). Because Hadoop has only received limited testing in Windows, this configuration is supported only for application development, and not for production uses [14]. All large-scale deployments of Hadoop in industry use Unix-like operating systems such as FreeBSD or Linux, which are the focus of this thesis.

CHAPTER 4

Optimizing Local Storage Performance

As characterized in Chapter 3, the portable implementation of Hadoop suffers from a number of bottlenecks in the software stack that degrade the effective bandwidth of the HDFS storage system. These problems include:

Task Scheduling and Startup — Hadoop applications with large numbers of small tasks (such as the search and sort benchmarks) suffer from poor overall disk utilization, as seen in Section 3.3. This is due to delays in notifying the JobTracker of the previous task completion event, receiving a new task, and starting a new JVM instance to execute that task. During this period, disks sit idle, wasting storage bandwidth.

Disk scheduling — The performance of concurrent readers and writers suffers from poor disk scheduling, as seen in Section 3.5.1. Although HDFS clients access massive files in a streaming fashion, the framework divides each file into multiple HDFS blocks (typically 64MB) and smaller packets (64kB). The request stream actually presented to the disk is interleaved between concurrent clients at this small granularity, forcing excessive seeks and degrading bandwidth, and negating one

of the key potential benefits that a large 64MB block size would have in optimizing concurrent disk accesses.

Filesystem allocation — In addition to poor I/O scheduling, HDFS also suffers from file fragmentation when sharing a disk between multiple writers. As discussed in Section 3.5.2, the maximum possible file contiguity — the size of an HDFS block — is not preserved by the general-purpose filesystem when disk allocation decisions are made.

Filesystem page cache overhead — Managing a filesystem page cache imposes a computation and memory overhead on the host system, as discussed in Section 3.4. This overhead is unnecessary because the streaming access patterns of MapReduce applications have minimal locality that can be exploited by a cache. Further, even if a particular application did benefit from a cache, the page cache stores data at the wrong granularity (4-16kB pages vs 64MB HDFS blocks), thus requiring extra work to allocate memory and manage metadata.

To improve the performance of HDFS, there are a variety of architectural improvements that could be used. In this section, portable solutions are first discussed, followed by non-portable solutions that could enhance performance further at the expense of compromising a key HDFS design goal.

4.1 Task Scheduling and Startup

There are several methods available to reduce the delays inherent in issuing and starting new tasks in the Hadoop framework, and their impact on application performance is evaluated here. These include decreasing the heartbeat interval at which the JobTracker is contacted, re-using the JVM for multiple tasks, and processing more than a single HDFS block with each task. All these changes are portable and would function effectively across all Hadoop host platforms.

Fast Heartbeat — Each TaskTracker periodically contacts the JobTracker with a heartbeat message to report its current status and any recently completed tasks, and request new tasks if work is available. By default, the polling interval is statically set by the JobTracker as either 3 seconds, or 1 second per 100 nodes in the cluster, whichever is larger. This allows the per-node heartbeat interval to increase on large clusters in an attempt to prevent the JobTracker from being swamped with too many messages. To examine the relationship between heartbeat interval and application performance, the interval was decreased to a fixed 0.3 seconds. This decreased task scheduling latency at the cost of increasing JobTracker processor load. For the small cluster size used in these experiments, there was no appreciable increase in JobTracker resource utilization.

JVM Reuse — By default, Hadoop starts a new Java Virtual Machine (JVM) instance for each task executed by the TaskTracker. This provides several benefits in terms of implementation convenience. With separate JVMs, it is easier to at-

tach log files to the standard output and error streams and prevent spurious writes from subsequent tasks. Further, separate JVMs provide stronger memory isolation between subsequent tasks. It is easy to guarantee that a task will have a full complement of memory available to it if the JVM used for the previous task has been killed and re-launched. It is harder to ensure that all memory from a potentially misbehaved task has been completely freed. Although this default choice simplified the implementation of the Hadoop framework, it incurs processor overhead with every new task and consequently delays application execution. Here, the configuration of Hadoop is modified to start a new JVM instance for every job, where a job can consist of hundreds or thousands of individual tasks per node. For the well-behaved applications used in the test suite, this change caused no reliability problems.

Large Tasks — When splitting a large input file into pieces to be processed by individual compute node, Hadoop by default splits the file into HDFS block-sized chunks (64MB), each of which is processed by an independent map task. Thus, it is common to run thousands of tasks to accomplish a single job. Here, that default is modified to assign up to 5GB of input data to a single task, thereby reducing the number of tasks and the amortizing the latency inherent in issuing each task across a larger amount of productive work.

The individual contribution of each of these changes is shown in Figure 4.1 for the search benchmark, along with the default and combined performance. In this

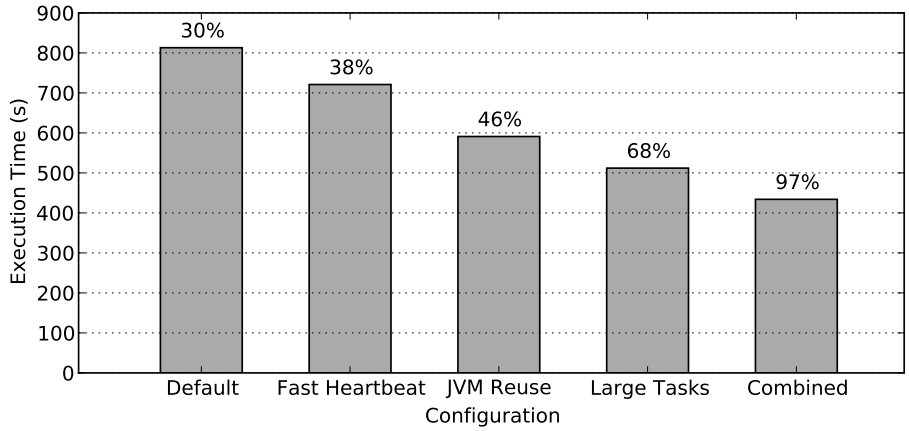


Figure 4.1: Task Tuning For Simplesearch Benchmark. Bar Label is HDFS Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

figure, the percent labels on top of each bar represent the HDFS disk utilization, or the percent of time that the HDFS disk had at least 1 request outstanding.

As shown in the figure, adjusting the polling interval for new tasks increased search performance by 11%, although disk utilization was still only 38%. Re-using the JVM between map tasks increased search performance further, yielding a 27% improvement over the default configuration. Making each map task process 5GB of data instead of 64MB before exiting improved search performance by 37% and boosted disk utilization to over 68%. Finally, combining all three changes improved performance by 46% and increased HDFS disk utilization to 97%.

The cumulative impact of these optimizations is shown for the simple search benchmark in Figure 4.2. Here, the disk and processor utilization over time are monitored. The behavior of the search benchmark compares favorably against the unoptimized original behavior shown in Figure 3.4. Previously, the HDFS disk was

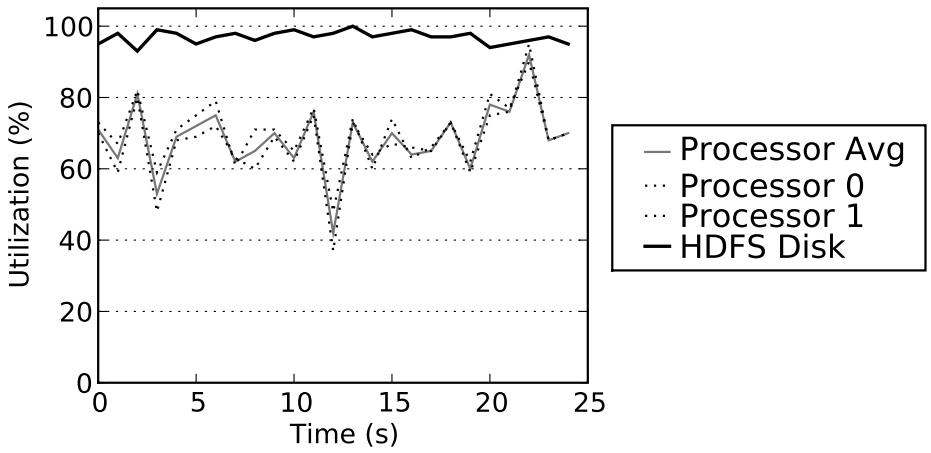


Figure 4.2: Optimized Simple Search Processor and Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

used in a periodic manner with frequent periods of idle time. Now, the HDFS disk is used in an efficient streaming manner with near 100% utilization. The average processor overhead is higher, as expected, due to the much higher disk bandwidth being managed.

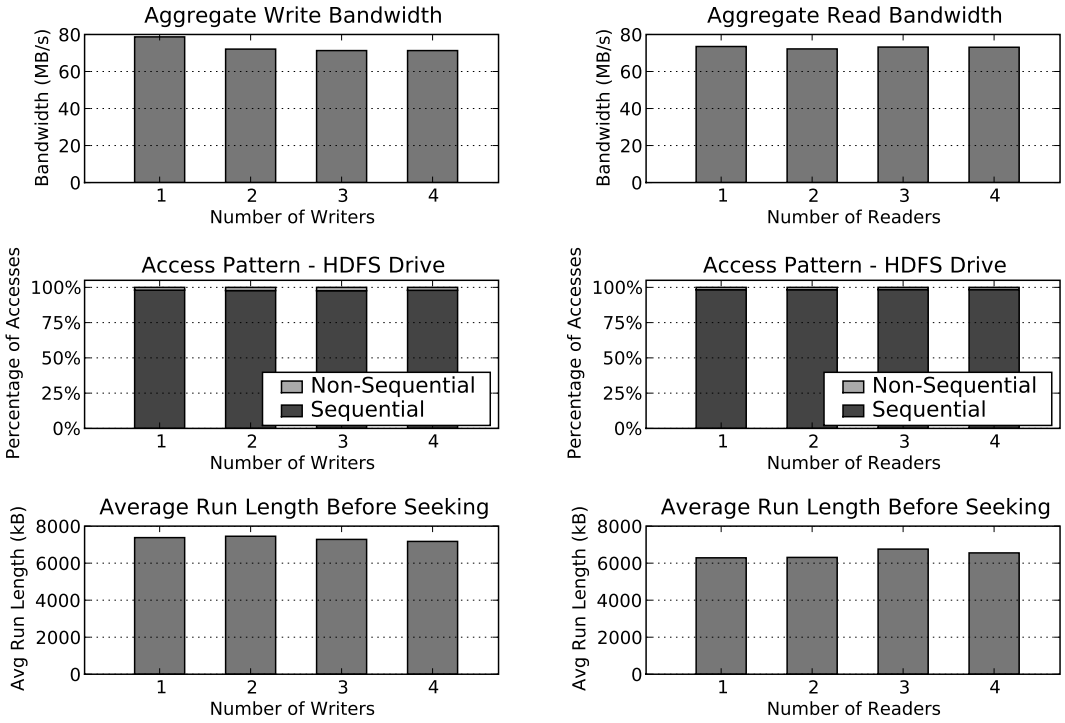
These specific changes to improve Hadoop task scheduling and startup impose tradeoffs, and may not be well suited to all clusters and applications. Many other design options exist, however, to eliminate the bottlenecks identified here. For example, increasing the heartbeat rate increases the JobTracker processor load, and will limit the ultimate scalability of the cluster. Currently, Hadoop increases the heartbeat interval as cluster size increases according to a fixed, conservative formula. The framework could be modified, however, to set the heartbeat dynamically based on the current JobTracker load, thus allowing for a faster heartbeat rate to be opportunistically used without fear of saturating the JobTracker node on a

continuous basis. As another example, re-using JVM instances may imposes long-term reliability problems. The Hadoop framework could be modified, however, to launch new JVM instances in parallel with requesting new task assignments, instead of serializing the process as in the current implementation. Finally, as a long-term solution, if task scheduling latency still imposes a performance bottleneck in Hadoop, techniques to pre-fetch tasks in advance of when there are needed should be investigated. The combined performance improvements shown in this section can be considered the best-case gains for any other architectural changes made to accelerate Hadoop task scheduling.

Improving Hadoop task scheduling and startup can improve disk utilization, allowing storage resources to be used continuously and intensely. Next, disk-level scheduling is optimized in order to ensure that the disk is being used efficiently, without excessive fragmentation and unnecessary seeks.

4.2 HDFS-Level Disk Scheduling

A portable way to improve disk scheduling and filesystem allocation is to modify the way HDFS batches and presents storage requests to the operating system. In the existing Hadoop implementation, clients open a new socket to the DataNode to access data at the HDFS block level. The DataNode spawns one thread per client to manage both the disk access and network communication. All active threads access the disk concurrently. In a new Hadoop implementation using



(a) Concurrent Writers (b) Concurrent Readers

Figure 4.3: Impact of HDFS-Level Disk Scheduling on Concurrent Synthetic Writers and Readers

HDFS-level disk scheduling, the HDFS DataNode was altered to use two groups of threads: a set to handle per-client communication, and a set to handle per-disk file access. Client threads communicate with clients and queue outstanding disk requests. Disk threads — each responsible for a single disk — choose a storage request for a particular disk from the queue. Each disk management thread has the ability to interleave requests from different clients at whatever granularity is necessary to achieve full disk bandwidth — for example, 32MB or above as shown in Figure 3.3. In the new configuration, requests are explicitly interleaved at the granularity of a 64MB HDFS block. From the perspective of the OS, the disk is

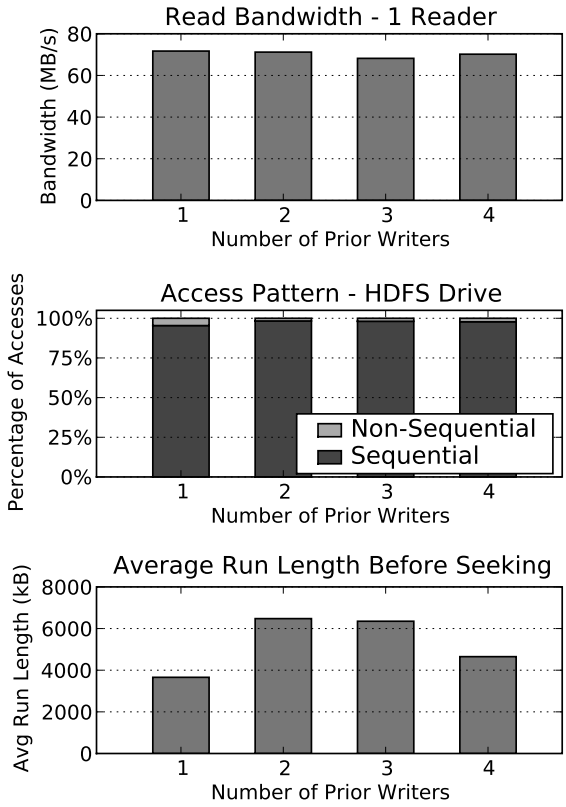


Figure 4.4: Impact of HDFS-Level Disk Scheduling on Data Fragmentation

accessed by a single client, circumventing any OS-level scheduling problems. The previous tests were repeated to examine performance under multiple writers and readers. The results are shown in Figure 4.3(a) and Figure 4.3(b).

Compared to the previous concurrent writer results in Figure 3.7(a), the improved results shown in Figure 4.3(a) are striking. What was previously a 38% performance drop when moving between 1 and 2 writers is now a 8% decrease. Random seeks have been almost completely eliminated, and the disk is now consistently accessed in sequential runs of greater than 6MB. Concurrent readers also show a similar improvement when compared against the previous results in Fig-

ure 3.7(b). In addition to improving performance under concurrent workloads, HDFS-level disk scheduling also significantly decreased the amount of data fragmentation created. Recall that, as shown in Figure 3.8, files created with 2 concurrent writers were split into fragments of under 300kB. However, when re-testing the same experiment with the modified DataNode, the fragmentation size exceeded 4MB, thus enabling much higher disk bandwidth as shown in Figure 4.4.

HDFS-level scheduling also has performance benefits in operating systems other than FreeBSD. Recall from Figure 3.9 that in Linux using the ext4 filesystem, HDFS performance degraded by 42% moving from 1 to 4 concurrent readers. Running the same synthetic writer and reader experiments with HDFS-level scheduling enabled greatly improved performance, as shown in Figure 4.5. In all three test scenarios — multiple writers, multiple readers, and fragmentation — HDFS throughput degraded by less than 3% when moving between 1 and 4 concurrent clients.

Although this portable improvement to the HDFS architecture improved performance significantly, it did not completely close the performance gap. Although the ideal sequential run length is in excess of 32MB, this change only achieved run length of approximately 6-8MB, despite presenting requests in much larger 64MB groups to the operating system for service. To close this gap completely, non-portable techniques are needed to allocate large files with greater contiguity and less metadata.

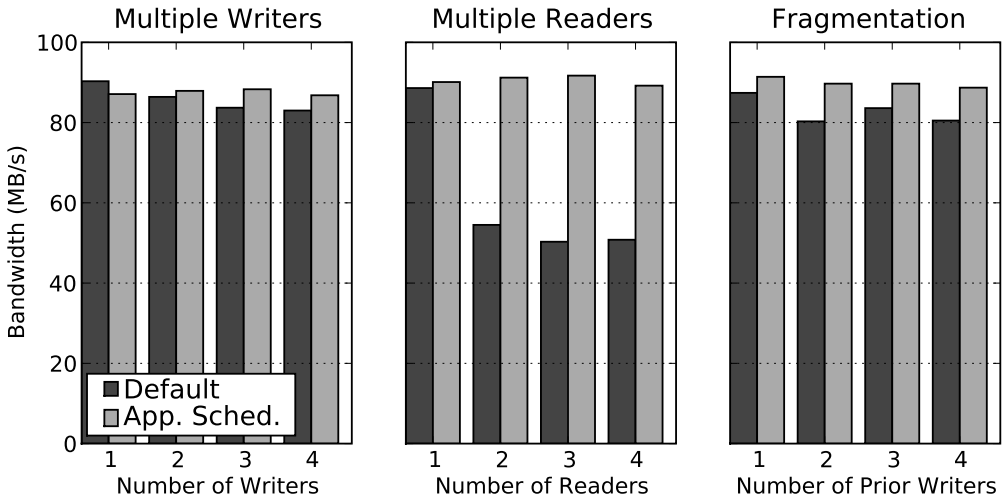


Figure 4.5: Impact of HDFS-Level Disk Scheduling on Linux ext4 Filesystem

4.3 Non-Portable Optimizations

Some performance bottlenecks in HDFS, including file fragmentation and cache overhead, are difficult to eliminate via portable means. A number of non-portable optimizations can be used if additional performance is desired, such as delivering usage hints to the operating system, selecting a specific filesystem for best performance, bypassing the filesystem page cache, or removing the filesystem altogether.

OS Hints — Operating-system specific system calls can be used to reduce disk fragmentation and cache overhead by allowing the application to provide "hints" to the underlying system. Some filesystems allow files to be pre-allocated on disk without writing all the data immediately. By allocating storage in a single operation instead of many small operations, file contiguity can be greatly improved. As an example, the DataNode could use the Linux-only *fallocate()* system call in con-

junction with the ext4 or XFS filesystems to pre-allocate space for an entire HDFS block when it is initially created, and later fill the empty region with application data. In addition, some operating systems allow applications to indicate that certain pages will not be reused from the disk cache. Thus, the DataNode could also use the *posix_fadvise* system call to provide hints to the operating system that data accessed will not be re-used, and hence caching should be a low priority. The third-party *jposix* Java library could be used to enable this functionality in Hadoop, but only for specific platforms such as Linux 2.6 / AMD64.

Filesystem Selection — Hadoop deployments could mandate that HDFS be used only with local filesystems that provide the desired allocation properties. For example, filesystems such as XFS, ext4, and others support *extents* of varying sizes to reduce file fragmentation and improve handling of large files. Although HDFS is written in a portable manner, if the underlying filesystem behaves in such a fashion, performance could be significantly enhanced. Similarly, using a poor local filesystem will degrade HDFS.

Cache Bypass — In Linux and FreeBSD, the filesystem page cache can be bypassed by opening a file with the *O_DIRECT* flag. File data will be directly transferred via direct memory access (DMA) between the disk and the user-space buffer specified. This will bypass the cache for file data (but not filesystem metadata), thus eliminating the processor overhead spent allocating, locking, and deallocating pages. While this can improve performance in HDFS, the implementation is

non-portable. Using DMA transfers to user-space requires that the application buffer is aligned to the device block size (typically 512 bytes), and such support is not provided by the Java Virtual Machine. The Java Native Interface (JNI) could be used to implement this functionality as a small native routine (written in C or C++) that opens files using O_DIRECT. The native code must manage memory allocation (for alignment purposes) and deallocation later, as Java's native garbage collection features do not extend to code invoked by the JNI. Implementing this in the DataNode architecture might be challenging, but it would only need to be implemented once, and then all Hadoop applications would benefit from the improved framework performance.

Local Filesystem Elimination — To maximize system performance, the HDFS DataNode could bypass the OS filesystem entirely and directly manage file allocation on a raw disk or partition, in essence replacing the kernel-provided filesystem with a custom application-level filesystem. This is similar to the idea of a user-space filesystem previously discussed in Section 2.4.3. A custom filesystem could reduce disk fragmentation and management overhead by allocating space at a larger granularity (e.g. at the size of an HDFS block), allowing the disk to operate in a more efficient manner as shown in Figure 3.3.

To quantify the best-case improvement possible with this technique, assume an idealized on-disk filesystem where only 1 disk seek is needed to retrieve each HDFS block. Because of the large HDFS block sizes, the amount of metadata

needed is low and could be cached in DRAM. In such a system, the average run length before seeking should be 64MB, compared with the 6MB runs obtained with HDFS-level scheduling on a conventional filesystem (See Figure 4.3). On the test platform using a synthetic disk utility, increasing the run length from 6MB to 64MB improves read bandwidth by 16MB/s and write bandwidth by 18MB/s, a 19% and 23% improvement, respectively. Using a less optimistic estimate of the custom application-level filesystem efficiency, even increasing the run length from 6MB to 16MB will improve read bandwidth by 14 MB/s and write bandwidth by 15 MB/s, a 13% and 19% improvement, respectively.

One way to achieve a similar performance gain while still keeping a traditional filesystem is to add a small amount of non-volatile flash storage to the system, and partition the filesystem such that the flash memory is used to store metadata and the spinning disk is reserved solely for large, contiguous HDFS blocks. This idea has been explored by Wang *et al.* who made the observation that, of all the possible data that would benefit from being saved in faster memory than a spinning disk, metadata would benefit the most [80]. To that end, they proposed a system called *Conquest* that improved storage performance by separating the filesystem metadata from the actual data and storing both on separate devices. In their system, metadata (and small data files) were stored solely on battery-backed memory, while the data portions of large files remained stored on disk. Their work shared some similarities with the preceding HeRMES architecture that coupled a

disk with a magnetic RAM, which, like flash memory, is non-volatile [59]. Both designs were created for general-purpose computing with a mix of small and large files (the same file mix for traditional filesystems), and as such can be optimized for DC-style data storage. Further, the memory technologies used by both examples have different usage requirements than modern flash memory. For example, storing metadata in flash memory instead of battery-back RAM might require a different design (such as a log structure), due to the block-erasure requirement of flash memory that makes in-place writes very slow compared to random reads.

4.4 Conclusions

In the previous chapter, the interactions between Hadoop and storage were characterized in detail. The performance impact of HDFS is often hidden from the Hadoop user. While Hadoop provides built-in functionality to profile Map and Reduce task execution, there are no built-in tools to profile the framework itself, allowing performance bottlenecks to remain hidden. User-space monitoring tools along with custom kernel instrumentation were used to gain insights into the black-box operation of the HDFS engine.

Although user applications or the MapReduce programming model are typically blamed for poor performance, the results presented showed that the Hadoop framework itself can degrade performance significantly. Hadoop is unable in many scenarios to provide full disk bandwidth to applications. This can be caused

by delays in task scheduling and startup, or fragmentation and excessive disk seeks caused by disk contention under concurrent workloads. The achieved performance depends heavily on the underlying operating system, and the algorithms employed by the disk scheduler and allocator.

In this section, techniques to improve Hadoop performance using the traditional local storage architecture were evaluated. HDFS scheduler performance can be significantly improved by increasing the heartbeat rate, enabling JVM reuse, and using larger tasks to amortize any remaining overhead. Although these specific techniques may involve tradeoffs depending on cluster size and application behavior, the performance gains show the benefits possible with improved scheduling, and motivate future work in this area. Further, HDFS performance under concurrent workloads can be significantly improved through the use of HDFS-level I/O scheduling while preserving portability. Additional improvements by reducing fragmentation and cache overhead are also possible, at the expense of reducing portability. All of these architectural improvements boost application performance by improving node efficiency, thereby allowing more computation to be accomplished with the same hardware.

CHAPTER 5

Storage Across a Network

The field of enterprise-scale storage has a rich history, both in terms of research and commercial projects. Remote storage architectures have been created for a variety of network configurations, including across a wide-area network (WAN) with high latency links, across a local-area network (LAN) shared with application data, and across a storage-area network (SAN) used solely for storage purposes. Further, previous research has introduced several models for network-attached disks without the overhead of a traditional network file server. These architectures share a common element in that clients access storage resources across a network, and not from directly attached disks, as done by Hadoop in its traditional local-storage design. Here, a number of existing network storage architectures are described and compared to the proposed remote-storage HDFS design presented in this thesis. In addition, existing data replication and load balancing strategies are described and related to the methods used by HDFS. These ensure reliability and high performance by exploiting the flexibility offered by network-based storage.

5.1 Wide Area Network

For clients accessing storage across a WAN such as the Internet, a variety of solutions have been developed. These systems – often referred to as *storage clouds* – can be divided into two categories: datacenter-oriented and Internet-oriented. Datacenter oriented solutions are exemplified by systems such as Sector [46] and Amazon's Simple Storage Service (S3) [66]. They are typically administered by a single entity and employ a collection of disks co-located in a small number of datacenters interconnected by high-bandwidth links. Clients access data in the storage cloud using unique identifiers that refer to files or blocks within a file, and are not aware of the physical location of the data inside the datacenter. To a client, the storage cloud is simply one large disk. Storage clouds and HDFS are similar in that both present an abstraction of a huge disk, and both use unique identifiers to access blocks within a file. But, in both the traditional local Hadoop architecture, and in the proposed remote-storage architecture, HDFS clients are aware of the location of data in the datacenter, and must contact the specific DataNode in order to retrieve it.

In contrast to this datacenter-driven approach, Internet-based distributed peer-to-peer storage solutions have also been developed. Examples of this architecture include OceanStore [55], the Cooperative File System (CFS) [33], and PAST [73]. In these systems, a collection of servers collaborate to store data. These servers are not co-located in a datacenter, but are instead randomly distributed across the Internet.

Client nodes choose storage nodes via distributed protocols such as hashing file identifiers, and are typically fully exposed to storage architecture concerns such as the physical location of the data being accessed. Implementations of these peer-to-peer storage systems differ in details, such as whether the storage servers are trusted [33] or untrusted [55],whether access is provided at the block level [33] or file level [73], and whether erasure coding [55] or duplication is used to provide data replication. Some peer-to-peer file systems have similarities with the global file system provided by the Hadoop framework. For example, CFS provides a read-only file system [33], while PAST provides "immutable" files [73]. Both of these have similar access semantics to the write-once, read-many architecture of HDFS.

To reduce the performance impact of accessing storage resources via a high latency WAN, a number of different techniques have been developed. These include employing parallel TCP streams between client and server [20], performing disk access and network I/O in parallel instead of sequentially on the storage server [24], and employing asynchronous I/O operations on clients to decouple computation and I/O access [21]. These techniques are valuable for Hadoop, even when accessing data across a low-latency network. They are partially but not consistently implemented in the existing HDFS framework. In addition to these optimizations for WAN access, aggressive client-side caching is often applied to frequently accessed files to entirely bypass the network access. Caching effective-

ness is dependent on data reuse frequency and the size of the working set. In data-intensive computing applications, however, files are accessed in a streaming fashion and either not reused at all, or reused only with huge working sets. Thus, client-side caching is not traditionally employed in file systems for MapReduce clusters [42], and it is not proposed for the remote storage architecture either.

In the case of the traditional Hadoop architecture or the new proposed remote architecture, all storage resources are co-located within the confines of a single datacenter, or even within a few racks in the same datacenter. Thus, it is similar to other related work that focuses on storage interconnected by a low-latency, high-bandwidth enterprise network.

5.2 Local Area Network

In contrast to storage clouds operating across wide area networks such as the Internet, other storage architectures have been developed to operate across a low-latency datacenter LAN shared with application-level traffic. Lee *et al.* developed a distributed storage system called Petal that uses a collection of disk arrays to collectively provide large block-level virtual disks to client machines via an RPC protocol [56]. Because co-locating disks across a low latency LAN allows for tighter coupling between storage servers and performance that is less sensitive to network congestion or packet loss, Petal is able to hide the physical layout of the storage system from clients and simply present a virtual disk interface. A simple master-

slave replication protocol is used to distribute data for redundancy and to allow clients to load-balance read requests, although the protocol is vulnerable to network partitioning. In contrast to Petal, Hadoop exploits the low-latency network to simplify the storage system implementation. A centralized NameNode service is used for convenience. This service must be queried for every file request to obtain a mapping between file name and the blocks (and storage locations) making up that file, and its response time directly impacts file access latency.

Storage architectures in the datacenter do not need to rely on traditional server-class machines with high-power processors and many disks per chassis. Saito *et al.* proposed a system that uses commodity processors, disks, and Ethernet networks tied together with software to provide a storage service [75]. Although the philosophy of using commodity parts is similar to Hadoop and other MapReduce frameworks, this architecture does not co-locate storage with computation. Storage is an independent service. In this system, a large numbers of small storage "bricks" (nodes containing a commodity processor, disk, NVRAM, and a network interface) running identical control software are combined in a single datacenter to form a "Federated Array of Bricks". The control software is responsible for presenting a common storage abstraction (such as iSCSI) to clients. To access the array, clients pick a brick at random to communicate with. That brick is responsible for servicing all requests received, but will often have to proxy data that is not stored locally. An erasure coding system using voting by bricks is employed so that the

system can tolerate failed bricks, overloaded bricks, or network partitioning. This erasure-coding algorithm was redesigned in a subsequent work to be fully decentralized [41]. Although the control software was designed for use in a datacenter, the concept of a small storage "brick" would work equally well across a WAN.

The "brick" architecture described comes close, in many ways, to the proposed design for remote storage in Hadoop that will be described fully in Chapter 7. It shares a common vision for decoupling storage and computation resources in the cluster, and using lightweight storage nodes to present a common abstraction to the clients of a unified pool of storage. Where it differs is in terms of software architecture. DataNodes in Hadoop do not proxy data on behalf of clients - the client must contact the desired DataNode direct and request blocks. (In Hadoop, DataNodes do proxy data for client write requests, but only as part of the replication process). Further, there is no erasure coding or voting in the Hadoop architecture. Fault tolerance is provided by full data replication as directed by the NameNode, a centralized master controller.

The final traditional type of remote storage architecture, like the local-area network designs described previously, functions over a low-latency network. Unlike before, however, this network is dedicated to storage traffic only, enabling the use of proprietary designs optimized specifically for storage workloads.

5.3 Storage Area Network

Storage systems for cluster computers are commonly implemented across dedicated storage area networks. These storage area networks traditionally use proprietary interconnect technology such as Fibre Channel, or special protocols such as iSCSI over more conventional IP networks. Regardless, in a storage area network, disks are accessed via a dedicated network that is isolated from application-level traffic. This means that compute nodes must either have an additional network interface to communicate with the storage network, or gateway servers must be utilized to translate between the storage network (using storage protocols) and the application network (using standard network file system protocols). MapReduce clusters are the only modern example of a large-scale computing system that does not employ network-based storage, and instead tightly couples computation and storage.

As distinguished from these conventional approaches, a non-traditional storage area network architecture was proposed by Hospodor *et al* [49]. In this system, a petabyte-scale storage system is built from a collection of storage nodes. Each node is a network-accessible disk exporting an object-based file system, and is joined with a 4-12 port gigabit Ethernet switch. By adding a switch to the existing smart disk architecture, many different network topologies can be realized with a variety of cost/performance tradeoffs. This network is dedicated for storage traffic only, and was not designed to be shared with application data. Further research

on this system focused on improving system reliability in the case of failure [39].

Storage area networks have been rejected for use in traditional MapReduce clusters (using a local storage architecture) due to their reliance on expensive, proprietary technologies. By eliminating the expensive SAN entirely, clusters built on a framework such as Hadoop decrease administrative overhead inherent in managing two separate networks, and also achieve a much lower per-node installation cost. The number of NICs, cables, and switches have all been reduced, lowering costs for installation, management, power, and cooling. MapReduce clusters can be constructed entirely out of commodity processors, disks, network cards, and switches that are available at the lowest per-unit cost. Thus, a larger number of compute nodes can be provisioned for the same cost as the architecture built around a SAN. The same logic holds true for the remote storage architecture proposed here for Hadoop, which also rejects the use of a dedicated SAN. Although a remote architecture necessitates more network ports, both storage and cluster traffic are designed to run across the same network.

In the field of network-based storage, regardless of the exact network topology used (*i.e.*, WAN, LAN, or SAN), an ongoing question is: what is the desired division of work between compute resources and storage resources? Should storage nodes be lightweight, or is there value in giving them more processing power?

5.4 Network Disks and Smart Disks

Disks do not have to be placed in conventional file servers in order to be accessed across a network. A number of "network disk" or "smart disk" architectures to transform disks into independent entities have been previously described, with many variations. These fall into two main categories: adding a network interface to a remote disk for direct access, and adding a processor to a locally-attached disk to offload application computation. Some proposals combine elements of both approaches to add processing and network capabilities to disks that are located remotely.

In the category of networked disks, Gibson *et al.* proposed directly attaching storage (disks) to the network through the use of the embedded disk controller. They referred to such devices as "network attached storage devices" or "network attached secure disks", depending on whether the emphasis was on storage or security. This architecture supports direct device to client transfers, without the use of a network server functioning as an intermediary (as in a traditional storage-area network architecture) [45, 44]. This work built upon previous research by Anderson *et al.* who proposed one of the first examples of a serverless network file system [23]. In such a system, any client can access block storage devices across the network at any time without needing to communicate with a centralized controller first. All the clients communicate as peers. This lightweight storage device would make an ideal platform for remote storage in a Hadoop cluster, provided that the

network attached storage devices could be manufactured cheaply enough to be cost competitive with simply placing disks in a commodity server. Regardless, it serves as an example of how lightweight systems can still effectively provide network storage resources.

In the category of smart disks, a number of designs have proposed making disks intelligent ("active") to process large data sets [19, 72, 54, 31, 68]. In these architectures, disks are outfitted with processing and memory resources and a programming model is used to offload application-specific computation from the general-purpose client nodes. This is similar to the current DC concept of moving computation to the data, but instead of putting disks in the compute nodes, it places compute nodes (in some embedded form with limited capabilities) in the disk itself. In these architectures, there are often two layers of computation: computation performed at the disk (in a batch-processing manner), and computation performed at dedicated compute nodes (in a general-purpose manner). Computation is done at the disk to reduce the amount of data that must be moved to the compute nodes, thus reducing network bottlenecks in the cluster.

Smart disks do not have to be restricted to only using general-purpose processors. Netezza is an example of a commercial smart-disk product that uses FPGAs to filter data. In this architecture, an FPGA, processor, memory, and gigabit Ethernet NIC are co-located with a disk [34]. The FPGA servers as a disk controller, but also allows queries (filters) to be programmed into it. Data is streamed off the disk

and through the FPGAs, and any data that satisfies the queries is then directed to the attached processor and memory for further processing. After processing in the local unit, data can be sent across the Ethernet network to clients.

As an example of combining both network disk and smart disk features, several architectures have proposed exploiting the computation power of networked smart disk to provide an object-based interface to storage instead of a block-based interface [60, 40]. These "object-based storage device" (OSD) systems can be thought of as another form of "active disk", where disk computation resources are used for application purposes. In this case, the disk is now responsible for managing data layout. This provides opportunities for tighter coupling with software stack, as many parallel file systems already represent data as objects. Such opportunities also exist in HDFS regardless of whether it is accessed locally or remotely. The DataNode service exports data at the HDFS block level, which is independent of the physical arrangement of data on the disk. Thus, several disks can be managed by a single DataNode as a single storage unit.

5.5 Data Replication

For data-intensive computing applications, the reliability of the storage system is of high importance. Data written to the storage system is commonly replicated across multiple disks to decrease the probability of data loss and enable load balancing techniques for read requests (discussed in the subsequent section). There

are many methods that can be used to determine where replicated data should be written to, for both wide-area networks and local-area networks.

For storage systems spanning wide-area networks, data can be written to random nodes to ensure an even distribution of storage traffic. This is the method employed by CFS, a peer-to-peer, read-only file system. In CFS, blocks are placed on random servers in the network, without regard to performance concerns. Such servers are adjacent in terms of a distributed hash ring for implementation convenience, but this translates to random nodes in terms of physical location. The first storage server is responsible for ensuring that sufficient active replicas are maintained at all times [33]. In addition to random placement, replicas can be placed so that overall latency from client to storage nodes is minimized. A generalized framework for this is proposed in [28]. In addition to latency, peer-to-peer file systems can also use scalar metrics such as the number of IP routing hops, bandwidth, or geographic distance [73].

Storage systems that are limited to a single datacenter may be less concerned about available bandwidth or access latency than systems spanning a wide-area network. Instead, datacenter-based storage systems typically focus on the current load on the storage servers when determining where to place or relocate replicas [58], thereby reducing imbalances across the cluster. A number of techniques have been employed to share information about current storage system load and decide optimal placement strategies, including as chained-declustering [56] and

erasure-coding with voting [75].

5.6 Load Balancing

In a storage system containing replicated copies of data, load balancing techniques are frequently employed to distribute read requests across multiple disks to reduce hot spots and improve overall storage bandwidth for popular files. Load balancing techniques have previously been proposed in two major categories: *centralized* architectures where a server or other network device is used to balance requests to a number of (slave) disks, and *distributed* architectures where the clients balance their requests without benefit of centralized coordination.

Centralized load balancing techniques have been proposed for both content servers and storage servers, and can function in a generalized fashion. In such systems, a front-end server distributes requests to a collection of back-end servers based on the content being requested and the current load of each back-end server [65]. By balancing based on the content being requested, cache effectiveness can be improved, and back-end servers specialized towards specific types of content.

Such centralized load balancing need not be limited to a centralized "server" in the traditional sense, as other network devices could play a similar role. Anderson *et al.* proposed a load-balancing architecture where a switching filter is installed in the network path between client and network-attached disks [22]. This filter is

part of the network itself (for example, a switch or NIC) and is responsible for intercepting storage requests, decoding their content, and transparently distributing the requests across all storage systems downstream. The client only sees a single storage system accessed at some virtual network address.

In contrast to these centralized techniques, distributed architectures are possible where the clients automatically load balance their requests across a collection of storage resources. Wu *et al.* introduced a distributed client-side hash-based technique to dynamically load-balance client requests to remote distributed storage servers with replicated (redundant) data on a LAN [83]. In this scheme, clients are aware of the existence of multiple copies of replicated data, and choose between the available replicas. This architecture is useful primarily for disks located on the same LAN, where network latency is low and essentially uniform, and the benefit gained by accessing a lightly loaded disk is high. When disks are located across a WAN, or when the network is congested, then network latency can dominate the disk latency, making it more efficient overall to simply use the closest disk network-wise [83].

A distributed approach to load balancing does not have to be done with storage clients, however. Lumb *et al.* proposed an architecture where a collection of networked disks (referred to as "bricks") collaborate to distribute reads requests [57]. In this system, each brick is a unit consisting of a few disks, processor, and memory for caching. Each brick receives all write and read requests. Writes are committed

to every disk to provide replication, and one brick also caches the write in RAM. Reads are received by every brick, and only the brick with cached data "claims" the request and services it. If the requested data is not currently cached by any brick, the request is placed in a queue and then a distributed shortest-positioning-time-first (D-SPTF) algorithm is used to pick queue entries to service and thus balance load. For storage networks with low latencies (10-200us), this distributed algorithm performed equivalently to load balancing on a centralized storage server with locally attached disks [57].

CHAPTER 6

The Case for Remote Storage

The MapReduce programming model, as implemented by Hadoop, is increasingly popular for data-intensive computing workloads such as web indexing, data mining, log file analysis, and machine learning. Hadoop is designed to marshal all of the storage and computation resources of a large dedicated cluster computer. It is this very ability to scale to support large installations that has enabled the rapid spread of the MapReduce programming model among Internet service firms such as Google, Yahoo, and Microsoft. In 2008, Yahoo announced it had built the largest Hadoop cluster to date, with 30,000 processor cores and 16PB of raw disk capacity [6].

While the Internet giants have the application demand and financial resources to provision one or more large dedicated clusters solely for MapReduce computation, they represent only a rarefied point in the design space. There are a myriad of smaller firms that could benefit from the MapReduce programming model, but do not wish to dedicate a cluster computer solely to its operation. In this market, MapReduce computation will either be lightweight — consuming only a fraction

of all nodes in the cluster — or intermittent — consuming an entire cluster, but only for a few hours or days at a time, or both. MapReduce will share the cluster with other enterprise applications. To capture this new market and bring MapReduce to the masses, Hadoop needs to function efficiently in a heterogeneous datacenter environment where it is one application among many.

Modern datacenters often employ virtualization technology to share computing resources between multiple applications, while at the same time providing isolation and quality-of-service guarantees. In a virtualized datacenter, application images can be loaded on demand, increasing system flexibility. This dynamic nature of the cluster, however, motivates a fresh look at the storage architecture of Hadoop. Specifically, in a virtualized datacenter, the local storage architecture of Hadoop is no longer viable. After a virtual machine image is terminated, any local data still residing on the disk may fall under the control of the next virtual machine image, and thus could be deleted or modified. Further, even if the data remained on disk, there is no guarantee that when Hadoop is executed again — several hours or days in the future — it will receive the same set of cluster nodes it had previously. They might be occupied by other currently running applications. In a virtualized datacenter, a persistent storage solution based on networked disks is necessary for Hadoop. To draw a distinction from the traditional local storage architecture of Hadoop, this new design will be referred to as *remote storage*.

This chapter will serve to further motivate the design of a remote storage archi-

tecture for Hadoop and provide the necessary background and related work. Subsequent chapters will investigate design specifics. In this chapter, current trends in datacenter systems will first be discussed, such as virtualization and the emergence of cloud computing and platform-as-a-service technology. Second, a virtualization framework called Eucalyptus will be described, as it provides a private cloud computing framework suitable for sharing a cluster between MapReduce and other applications. The operation of Hadoop in this virtualized environment will be discussed, as this motivates why persistent network-based storage is necessary. Third, the concept of accessing storage resources across a network will be shown to be viable due to the access characteristics of Hadoop and the raw performance potential of modern network and switching technologies. Finally, some of the inherent advantages of remote storage architectures will be described. These are due to the decoupling of storage and computation resources, which previously were tightly coupled.

6.1 Virtualization and Cloud Computing

Virtualization technology is transforming the modern datacenter. Instead of installing applications directly onto physical machines, applications and operating systems are installed into virtual machine images, which in turn are executed by physical servers running a hypervisor. Virtualizing applications provides many benefits, including consolidation — running multiple applications (with different

operating system requirements) on a single physical machine — and migration — transparently moving applications across physical machines for load balancing and fault tolerance purposes. In this environment, the datacenter becomes a pool of interchangeable computation resources that can be leveraged to execute whatever virtual machine images are desired.

Once all applications are encapsulated as virtual machine images and the datacenter is configured to provide generic computation resources, it becomes possible to outsource the physical datacenter entirely to a third-party vendor. Beginning in 2006, Amazon started their Elastic Compute Cloud (EC2) service, which allows generic x86 computer instances to be rented on-demand [12]. In this canonical example of *public cloud computing*, customers can create virtual machine images with the desired operating system and applications, and start and stop these images on demand in Amazon's datacenter. Customers are billed on an hourly basis only for resources actually used. Such a capability is particularly useful for applications that vary greatly in terms of resource requirements, saving clients from the expense of building an in-house datacenter that is provisioned to support the highest predicted load.

Not every application, however, is suitable for deployment to public clouds operated by third party vendors and shared with an unknown number of other customers. Medical records or credit card processing applications have security concerns that may be challenging to solve without the cooperation of the cloud

vendor. Further, many other business applications may require higher levels of performance, quality-of-service, and reliability that are not guaranteed by a public cloud service that, by design, keeps many details of the datacenter architecture and resource usage secret. Thus, there is a motivation to maintain the administrative flexibility of cloud computing but to run the virtual machine images on locally-owned machines behind the corporate firewall. This is referred to as *private cloud computing*. To meet this need, a new open-source framework called Eucalyptus was released in 2008 to allow the creation of private clouds. Eucalyptus implements the same API as Amazon's public cloud computing infrastructure, allowing for application images to be migrated between private and public servers. By maintaining API compatibility, the private cloud can be configured, if desired, to execute images onto the public EC2 system in peak load situations, but otherwise operate entirely within the private datacenter under normal load. Further, API compatibility allows many of the same administrative tools to be used to manage both platforms.

The private cloud computing model proposed by Eucalyptus is an attractive solution to an enterprise that wants to share a datacenter between MapReduce (Hadoop) computation and other programming models and applications. To explore this usage model, the Eucalyptus architecture is described along with its default local and network storage options.

6.2 Eucalyptus

Eucalytpus is a cloud computing framework that allows the creation of private clouds in enterprise datacenters [10, 13]. Although there are different ways to accomplish this goal, Eucalyptus was chosen for this thesis because it provides a coherent vision for sharing a single datacenter or cluster computer between many applications through the use of virtualization technology. Further, its vision is compatible (and, in many ways, identical) with the current industry leader for public cloud computing. Eucalyptus provides API compatibility Amazon Web Services (AWS), which allows management tools to be used in both environments and for computing images to be migrated between clouds as desired. Further, Eucalyptus is available as an open-source project that can be easily profiled, modified, and run on the same commodity hardware (x86 processors, SATA disks, and Ethernet networks) that supports traditional Hadoop clusters. This framework is designed for compatibility across a broad spectrum of Linux distributions (*e.g.*, Ubuntu, RHEL, OpenSUSE) and virtualization hypervisors including KVM [15] and Xen [17]. It is the key component of the Ubuntu Enterprise Cloud (EUC) product, which advertises that an entire private cloud can be installed from the OS up in under 30 minutes. During testing, installation was completed in that time period, but further configuration (and documentation reading to understanding the various configuration options) took significantly longer.

A Eucalyptus cluster consists of many cloud nodes, each running one or more

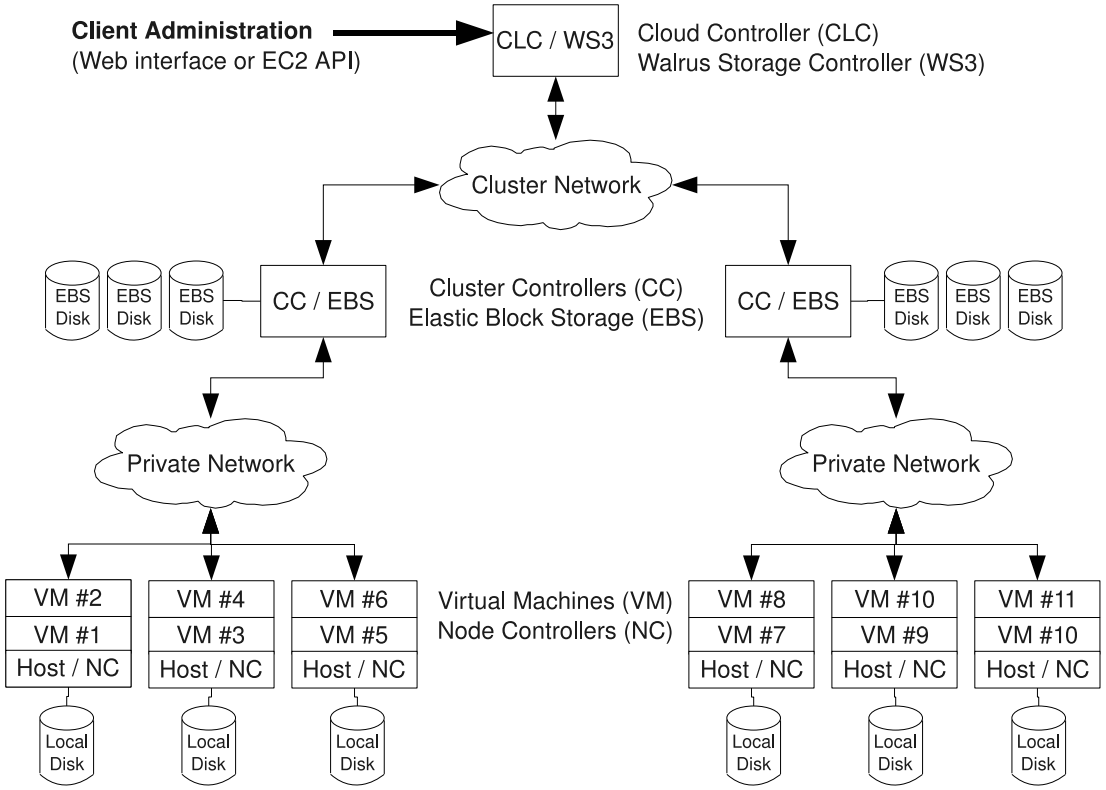


Figure 6.1: Eucalyptus Cluster Architecture [82]

virtual machine images and each equipped with at least one local disk to store the host OS and hypervisor software. Beyond the cloud nodes, a number of specialized nodes also exist in the cluster to provide storage and management services. The arrangement of a Eucalytpus cluster and its key software services is shown in Figure 6.1. These services include:

Cloud Controller (CLC) — The cloud controller provides high-level management of the cloud resources. Clients wishing to instantiate or terminate a virtual machine instance interact with the cloud controller through either a web interface or SOAP-based APIs that are compatible with AWS.

Cluster Controller (CC) — The cluster controller acts as a gateway between

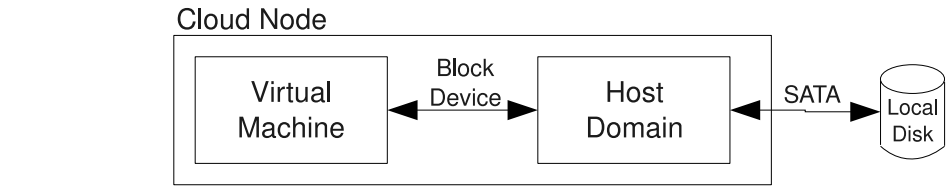
the CLC and individual nodes in the datacenter. It is responsible for controlling specific virtual machine instances and managing the virtualized network. The CC must be in the same Ethernet broadcast domain as the nodes it manages.

Node Controller (NC) — The cluster contains a pool of physical computers that provide generic computation resources to the cluster. Each of these machines contains a node controller service that is responsible for fetching virtual machine images, starting and terminating their execution, managing the virtual network endpoint, and configuring the hypervisor and host OS as directed by the CC. The node controller executes in the host domain (in KVM) or driver domain (in Xen).

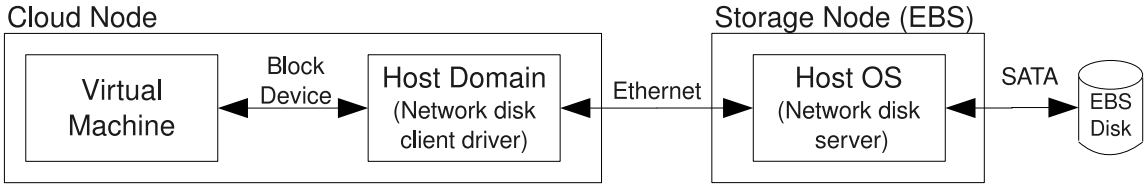
Elastic Block Storage Controller (EBS) — The storage controller provides persistent virtual hard drives to applications executing in the cloud environment. To clients, these storage resources appear as raw block-based devices and can be formatted and used like any physical disk. But, in actuality, the disk is not in the local machine, but is instead located across the network. An EBS service can export one or more disks across the network.

Walrus Storage Controller (WS3) – Walrus provides an API-compatible implementation of the Amazon S3 (Simple Storage Service) service. This service is used to store virtual machine images and application data in a file, not block, oriented format.

In Eucalyptus, cluster administrators can configure three different types of storage to support virtualized applications. The first type of storage is provided by



(a) Eucalyptus Local Storage Architecture



(b) Eucalyptus Network Storage Architecture

Figure 6.2: Eucalyptus Storage Architectures — Local Disk and Network Disk

the WS3 controller and allows data to be accessed at the object level via HTTP. It is not investigated further here because in its current Eucalyptus implementation it is provided by a single centralized service, and thus represents an obvious bottleneck for cluster scalability.[1] The second two types of storage are suitable for many types of data-intensive applications, however. The two options are *ephemeral local storage* that exists only as long as the virtual machine is active, and *persistent network-based storage*. From the perspective of an application running inside a virtual machine instance, both options appear identical. A standard block-based device abstraction is used which allows guests to format the device with a standard filesystem and use normally. These two architectures are shown in Figure 6.2.

The first architecture, local storage, uses a file on the locally-attached hard drive

[1]Eucalyptus WS3 is API compatible with Amazon's S3 service, which does scale to support massive numbers of clients. Thus, it is not the S3 API that limits scalability, merely the current centralized implementation in Eucalyptus.

of the cloud node for backing storage. This file is ephemeral and only persists for the life of the target virtual machine, and is deleted by the node controller when the virtual machine is terminated. Under the direction of the node controller, the hypervisor maps the backing file into the virtual machine via a block-based storage interface. The virtual machine can use the storage like any other disk.

In contrast to the first architecture, the network storage architecture eschews the local disk in favor of a networked disk that can provide persistent storage even after a specific virtual machine is terminated. Because the storage is network-based, when that virtual machine is restarted later, it can easily access the same storage resources regardless of where in the cluster it is now assigned. It does not need to be assigned to its original node. In this architecture, a file is used as backing storage on one of the EBS-attached hard drives. On the EBS node, a server process exports that file across the network as a low-level storage block device. Eucalyptus uses the non-routable (but lightweight) ATA over Ethernet protocol for this purpose, which requires that the virtual machine and EBS server be on the same Ethernet segment [8]. Across the network on the cloud node, an ATA over Ethernet driver is used in the host domain to mount the networked disk. The driver is responsible for encapsulating ATA requests and transmitting them across the Ethernet network. The node controller instructs the hypervisor to map the virtual disk provided by the driver into the virtual machine using the same block-based storage interface used in the local storage architecture.

To run Hadoop in the Eucalyptus cloud environment, both ephemeral and persistent storage resources are necessary. Ephemeral or scratch storage is used for temporary data produced in MapReduce computations. Typically, a Map process will buffer temporary key/value pairs in memory after processing, and spill them to disk when memory resources run low. This data does not need persistent storage, as it is consumed and deleted during the Reduce stage of the application, and can always be regenerated if lost due to failure. The local storage architecture provided by Eucalyptus is well suited for this role, as this storage is deleted when the virtual machine is stopped.

Although ephemeral storage is efficiently supported by the local storage design in Eucalyptus, persistent HDFS data is not. Persistent data cannot be left on the local disk after MapReduce computation is finished because Eucalyptus will delete it. Even if this behavior was changed to protect the data, other problems remain. For instance, other applications might need the local storage resources in the future. Or, other applications might still be running on the node when MapReduce computation is resumed at a later point in time, posing the question of what to do. Should the data should be migrated to where it is needed, the current application migrated elsewhere to allow MapReduce to run on the node, or the data be accessed remotely instead? All three options pose challenges.

A naive scheme to provide persistent network storage for HDFS without otherwise changing the storage architecture would be to store the data inside the virtual

machine image, which is located (when not in use) on a network drive. When MapReduce computation is started, this much larger image would be copied to a cluster node, and then Hadoop could use local storage exclusively for the duration of program execution. The data would be copied back to the network drive (with the virtual machine image) when finished. Such a scheme has several drawbacks. First, MapReduce startup latency would be excessively high, due to the volume of data that needs to be moved, and the fact that the copy would need to be 100% complete on all nodes before MapReduce could initialize and begin execution. A similarly lengthy copy would also be needed once MapReduce computation is finished. Second, the full upfront data migration inherent in this scheme will be at least partially unnecessary. The MapReduce application will certainly not access all the data copied immediately, and even over long time scales may access only a portion of the full HDFS data set. Third, this design requires twice the storage capacity in order to store data both on local nodes and in network storage with the virtual machine images. Finally, this design poses a bandwidth concern. There is no guarantee that virtual machine images are stored in the same rack as the cluster nodes. In fact, virtual machine images may be stored in a centralized location elsewhere in the datacenter. In such a configuration, data would need to be copied across the cross-switch links, increasing the potential for network congestion.

These reasons motivate the design of a better persistent network-based storage architecture. This architecture should allow MapReduce applications to access

only the specific data currently needed, should store data in the same place even when MapReduce computation is (temporarily) halted, and co-locate that storage with computation in the same rack and attached to the same network switch. There are many possible ways to enable this in a virtualized environment such as Eucalyptus, and specific options will be discussed later in this thesis in Chapter 7. But, regardless of the specific network disk architecture used to provide persistent storage, long term performance trends support the vision of accessing HDFS data across the network.

6.3 Enabling Persistent Network-Based Storage for Hadoop

There are several major reasons why, at a high level, networked disks can provide high levels of storage performance for DC clusters running frameworks such as Hadoop.

First, DC applications like Hadoop use storage in a manner that is different from ordinary applications. Application performance is more dependant on the storage bandwidth to access their enormous datasets than the latency of accessing any particular data element. Furthermore, data is accessed in a streaming pattern, rather than random access. This means that data could potentially be streamed across the network in a pipelined fashion and that the additional network latency to access the data stream should not affect overall application performance.

Second, network bandwidth exceeds disk bandwidth for commodity technolo-

gies, making it possible for an efficient network protocol to deliver full disk bandwidth to a remote host. To show how a commodity network can be provisioned to deliver the full bandwidth of a disk to a client system, network and hard drive performance trends over the past two decades were evaluated, as shown in Figure 6.3. The disk bandwidth was selected as the high-end consumer-class (not server-class) drive introduced for sale in that particular year. The network bandwidth was selected from the IEEE standard, and the network dates are the dates the twisted-pair version of the standard was ratified. This is typically later than the date the standard was originally proposed for fiber or specialty copper cables, which are too expensive for DC cluster use.

Since the introduction of 100Mb/s Fast Ethernet technology, network bandwidth has always matched or exceeded disk bandwidth. Thus, it is reasonable to argue that the network will not constrain the streaming bandwidth of disks accessed remotely, and that such bandwidth will be cost-effective to provide. Note that this is only single device bandwidth – more network bandwidth could be provided for faster disks by trunking links between hosts and the switch. Similarly, in the case of faster networks, more disk bandwidth could be achieved by ganging multiple disks on a single network link, thus allowing the network link to be more efficiently and fully utilized.

Third, modern network switches offer extremely high performance to match that of the raw network links. A typical 48- or 96-port Ethernet switch can provide

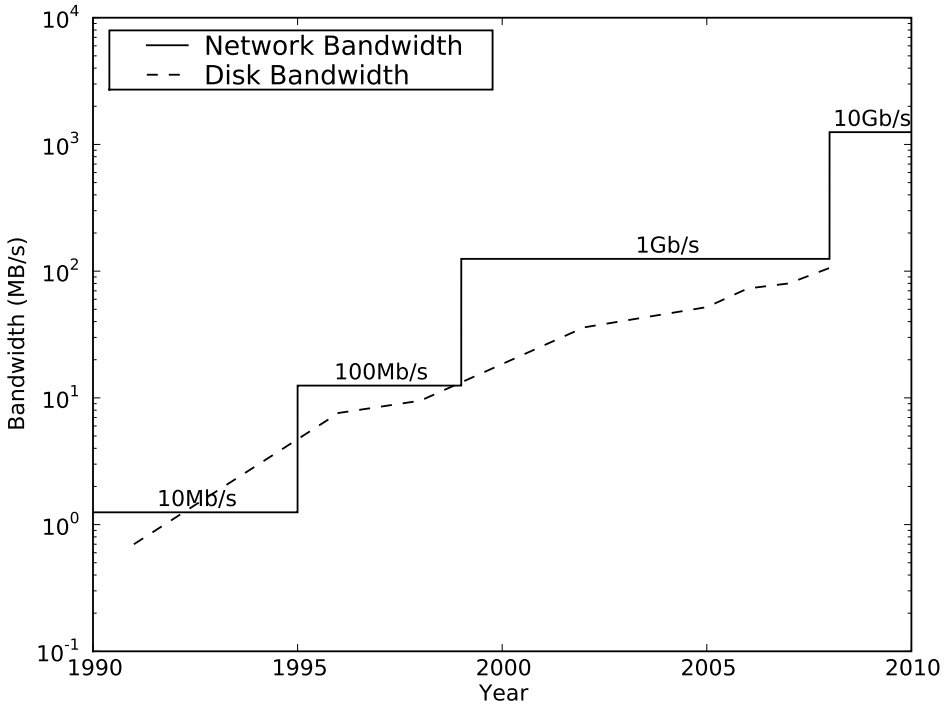


Figure 6.3: Scaling Trends - Disk vs Network Bandwidth

the full bisection bandwidth across its switching fabric, such that an entire rack of hosts can communicate with each other at full network speed. Furthermore, even a modestly priced (around $3000) datacenter switch not only offers full switching bandwidth, but also provides low latency forwarding of under 2μs for a minimum-sized Ethernet frame [5]. In addition, Ethernet switches are starting to emerge in the marketplace that perform cut-through routing, which will lead to even lower forwarding latencies. Compared to hard disk[2] seek latencies, which are measured in milliseconds, the forwarding latency of modern switches is negligible. Data-

[2]While solid-state disks have lower latencies, they are still 100s of microseconds. Regardless, conventional hard disks are the likely choice for mass storage in DC clusters in the foreseeable future due to capacity and cost.

center switches also provide the capability to do link aggregation, where multiple gigabit links are connected to the same host in order to increase available bandwidth. These high performance switches will incur minimal overhead to network storage that is located in the same rack as the client computation node.

In this chapter, remote storage has been motivated as a requirement for Hadoop to function in a virtualized datacenter shared with other applications. Further, the concept of accessing storage resources has been shown to be viable due to the access characteristics of Hadoop and the raw performance potential of modern network and switching technologies. Next, potential benefits of a storage architecture incorporating remote or network disks are discussed.

6.4 Benefits of Remote Storage

Remote storage is necessary to allow Hadoop to function in a virtualized datacenter shared with many other applications. Although using a remote storage architecture may entail performance tradeoffs compared to a local disk architecture, it does have several potential advantages. These arise from the fact that computation and storage resources are no longer bound together in one tightly coupled unit, as they are in a traditional Hadoop node.

Resource Provisioning — Remote storage allows the ratio between computation and storage in the cluster to be easily customized, both during cluster construction and during operation. This is in contrast to the traditional Hadoop local

storage architecture, which places disks inside the compute nodes and thereby assumes that the storage and computation needs will scale at the same rate. If this assumption is not true, the cluster can become unbalanced, forcing the purchase and deployment of extra disks or processors that are not strictly necessary, wasting both money and power during operation. For example, if more computation (but not storage) is needed, extra compute nodes without disks can be added, but they will need to retrieve all data remotely from nodes with storage. Or, unneeded disks will be purchased with the new compute nodes, increasing their cost unnecessarily. Similarly, if more storage or storage bandwidth (but not computation) is needed beyond the physical capacity of the existing compute nodes, extra compute nodes with more disks will need to be added even though the extra processors are not necessary.

Load Balancing — Remote storage has the potential for more effective load balancing that eliminates wasted storage bandwidth. Instead of over-provisioning all compute nodes with the maximum number of disks needed for peak local storage bandwidth, it would be cheaper to simply provision the entire rack with enough network-attached disks to supply the average aggregate storage bandwidth needed. Individual compute nodes could consume more or less I/O resources depending on the instantaneous (and variable) needs of the application. The total number of disks purchased could thus be reduced, assuming that each compute node is not consuming 100% of the storage bandwidth at all times and as-

suming that many disks are purchased to increase I/O bandwidth and not simply for the raw storage capacity.

Fault Tolerance — A failure of the compute node no longer means the failure of the associated storage resources. Thus, the distributed file system does not have to consume both storage and network bandwidth making a new copy of the data from elsewhere in the cluster in order to maintain the minimum number of data replicas for redundancy. Disk failure has a less detrimental impact, too. New storage resources can be mapped across the network and every disk connected to the same network switch offers equivalent performance.

Power Management — In a cluster with a remote storage architecture, fine-grained power management techniques can be used to sleep and wake stateless compute nodes on demand to meet current application requirements. This is not possible in a local storage architecture, where the compute nodes also participate in the global file system, and thus powering down the node removes data from the cluster. Further, because computation is now an independent resource, it is also possible to construct the cluster with a heterogeneous mix of high and low power processors. The runtime environment can change the processors being used for a specific application in order to meet administrative power and performance goals.

CHAPTER 7

Remote Storage in Hadoop

In the previous chapter, the motivations for a remote storage architecture for Hadoop were discussed. MapReduce frameworks such as Hadoop currently require a dedicated cluster for operation, but such a design limits the spread of this programming paradigm to only the largest users. Smaller users need to run MapReduce on a cluster computer shared with other applications. Virtualization might be used to facilitate sharing, as well as gain benefits such as increased flexibility and security isolation. In such an environment, the MapReduce framework will be loaded and unloaded on demand and typically execute on a different set of nodes with each iteration. This motivates the deployment of a remote storage architecture for Hadoop, because the traditional local architecture has problems in this environment. For example, if data is stored on a local disk and then the MapReduce framework is stopped, the next application to execute on that node could potentially delete the data and re-use the disk. Or, even if the data remains intact, there is no guarantee that MapReduce will be scheduled on the same node in a future invocation, rendering the data inaccessible. Storing persistent HDFS

data on network disks instead of locally-attached disks will eliminate these problems.

The desired performance and scale of Hadoop using a remote storage architecture is somewhat different from Hadoop using a local storage design, due to the different usage scenario. First, the scale of the cluster is inherently smaller. Following the previous logic, if the application scale was large enough to require thousands of nodes, then such an application could certainly justify a dedicated cluster computer built with the traditional Hadoop architecture. MapReduce applications sharing a cluster with other workloads are necessarily smaller, requiring tens to perhaps hundreds of nodes on a part-time basis. Second, in such a usage scenario, the ability to share the cluster between multiple applications has a higher priority than the ability to maximize the performance of a given hardware budget. This is not to say that performance is an unimportant goal, just that high performance is not the only goal of the system. Applications requiring the highest possible performance can justify a dedicated cluster built, once again,with the traditional Hadoop local architecture.

In this Chapter, the design space of viable remote storage architectures for Hadoop is explored, and several key configurations identified. Next, these configurations are evaluated in terms of achieved storage bandwidth and processor efficiency to identify the best approach. With the most efficient remote storage architecture selected, problems related to replica target assignments by the Na-

meNode are identified that dramatically hurts performance due to storage congestion. An improved scheduling framework is proposed and evaluated to eliminate this bottleneck. Next, the impact of virtualization on storage and network I/O bandwidth is examined in order to test the viability of remote storage in a cloud computing framework. Finally, a complete design is described for data-intensive MapReduce computation in a cloud computing environment shared with many other applications.

7.1 Design Space Analysis

There is a large design space of possible remote storage architectures for Hadoop. In this section, a few key architectures that can achieve high storage bandwidth are described, compared, and evaluated in terms of processor overhead. These architectures are independent of any storage architecture provided by a virtualization or cloud computing system such as Eucalyptus. Integration with existing systems will be discussed later in this chapter.

When evaluating potential network storage architectures for Hadoop, a few restrictions were imposed, including the use of commodity hardware, a single network, and a scalable design.

Commodity Hardware — Any proposed architecture has to be realizable with only commodity hardware, such as x86 processors and SATA disks. This lowers the up-front installation cost of the cluster computer. As a practical matter, this

necessitates the use of Ethernet as the network fabric.

Single Network — Any proposed architecture has to use a single (converged) network in the datacenter, carrying both storage and application network traffic. Using a single network reduces cluster installation cost (from separate network cards, switches, and cabling) and administrative complexity. This restriction eliminates a number of designs involving dedicated storage-area networks.

Scalable Design — Any proposed architecture has to be scalable to support MapReduce clusters of varying sizes. Although it is unlikely that this design will be used for thousands of nodes in a shared datacenter — because any application at that scale could justify a cluster for dedicated use — scalability to tens or hundreds of nodes is a reasonable target. Ideally, a remote storage design should maintain similar scalability to the traditional local storage architecture. Because of this goal, no designs involving centralized file servers were considered. This restriction eliminated using NFS servers, and, perhaps more importantly, using the S3 storage service provided with Eucalyptus (named WS3). S3 would be a complete replacement for HDFS, as it provides all the necessary data storage and file namespace management functionality. Hadoop applications can directly use S3 storage without the need for any NameNode or DataNode services. S3 allows data to be manipulated at the file or object level via HTTP, and does not use a traditional disk block abstraction. But, as previously described in Section 6.2, Eucalyptus implements S3 as a centralized service provided by a single node, and thus

is an obvious bottleneck impeding the performance and scalability of MapReduce on even a small-scale cluster. This is an implementation issue, not a fundamental challenge with S3. Amazon provides a commercial S3 service hosting hundreds of billions of objects, and uses a decentralized architecture to support large numbers of concurrent clients.

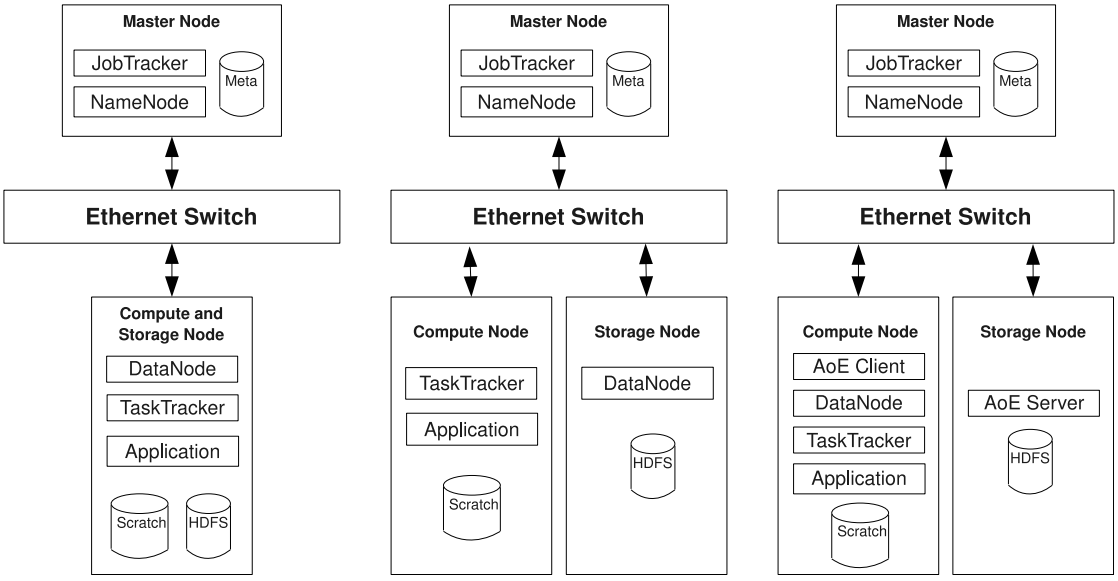
Preserve Locality — Any proposed architecture should preserve storage locality, albeit at the level of the same rack (attached to the same network switch), instead of at the same node. This requires effort by both the storage allocator (when deciding where to store blocks) and job scheduler (when deciding where to run tasks), and some level of integration between the two, such as when the job scheduler queries the storage system for location information. As a practical matter, this discourages the wholesale replacement of the DataNode and NameNode services with an alternative architecture such as Amazon S3. For example, if S3 were used, the Hadoop job scheduler would have no API available to determine the physical location of data in the cluster, and thus would be unable to schedule tasks in a locality-aware fashion. Such an API would have to be added.

Based on these restrictions, a number of viable storage architectures for Hadoop were identified.

7.1.1 Architecture Overview

The storage architectures under consideration for Hadoop are shown in Figure 7.1. The architectures include the default local architecture, a remote architecture using the standard Hadoop network data transfer protocol, and a remote architecture using the ATA over Ethernet (AoE) protocol. In the figure, the 4 key Hadoop software software services are shown, including the MapReduce engine components (JobTracker plus one of many TaskTrackers) and HDFS components (NameNode plus one of many DataNodes). In all architectures, the JobTracker and NameNode services continue to run on dedicated nodes with local storage. For a small cluster, they can share a single node, while in a larger cluster, separate nodes may be needed.

The location of key disks in the cluster are also shown. Disks labeled *HDFS* are used exclusively to store HDFS block data. Disks labeled *Meta* are used to store Hadoop metadata used by the JobTracker and NameNode, such as the filesystem namespace and mapping into HDFS blocks. Finally, disks labeled *Scratch* store MapReduce intermediate (temporary) data, such as key/value pairs produced by a Map task but not yet consumed by a Reduce task. Storage for the operating system and applications is not shown, as that space could be shared with the scratch or metadata disks without degrading performance significantly. HDFS storage is shown with a dedicated disk due to provide high storage bandwidth. If needed based on application requirements, storage bandwidth could be increased



(a) Local Architecture (b) Split Architecture (c) AoE Architecture

Figure 7.1: Comparison of Local and Remote (Split, AoE) Storage Architectures

by adding multiple disks.

The local architecture, shown in Figure 7.1(a), is the traditional Hadoop storage architecture initially described in Chapter 2. This architecture was designed with the philosophy of moving the computation to the data. Here, the DataNode service uses the HDFS disk that is directly attached to the system for persistent storage, and the TaskTracker service uses the scratch disk that is directly attached to the system for temporary storage.

In contrast to the local architecture, the Split architecture shown in Figure 7.1(b) accesses data across the network. This architecture exploits the inherent flexibility of the Hadoop framework to run the DataNode service on machines other than those running the application and TaskTracker service. In essence, these two services, which previously were tightly coupled, are now split apart. Hadoop already

uses a network protocol to write block replicas to multiple nodes across the network, and to read data from remote DataNodes in case the job scheduler is unable to assign computation to a node already storing the desired data locally. In this design, there is never any local HDFS storage. The computation nodes (running the application and TaskTracker service) are entirely disjoint from the storage nodes (running the DataNode service and storing HDFS blocks). Scratch storage is still locally provided, and will be used to store temporary key/value pairs produced by the Map stage and consumed by the Reduce stage. These intermediary values are not stored in the HDFS global filesystem because of their short lifespan. This Split architecture has the advantage of being simple to implement using existing Hadoop functionality.

The final AoE architecture shown in Figure 7.1(c) replaces the standard Hadoop network protocol with a different protocol — ATA over Ethernet — to enable remote storage. Here, the DataNode, TaskTracker, and application reside on the same host, and communicate via local loopback. The actual HDFS disk managed by the DataNode is not locally attached, however. The AoE protocol is used instead to map a remote disk, attached somewhere else in the Ethernet network, to the local host as a block device. In this design, the DataNode and the rest of the Hadoop infrastructure are unaware that storage is being accessed across the network. AoE provides an abstraction that storage in this configuration is still locally-attached. As such, this architecture is similar to the default network storage

architecture provided by Eucalyptus. As shown previously in Figure 6.2(b), that architecture also uses AoE to transparently provide the illusion of locally-attached storage. In this AoE architecture for Hadoop, scratch storage is still locally provided for application use.

There are several key differences between the Split and AoE architectures that both provide persistent network storage. First, the network protocol used for storage traffic is different. The Split architecture uses the native Hadoop socket-based protocol to transfer data via TCP, while the AoE architecture uses the ATA over Ethernet protocol. AoE was conceived as a lightweight alternative to more complex protocols operating at the TCP/IP layer. This non-routable protocol operates solely at the Ethernet layer to enable remote storage. Second, the two architectures differ in terms of caching provided. In the Split architecture, the only disk caching is provided at the storage node by the OS page cache, which is on the opposite side of the network from the client, thus incurring higher latency. But, in the AoE architecture, caching is inherently provided both at the storage node and at the compute node, thus providing lower latency, but also a duplication of effort. Depending on application behavior, there may be no effective performance difference however, as data-intensive applications typically have working sets too large to effectively cache. Third, both architectures differ in terms of the responsibilities of the storage node. Conceptually, the AoE server is a less complicated application than the DataNode service, as it only needs to respond to small AoE

packets and write or read the requested block, not manage the user-level filesystem and replication pipeline. The processor overhead of each architecture will be evaluated later in this chapter, but reducing the processor overhead of the storage node is a desirable goal, as that would allow those nodes to be built from slower, lower-power, and cheaper processors.

Next, these three architectures will be evaluated in terms of bandwidth and processor overhead.

7.1.2 Storage Bandwidth Evaluation

In this section, the three architectures under consideration are evaluated in terms of storage bandwidth provided to the MapReduce application. For test purposes, a subset of the FreeBSD-based test cluster previously described in Section 3.1 was isolated with separate machines used for the master node, compute node, and storage node. A synthetic Hadoop application was used to write and then read back 10GB of data from persistent HDFS storage in a streaming fashion. In all tests, the raw disk bandwidth for the Seagate drive used for HDFS storage is less than the raw network bandwidth, and as such the network should not impose a bandwidth bottleneck on storage. The results of this test are shown in Table 7.1.

The bandwidth to local storage is first shown for comparison purposes, as this provides a baseline target to reach. The Split architecture achieves 98-100% of the local storage bandwidth using the default cluster configuration. The AoE architec-

Configuration	Bandwidth (MB/s)	
	Write	Read
Local	67.3	70.1
Split	67.8	68.7
AoE-Default	16.8	40.5
AoE-Modified	46.7	57.6

Table 7.1: Storage Bandwidth Comparison

ture, however, shows a significant performance penalty with its default configuration, achieving only 25-57% of the local storage bandwidth.

The cause of the AoE performance deficit was investigated and tracked to several root causes. First, AoE is sensitive to Ethernet frame size. Each AoE packet is an independent entity, and thus can only write or read as many bytes as fits into the Ethernet frame, subject to the 512 byte granularity of ATA disk requests. Thus, a standard 1500-byte Ethernet frame can carry 1kB of storage data, and a 9000 byte "jumbo frame" packet can carry at most 8.5kB of storage data. Thus, the AoE client (in this case, the compute node) has to fragment larger storage requests into a number of consecutive AoE packets.

Second, AoE is sensitive to disk request size. Each standard Ethernet frame arrives at the storage node with only a 1kB payload of data for the AoE server. A naive server implementation will dispatch each small request directly to the disk. Regardless of whether consecutive request are to sequential data on disk or not, the disk controller and low-level storage layers will be overwhelmed by the sheer number of small requests, limiting effective bandwidth. As an example of this performance bottleneck, the Seagate drive used in the test was characterized on a

local system. Sending sequential 1kB write requests to the drive yielded a write bandwidth of only 22.1 MB/s, compared to 110 MB/s using 64kB requests. A more sophisticated AoE server implementation could coalesce adjacent sequential requests into one large request that is delivered to the disk. This would decouple the size of the Ethernet frame from the size of the disk request, and allow the storage hardware to be used more efficiently.

After investigating the AoE system, the test configuration was modified and re-tested. The network was configured to use 9000 byte packets (*i.e.,* jumbo frames), and the default AoE server application was replaced with an implementation that performs packet coalescing of adjacent requests. The improved performance results are also shown in Table 7.1. At best, the AoE architecture achieves 70% of the target write bandwidth and 82% of the target read bandwidth.

The higher performance of the Split architecture compared to AoE highlights a fundamental difference in design. The Split architecture uses the native Hadoop protocol to transfer data in a streaming manner. Data is delivered directly to the receiver (either the DataNode when writing a block, or the client application when reading a block). By design, the recipient can begin processing the data immediately without waiting for the transfer to complete in its entirety. In contrast, the AoE protocol operates in a synchronous fashion. The traditional block-based interface used to link it with the data consumers (the DataNode and applications) provide no mechanism to announce the availability of a block until all data is re-

ceived. Thus, the consumer is not provided with any data until the last byte in the transfer arrives. This is a challenge inherent to all block-based protocols, and is not limited to just ATA over Ethernet.

After evaluating storage bandwidth, the processor overhead of each architecture is examined to determine relative efficiency per byte transferred.

7.1.3 Processor Overhead Evaluation

In this section, the three architectures under consideration are evaluated in terms of processor overhead per unit of bandwidth. Rather than focus on raw performance, this discussion focuses on the efficiency of each architecture.

To evaluate the architectures shown in Figure 7.1, a subset of the FreeBSD-based test cluster previously described in Section 3.1 was isolated. Two nodes were used for the local architecture, and three nodes were used for both remote architectures under test. The first node — the *master node* — runs the JobTracker and NameNode services. The second node — the *compute node* — runs the application and TaskTracker service, and also the DataNode in the case of the local architecture. Finally, the third node — the *storage node* — houses the HDFS disk and runs the service that exports storage data across the network (either the DataNode or an AoE server).

In each of the three test configurations, a synthetic Hadoop application was used to write and then read back 10GB of data from persistent HDFS storage in a streaming fashion. User-space system monitoring tools were used on both the

compute node and storage node (in the remote architectures) to capture processor overhead and categorize time consumed by user-space processes, the operating system, and interrupt handlers. The master node was not profiled, because its workload (HDFS namespace management and job scheduling) remains unchanged in any proposed design. User-space processes include (when applicable) the test application, Hadoop framework, and AoE server application. Operating system tasks include (when applicable) the network stack, network driver, AoE driver, filesystem, and other minor responsibilities. Interrupt handler work includes processing disk and network interrupts, among other less significant tasks. Figure 7.2 summarizes the processor overhead for each storage architecture, normalized to storage bandwidth. Conceptually, this is measuring processor cycles per byte transferred. But, due to limitations in the available monitoring tools, the actual measurement units are aggregate processor percent utilization per megabyte transferred, times 100.

Before describing the performance of each individual architecture in detail, there are a few high level comments on the test and system behavior to discuss. First, the synthetic test application used here is very lightweight, doing minimal processing beyond writing or reading data. Thus, the majority of the user-space overhead is incurred by the Hadoop framework. Other MapReduce applications would likely have higher overall processor utilization, as well as higher user-space utilization on the compute node. Second, the synthetic write test consumes more

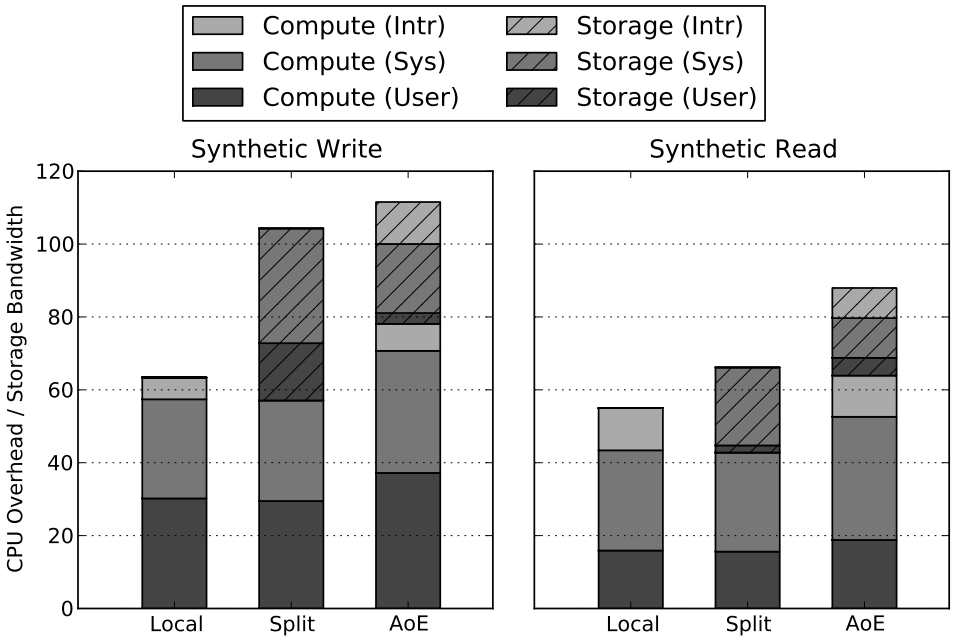


Figure 7.2: Processor Overhead of Storage Architecture, Normalized to Storage Bandwidth

processor resources per megabyte transferred than the synthetic read test. When writing data to HDFS, the Hadoop framework has a number of responsibilities that are not needed when reading data. These responsibilities include communicating with the NameNode for allocation and ensuring that data is transferred to all desired replicas successfully. To accomplish this, the outgoing data stream is buffered several times, fragmented into smaller pieces, and handled by several threads, each one incurring additional overhead. This complexity is not needed when reading data from HDFS. Conceptually, all a DataNode needs to do to transfer data to a client is locate the requested HDFS block and call sendfile() on the file. (As a technical point, sendfile is not supported in the Java Virtual Machine on FreeBSD, but its block-based replacement is not significantly more complex.)

Support for this observation that HDFS writes are more complex than HDFS reads is shown in the processor overhead observed in the local architecture, as reading requires about half the user-space compute resources as write for an equivalent bandwidth.

As shown in Figure 7.2, the baseline local storage architecture is the most efficient, computation-wise. On the compute node, user-space processor time is spent running the application and Hadoop services including the TaskTracker and DataNode. As previously described, the overhead in Hadoop of writing data to HDFS is higher than reading data from HDFS, as evidenced by the difference in processor time between the write and read tests. System processor time on the compute node is consumed accessing the local HDFS disk and using the local loopback as an interprocess communication mechanism between the TaskTracker and the DataNode. The system overhead is symmetric for both reading and writing, and the measured overhead is consistent for both tests. Finally, the interrupt processing time is incurred managing local loopback and disk data transfer.

After testing the local architecture, the two remote storage architectures were profiled, starting with the Split architecture. This is the most efficient remote storage architecture, but enabling remote storage does incur processor overhead compared to the default local architecture. In this configuration, the user-space processing time for the compute node remains unchanged, but the work performed in that time is significantly different. Specifically, the DataNode service has been

migrated to the storage node, which means that the remaining Hadoop services are consuming more processing resources to send/receive data across the network instead of across local loopback. Thus, this part of the framework is functioning less efficiently. The DataNode service now runs on the storage node, and consumes user-space processing resources there. Once again, there is a significant different in processor overhead for the DataNode when comparing writing data against reading data.

In the Split configuration, the system processing time is also used for different tasks. Instead of transferring data across local loopback, system time is used instead in the TCP network stack. The net impact on system utilization at the compute node is unchanged, but additional system resources are required at the storage node for network processing and HDFS disk management. In addition, interrupt handling time is negligible at both the compute and storage nodes. Disk I/O does not trigger computationally intensive interrupts. Although network interrupts would normally be computationally intensive, the driver for the specific Intel Pro/1000 network interface card used in the cluster employs an interrupt moderation scheme that, in cases of high network utilization (such as during these experiments), operates the NIC in a polling mode that is not interrupt driven. Rather, received packets are simply transferred to the host at the same time that the driver schedules new packets to transmit.

The third storage architecture tested, AoE, was the least efficient computation-

wise. The compute node incurs all the user-space overhead running the application, TaskTracker, and DataNode, just as in the local architecture. In fact, the overall user-space overhead is higher when compared against the local architecture, a change that is attributed to the DataNode running less efficiently when accessing the higher latency remote (AoE) disk. The compute node also incurs system overhead using interprocess communication between the TaskTracker and DataNode services. Further, it is responsible for running the AoE driver to access the remote disk. The AoE driver accounts for the increase in system time on the compute node when compared against the local architecture. Finally, interrupt processing time is incurred on the compute node to receive AoE packets.

On the AoE storage node, a small amount of user-space time is used to run the AoE server application, while a larger amount of system time is used to access the HDFS disk and process AoE packets. Similarly, interrupt processing time is incurred to receive AoE packets. When comparing the write test versus the read test, the highest interrupt processing overhead is incurred on the system receiving AoE payload data. In the write test, the storage node is receiving the data stream, whereas in the read test, the compute node is receiving the data stream.

When comparing the interrupt processing time in the Split and AoE architectures, an interesting difference emerges. In the split architecture, there is negligible interrupt processing time, but a significant overhead in the AoE configuration. Both architectures were tested using the same Intel NICs that should minimize

interrupt processing. The cause of this difference is the behavior of the network protocol used. Take the case of reading HDFS data, for example. In the Split configuration, a single request packet can request a large quantity of data in return. This response data is sent using TCP, a reliable protocol that employs acknowledgement packets. When the long-running stream of TCP data is received by the compute node, that node sends acknowledgement packets (ACKs) in the opposite direction. Because ACKs are transmitted regularly, the device driver can learn of recently received packets at the same time without need of an interrupt. (Similarly, the storage node is sending HDFS data constantly, and can learn of received acknowledgement packets at the same time). In contrast, the AoE protocol running at the Ethernet layer uses a simpler request/response design. The compute node issues a small number of requests for small units of storage data (limited to an Ethernet frame size), and waits for replies. Because no more packets are being transmitted, the network interface card must use an interrupt to alert the device driver when the reply packets are eventually received. This argues for a fundamental efficiency improvement of the Split architecture over the AoE architecture, and for using a network protocol (such as TCP) that can transfer data in long streaming sessions, instead of the short request/reply protocol of AoE that limits message size to the Ethernet frame limit.

The data shown in Figure 7.2 indicates that the Split architecture using the standard Hadoop network protocol is more processor efficient than the AoE architec-

ture. Further, as discussed in Section 7.1.2, the Split configuration also had a higher out-of-the-box bandwidth than AoE, and required much less system configuration and tuning to get running efficiently. Another strike against the AoE architecture is that its storage node processor requirements are not significantly lower than for the Split architecture by the time interrupt overhead is included. This negates a big hoped-for advantage of AoE discussed previously, which was the ability to use a cheaper storage node processor. In fact, subsequent testing of the Split architecture showed that it is already processor efficient, and that the storage node does not need to be a high-powered system. The DataNode daemon was able to achieve equivalent storage bandwidth to the server-class Opteron processor used in the standard test cluster when the storage node was temporarily replaced with a system using an 8 watt Atom 330 processor, and it still showed over 50% processor idle time. Thus, for the remainder of this thesis, the focus will be on improving the performance and behavior of the native Hadoop remote storage system using the Split architecture.

In the next section, the performance of the Split architecture for remote storage will be evaluated with larger numbers of nodes. Here, it will be shown that the lack of locality in the cluster degrades the performance of the NameNode scheduler as soon as the cluster size is increased beyond 1 compute node and 1 server node. Modifications to the NameNode are proposed and evaluated to mitigate this problem.

7.2 NameNode Scheduling

After evaluating the computation overhead of the various remote storage architectures using a simple 1 client cluster, the same architectures were tested in a larger cluster. Unfortunately, the most efficient remote storage architecture, Split, exhibited poor scalability. The cause of this poor performance was traced to DataNode congestion instigated by poor NameNode scheduling policies. The NameNode scheduler was subsequently modified to reduce the performance bottleneck.

The performance bottleneck can be most clearly shown by comparing a simple cluster configuration with one HDFS client and one HDFS server to another cluster with two HDFS clients and two HDFS servers. To demonstrate this, the local architecture was configured with 2 or 3 nodes (1 master plus 1 or 2 compute/storage nodes), and the Split and AoE architectures were configured with 3 or 5 nodes (1 master node, 1 or 2 compute nodes, and 1 or 2 storage nodes). A simple synthetic writer and reader application was used with 1 task per compute node to access HDFS storage. 10GB of data per task was first written to HDFS, and then read back. HDFS replication was disabled for simplicity.

In this test setup, doubling the number of compute nodes and storage nodes should double the aggregate storage bandwidth. Unfortunately, the actual performance did not match the ideal results. The poor scalability of the Split configuration by default is shown in Table 7.2, along with the other architectures for

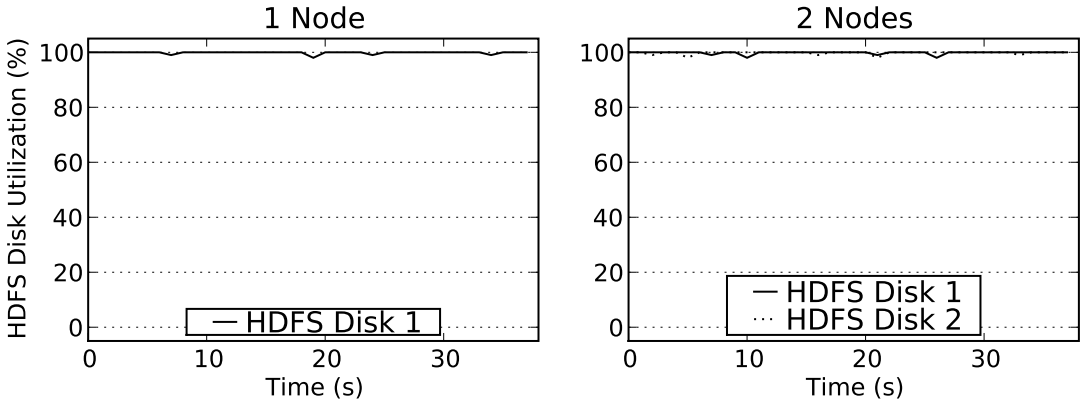
Configuration	1 Writer	2 Writers	1 Reader	2 Readers
Local	67.3	145.2	70.1	142.6
Split	67.8	61.6	68.7	51.8
AoE	16.3	27.2	37.2	79.1

Table 7.2: Aggregate Storage Bandwidth (MB/s) in 1 and 2-Node Configurations

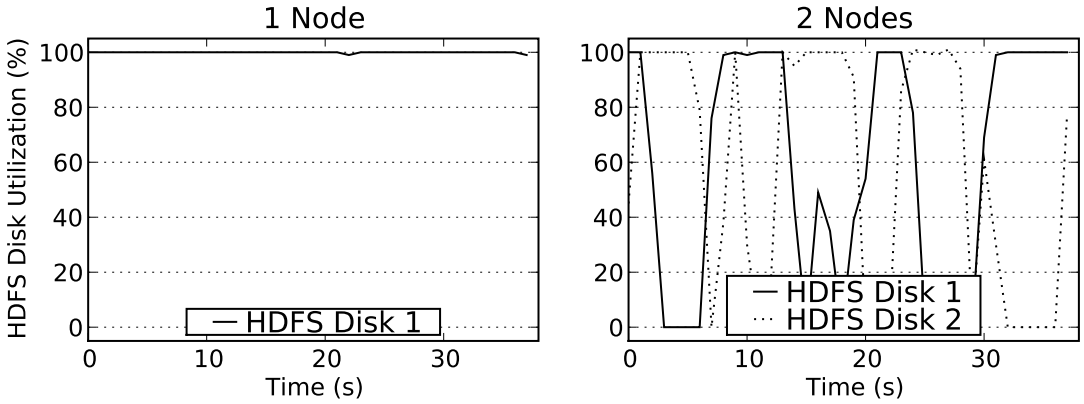
comparison.

The best scaling performance is exhibited by the traditional Hadoop local storage architecture. Here, the write and read bandwidth doubles when the number of HDFS clients and the number of HDFS disks double. Similarly, in the AoE architecture, the write bandwidth increases by 66% and the read bandwidth by 100% when the size of the cluster is doubled. (The low absolute performance of the AoE configuration can be improved through judicious configuration and the use of more sophisticated AoE server applications. This section ignores the absolute performance in favor of focusing on the scalability of the architecture.) But, while both the local and AoE architectures exhibit good performance scaling, the split architecture does not. Doubling the amount of cluster hardware actually decreases the write bandwidth by 10%, and decreases the read bandwidth by 25%. Obviously, it will be impossible to build a large Hadoop cluster if the system slows down as more nodes are added!

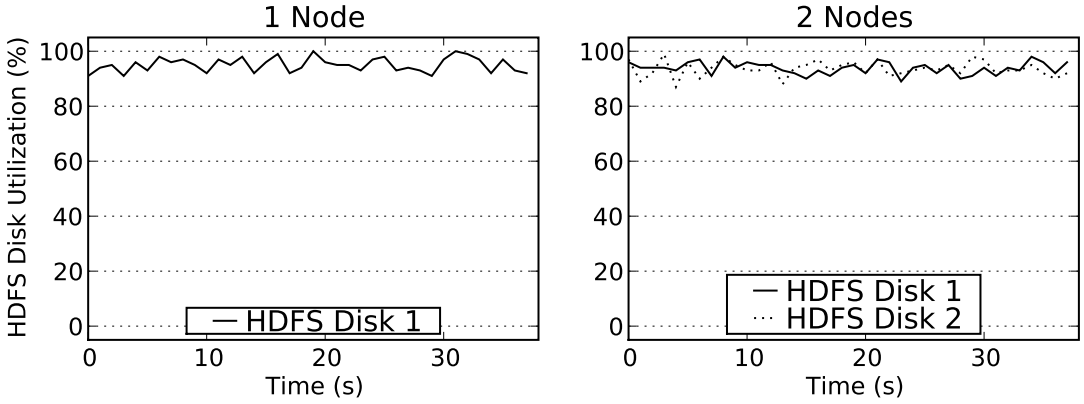
To identify the cause of the poor storage bandwidth, disk utilization was measured by a user-level utility. Utilization was captured at the local HDFS disk regardless of architecture, and thus is after the influence of the DataNode service or AoE server. Results for all three storage architectures are shown in Figure 7.3.



(a) Local Architecture – 1 and 2-Node Configurations



(b) Split Architecture – 1 and 2-Node Configurations



(c) AoE Architecture – 1 and 2-Node Configurations

Figure 7.3: HDFS Disk Utilization of Storage Architectures in 1 and 2-Node Configurations

Here, both the local and AoE architectures demonstrate consistent HDFS disk usage, keeping the disk busy at least 90% of the time. The Split architecture in the 1-disk configuration also kept the HDFS disk busy 98% of the time. But, the Split architecture in the 2-node configuration exhibits periodic behavior with long idle times, which result in a drive utilization of only 55-60%, and a significant performance degradation. Because the test application streams data continuously, the long periods of time where one disk is idle imply that the other disk must be handling two data streams at once, thus doing additional work and causing additional slowdowns from excessive seeks. Read bandwidth is worse than write bandwidth because reads suffer both from fragmentation (when the data was originally written), and congestion caused by an unbalanced cluster.

The cause of this behavior in the Split architecture is the lack of locality as seen by the NameNode. Each synthetic writer (one per client node) writes a single large file composed of many HDFS blocks. In the 1-node configuration, only a single file is created, whereas in the 2 node configuration, 2 files are written. Inspection of the specific HDFS block assignments for the files via the Hadoop web interface revealed that in the local and AoE architectures, each client is assigned HDFS blocks exclusively on the "local" disk. In the local architecture, the disk is genuinely local, whereas in the AoE configuration, the disk simply appears to be local, but is actually accessed across the network. For the purposes of the NameNode, that disk is still managed by a single DataNode, and is treated identically. In the Split

configuration, however, there is no apparent locality in the system. The pool of TaskTrackers and the pool of DataNodes are entirely disjoint.

Locality plays an important part in the logic the NameNode uses to optimize replica placement in the cluster. Specifically, a client contacts the NameNode to request a new HDFS block ID for a file and n target DataNodes on which to store block replicas. When choosing n replicas, the NameNode will assign the first replica to the DataNode co-located with the client. If that node is not available, a random DataNode will be selected. The second replica will be assigned to a DataNode in a different rack as the client, using Hadoop's built-in rack awareness framework. The third replica will be placed in the same rack as the first replica, and the forth and any further replicas will be assigned to random DataNodes in the cluster. The NameNode will return the list of assigned replicas to the client, who must contact them directly in order to write data. In practice, the client will contact the first DataNode and establish a replication pipeline through it to subsequent DataNodes.

Replica choices are always subject to the following availability rules which can veto a selection made according to the process just described. First, sufficient disk space must be available at the DataNode to store all pending HDFS blocks that have been assigned to it, but perhaps not written yet. If space is not available, a different replica must be chosen. Second, course-grained load balancing is applied based on the number of files being written. If a selected DataNode has more

than twice the number of current clients (sampled and reported periodically) as the average load of all DataNodes in the entire cluster, a different replica must be selected. Third, course-grained load balancing is applied to ensure that all racks have roughly the same number of replicas, defined as within 2 of the average. This rule typically takes effect only when there are more than three replicas for a specific block, and thus it has no impact in the default Hadoop configuration.

The impact of this design is clear in the experimental results. In the Split architecture with no DataNode locality, the NameNode is forced into a situation where the first replica is randomly assigned. The effect of this random assignment policy can be seen in Figure 7.3(b), where each disk is either oversubscribed (accessed by two clients at the same time), or under-subscribed (completely idle), depending on the random selection process. In the 2-client and 2-datanode test cluster, conflicts occurred roughly 50% of the time. This problem did not exist in the local or AoE architectures because the preference for using the local node tended to ensure that all disks were equally busy, all the time.

To mitigate this problem for the Split architecture, the NameNode target selection system was modified. Persistent assignments were introduced by caching targets and re-using the assignments for future blocks. This prevents clients from rapidly switching between nodes when randomly selecting a target, as was seen in the unmodified system. Further, the target allocation scheme was modified to first select a target in the local rack, then in the remote rack, and randomly thereafter.

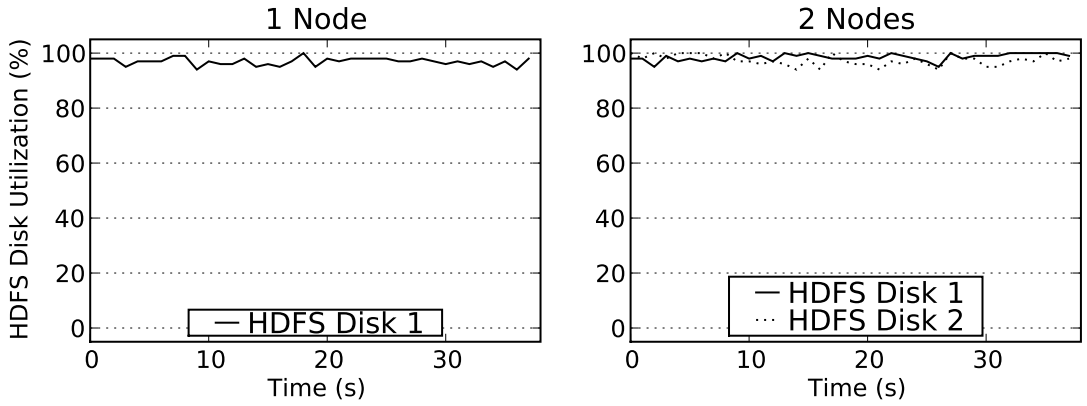


Figure 7.4: Disk Utilization of Split Storage Architecture with NameNode Modifications in 1 and 2-Node Configurations

Finally, a load balancing thread was added to periodically balance the number of active clients for each DataNode.

The improved performance of the Split architecture over time with these modifications is shown in Figure 7.4. Here, both HDFS disks are in use continuously. File tracing through the Hadoop web interface revealed that both clients are consistently using different DataNodes in a manner identical to the way clients use the same local node in the local storage architecture. With the NameNode improvements, the split architecture now shows a substantial performance improvement in aggregate bandwidth when going from a 1-node to 2-node configuration, as shown in Table 7.3. What was previously a 10% drop in write bandwidth is now a 92% improvement, and what was previously a 25% drop in read bandwidth is now a 91% improvement. Thus, the performance of the Split architecture is now competitive with the traditional Hadoop local storage architecture.

This chapter thus far has focused on providing persistent network-based stor-

Configuration	1 Writer	2 Writers	1 Reader	2 Readers
Local	67.3	145.2	70.1	142.6
Split-Modified	70.4	130.7	69.8	131.4
AoE	16.3	27.2	37.2	79.1

Table 7.3: Storage Bandwidth (MB/s) in 1 and 2-Node Configurations with NameNode Modifications

age for Hadoop independent of any virtualization framework. Now, the virtualization framework itself is evaluated to determine its performance impact on the storage architecture.

7.3 Performance in Eucalyptus Cloud Computing Framework

The new storage architecture for Hadoop combines local storage for temporary data with network-based storage for persistent HDFS data. Such an architecture is compatible with the Eucalyptus cloud computing framework previously discussed in Section 6.2, which uses a virtualization framework to provide isolation between applications. In such an environment, Hadoop temporary data can reside on local storage provided by the cloud environment using an architecture such as the one shown in Figure 6.2(a). Further, persistent data can reside on network-accessible storage nodes outside of the cloud environment. The native cloud network storage, shown in Figure 6.2(b), is not used for HDFS storage, as the Hadoop data transfer protocol functions more efficiently than AoE for this application.

Such a storage architecture depends heavily on the performance of the virtualized machine. High disk I/O bandwidth between the virtual machine and the

native disk is critical for scratch storage performance, and high network I/O bandwidth between the virtual machine and an external host is critical for persistent storage performance. Here, these two metrics are evaluated using the Eucalyptus cloud computing framework in order to determine its suitability. Given the research that has been invested in I/O virtualization in recent years, and the ease-of-installation that was promised by Eucalyptus, the hope was that performance would be suitable out-of-the-box.

7.3.1 Test Configuration

To test Eucalyptus, a simplified two-node cluster was used. This cluster is not the same as the one used for previous experiments. The software is different because Eucalyptus runs on Linux systems, not FreeBSD. Further, the hardware is different because Eucalyptus requires processor support for hardware virtualization extensions. In the test cluster, a front-end node with two network interfaces was connected to both the campus network and a private test network, and a back-end node was connected only to the private network. Both networks ran at gigabit speeds.

The front-end node was equipped with two AMD Opteron processors running at 2.4GHz with 4GB of RAM and a 500GB hard drive. It was configured to run the CLC, CC, EBS, and WS3 services as shown in Figure 6.1. The back-end node was equipped with two quad-core AMD Opteron processors running at 3.1GHz

with 16GB of RAM and a 500GB hard drive. These processors support the AMD-V virtualization extensions as required for KVM support in Linux. The back-end node was configured to run the NC service and all virtual machine images.

Two different software configurations were used:

Eucalyptus with KVM — In the first configuration, Eucalyptus with the KVM hypervisor was used. This is a default installation of Ubuntu Enterprise Cloud (UEC), which couples Eucalyptus 1.60 with Ubuntu 9.10 [82]. The key benefit of UEC is ease-of-installation — it took less than 30 minutes to install and configure the simple two-node system.

Eucalyptus with Xen — In the second configuration, Eucalyptus was used with the Xen hypervisor. Unfortunately, Ubuntu 9.10 is not compatible with Xen when used as the host domain (only as a guest domain). Thus, the CentOS 5.4 distribution was used instead because of its native compatibility with Xen 3.4.2. The guest VM image still used Ubuntu 9.10.

Two microbenchmarks were used inside the virtual machine to evaluate I/O performance. First, the simple *dd* utility was used to generate storage requests similar to those produced by Hadoop (large sequential writes and reads), but without the computation overhead of Java and the rest of the MapReduce framework. When using dd, 20GB tests were conducted using a 64kB block size. Second, the lightweight *netperf* utility was used to stress the virtual network with a minimum computation overhead.

To provide a performance baseline, the storage and network components were profiled with these utilities outside of the virtual machine. For storage, the Seagate Barracuda 7200.11 500GB hard drive (as profiled previously in Section 3.2) has a peak write and read bandwidth of approximately 111 and 108MB/s, respectively, assuming large block sizes (64kB+) and streaming sequential access patterns. For networking, the gigabit Ethernet network has a max application-level TCP throughput of 940Mb/s for both transmit and receive. In an ideal cloud computing system, this performance would be available to applications running inside the virtual environment.

7.3.2 Performance Evaluation

In testing, the storage and network performance significantly degraded under Eucalyptus with the KVM hypervisor and other default settings. Write bandwidth to the local disk is only 1.3 MB/s, a 98% reduction, while read bandwidth to local disk is 71.9 MB/s, a 38% reduction. Network bandwidth to the front-end node suffered too, achieving only 667 Mb/s transmitting and 431 Mb/s receiving, a 29% and 54% reduction from the non-virtualized performance, respectively.

To investigate the cause of the poor out-of-the-box storage performance, follow-up tests were conducted with a variety of non-default configurations. Several virtual machine monitors (VMMs) were used with Eucalyptus, including none (indicating that only the host domain was used for comparison purposes), KVM,

and Xen. The storage target was either a sparse file on local disk (by default), a fully-allocated file on local disk, or the raw disk mapped in its entirety into the guest. Several KVM I/O virtualization mechanisms were used, including a fully-virtualized SCSI driver (emulating a LSI Logic 53c895a controller) and a para-virtualized Virtio driver [16, 74]. Similarly, Xen used either a fully-virtualized SCSI driver or para-virtualized XVD driver. Performance results are reported in Table 7.4 for write bandwidth and Table 7.5 for read bandwidth.

Several metrics are reported for each configuration. First, the application-level bandwidth (as seen in the guest domain by the dd application) is provided. Next, several disk utilization metrics were measured in the host domain (not the guest domain) by the *iostat* utility to track disk access efficiency after the influence of the I/O virtualization mechanism. These metrics include `avgrq-sz`, the average disk request size measured in kB, `avgqu-sz`, the average queue depth measured in disk requests, and percent utilization, the percent of time that the disk had at least one request outstanding.

Several conclusions can be drawn from the expanded suite of test configurations. First, pre-allocating the backing file on local disk (instead of using a sparse file that grows as data is written) eliminates the abnormally low write bandwidth of 1.3 MB/s initial reported, boosting it to 62.6 MB/s. The tradeoff implicit in this change is the time required to initialize the file on disk, which can be amortized by long-running virtual machine instances. Second, using para-virtualized device

VMM	Driver	Bandwidth	Avgrq-sz	Avgqu-sz	% Util
None	N/A	111	512	140	100%
KVM(*)	SCSI/sparse file	1.3	15	0.9	90%
KVM	SCSI/full file	62.6	128	0.82	81%
KVM	SCSI/disk	71.5	128	0.57	64%
KVM	Virtio/full file	87.0	490	42	100%
KVM	Virtio/disk	110	512	60	100%
Xen(*)	SCSI/full file	58.4	498	142	100%
Xen	SCSI/disk	65.8	126	0.87	86%
Xen	XVD/disk	102	350	3.0	100%

Table 7.4: DD Write Bandwidth (MB/s) to Local Disk and Disk Access Pattern Measured at Host Domain. Entries marked (*) are Eucalyptus Default Configurations.

VMM	Driver	Bandwidth	Avgrq-sz	Avgqu-sz	% Util
None	N/A	108	256	0.94	96%
KVM(*)	SCSI/sparse file	71.9	225	1.1	96%
KVM	SCSI/full file	71.4	241	0.64	64%
KVM	SCSI/disk	70.5	256	0.7	68%
KVM	Virtio/full file	75.9	256	0.7	69%
KVM	Virtio/disk	76.2	256	0.5	57%
Xen(*)	SCSI/full file	83.1	121	1.6	99%
Xen	SCSI/disk	42.8	7	22.4	99%
Xen	XVD/disk	94.8	64	2.2	99%

Table 7.5: DD Read Bandwidth (MB/s) to Local Disk and Disk Access Pattern Measured at Host Domain. Entries marked (*) are Eucalyptus Default Configurations.

drivers (virtio and XVD) instead of fully-virtualized devices increases bandwidth in both KVM and Xen. Para-virtualized drivers are able to use the underlying disk efficiently, with both large requests and deep queues of pending requests. The tradeoff here is that this requires device support in the guest operating system, although such support is nearly universal today. Third, mapping the entire local disk into the guest domain, instead of mapping a file on local into the guest domain, improves performance further. The tradeoff here is that the disk can no

longer be shared between virtual machines without partitioning the device. By combining these techniques together, local disk bandwidth from the guest domain is increased to within 80-100% of the non-virtualized bandwidth, depending on the hypervisor used. Thus, local storage is a viable platform for Hadoop scratch storage assuming that the virtual environment is properly configured before use.

Like storage bandwidth, network bandwidth also was improved by switching to para-virtualized drivers instead of fully-virtualized drivers. Using the virtio driver in KVM yielded a transmit bandwidth of 888.7 Mb/s and a receive bandwidth of 671.6 Mb/s, which is a 5% and 28% drop over the ideal performance, respectively. Xen did slightly better, generating 940 Mb/s and 803 MB/s for transmit and receiving from the guest domain, which is a 0% and 14% degradation, respectively. Other work has shown that Xen, properly configured, is able to saturate a 10Gb/s Ethernet link from a guest domain [69]. This is an active topic of research that is receiving significant attention from the virtualization community. Thus, network storage is a viable platform for Hadoop persistent storage assuming that the virtual environment is properly configured before use. Next, virtualization is combined with remote storage to provide a complete architecture for Hadoop execution in a datacenter shared with other applications.

7.4 Putting it All Together

In this final section, a complete vision is presented for implementing the proposed remote storage architecture for MapReduce and Hadoop in a datacenter running a virtualization framework such as Eucalyptus. This design exploits the high bandwidth of datacenter switches and co-locates computation and storage inside the same rack (connected to the same switch), instead of co-locating computation and storage in the same node as in the traditional local storage architecture. A generic rack in the datacenter is shown in Figure 7.5. Here, all nodes in the rack are connected to the same Ethernet switch with full bisection bandwidth. Uplink ports from the switch (not shown) interconnect racks, allowing this design to be generalized to a larger scale if desired. For simplicity, operating system details are not shown in the figure. But, all nodes have an operating system, and that OS is stored on a local disk or flash memory. For example, the master node metadata disk could also store the host OS running the JobTracker and NameNode services.

There are three types of nodes in the cluster: master, compute, and storage. Both the master and storage nodes are specialized nodes exclusively for Hadoop-specific purposes. In contrast, the compute node is a standard cloud computing node that can be shared and re-used for other non-MapReduce applications as required.

Master node — The master node runs the JobTracker and NameNode services. In a small Hadoop installation, a single master node could run both services, and

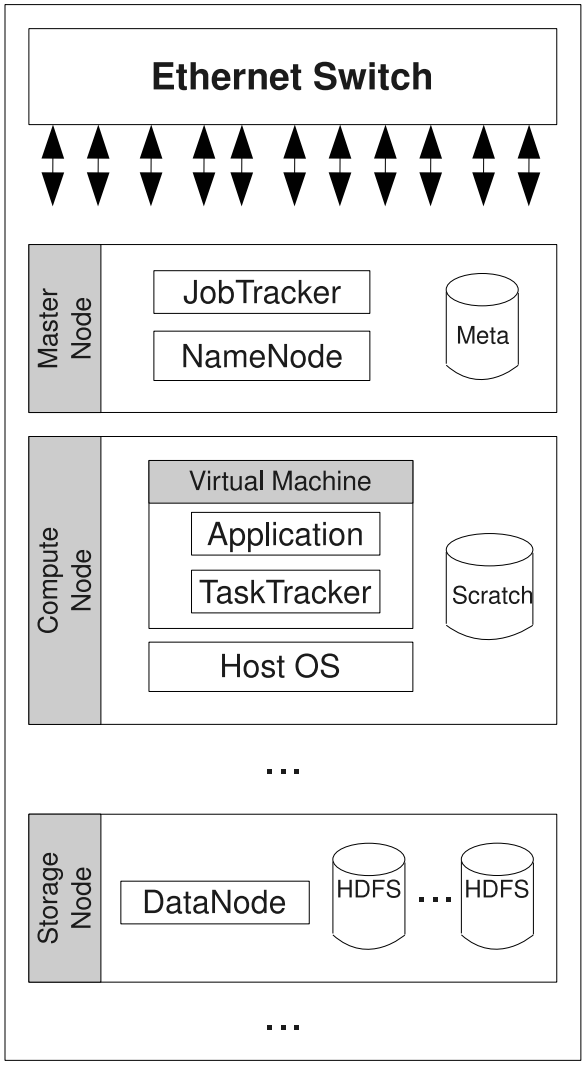


Figure 7.5: Rack View of Remote Storage Architecture for Hadoop

in a larger Hadoop installation two master nodes could be used; one for the JobTracker, and the other for the NameNode service. The master node is not virtualized like other nodes in the datacenter, and its persistent metadata is stored on a locally-attached disk. The services running on the master node are more latency sensitive than MapReduce applications accessing HDFS block storage.

Compute node — Compute nodes run MapReduce tasks under the control of

a TaskTracker service. There are many compute nodes placed in each datacenter rack. Each compute node is virtualized using a cloud computing framework such as Eucalyptus, and thus these nodes can be allocated and deallocated on demand in the datacenter based on application requirements. The local storage architecture, previously shown in Figure 6.2(a), allows the virtual machine access to local storage that is well suited to temporary or scratch data produced as part of the MapReduce computation process, such as intermediary key/value pairs which are not saved in persistent HDFS storage. After the MapReduce computation has ended, compute nodes can be re-used for other purposes by the cloud controller. The temporary data stored to local disk can be deleted and the storage re-used for other purposes. Later, when MapReduce computation is started up again, the cloud controller should try to allocate compute nodes in the same rack as the non-virtualized storage nodes whenever possible, thereby preserving some locality in this architecture.

Storage Node — Storage nodes provide persistent storage for HDFS data under the control of a DataNode daemon. There are many storage nodes placed in each rack, and each node contains at least one HDFS disk. Storage nodes are not virtualized. The HDFS data stored here is persistent regardless of whether or not MapReduce applications are currently running. As such, these nodes do not benefit from management by the virtualization / cloud computing framework, and would perform better by avoiding the overhead of virtualization. The DataN-

ode daemon can either run continuously, or the cloud framework can be configured to start/stop DataNode instances whenever the MapReduce images are started/stopped on the compute nodes. For energy efficiency, storage nodes could be powered down when no MapReduce computations are active. The number of HDFS disks placed in a storage node could vary depending on the physical design of the rack and rackmount cases, the processing resources of the storage node (more disks require more processing resources), and the network bandwidth to the datacenter switch. A 10 Gb/s network link could support more disks than a gigabit Ethernet link.

The ratio between compute nodes and storage nodes is flexible based on application requirements and the number of disks placed in each storage node. It can be changed during cluster design, and even during cluster operation if a few network ports and space in the rack is left open for future expansion. This flexibility is one of the key advantages of a remote storage architecture.

If desired, both the storage and compute nodes could be shared with other concurrent applications. Sharing of the storage node is possible because the DataNode daemon uses the native filesystem to store HDFS block data, and thus the disk could be used by other native applications. (Whether sharing is *likely* is a different question, however. HDFS data may consume most if not all of the disk space on the storage node, leaving no available resources left for other uses.) Sharing of the compute node is also possible because of the cloud computing framework.

Additional virtual machines could be started and assigned to other CPU cores. The cloud storage architecture provides a standardized method to provision each VM with independent local storage. Obviously, sharing these nodes entails performance tradeoffs, particularly with regards to finite storage bandwidth.

The master node could be a standard virtualized node, if desired. Unlike the compute node, however, the master node could not use the local storage resources provided by the cloud framework. The master node needs to store persistent non-HDFS data such as the filesystem namespace (managed by the NameNode) even when MapReduce computation is not active. Thus, the only suitable storage location provided by the cloud environment is the remote AoE-based storage shown in Figure 6.2(b). The benefit of running the master node in a virtualized environment is that it allows that node to be re-used for other application purposes when MapReduce computation is not active. The drawback is that the network-based storage provided by the cloud environment has higher latency than local storage, and the NameNode and JobTracker running on the master node are more latency sensitive than MapReduce applications.

The design of the storage nodes in this architecture can vary widely depending on technology considerations. For instance, storage nodes could be lightweight, low-powered devices consisting of a disk, embedded processor, and gigabit network interface. This is similar to the network-attached secure disks concept discussed in Section 5.4, where the standard disk controller also contains a network

interface, allowing a disk to be directly attached to the network. As a proof of concept, the DataNode daemon was tested on an 8 watt Atom 330 processor and achieved equivalent network storage bandwidth with over 50% processor idle time. Or, depending on technology considerations like network speed, it may be more effective to deploy a smaller number of large storage nodes with multiple disks, a high-powered processor, and a single ten-gigabit network interface. Regardless of the exact realization of the storage node, the interface provided (that of the DataNode daemon) would stay the same.

This new storage architecture requires cooperation from the cluster scheduler to operate efficiently. MapReduce computation should be executed on compute nodes located in the same rack as the persistent storage nodes. Otherwise, data in the hierarchical network will be transferred over cross-switch links, increasing the potential for network congestion. The cluster-wide node scheduler (*e.g.,* as provided in Eucalyptus) needs to be modified to take the location of storage nodes into account when assigning virtual machine images to specific hosts. To ensure good MapReduce performance, it may be desirable or necessary to migrate other applications away from racks containing persistent storage nodes, in order to make room for MapReduce computation and to enable its efficient operation.

CHAPTER 8

Conclusions

This thesis was initially motivated by debate in academic and industrial circles regarding the best programming model for data-intensive computing. Two leading contenders include parallel databases and MapReduce, each with their own strengths and weaknesses [37, 67, 77]. The MapReduce model has been demonstrated by Google to have wide applicability to a large spectrum of real-world programs. The open-source Hadoop implementation of MapReduce has allowed this model to spread to other well-known Internet service providers and beyond. But, Hadoop has been called into question recently, as published research shows its performance lagging by 2-3 times when compared with parallel databases [67].

To close this performance gap, the first part of this thesis focused on a previously neglected portion of the Hadoop MapReduce framework: the storage system. Data-intensive computing applications are often limited by the available storage bandwidth. Unfortunately, the performance impact of the Hadoop Distributed File System (HDFS) is hidden from Hadoop users. While Hadoop provides built-in functionality to profile Map and Reduce task execution, there are no

built-in tools to profile the framework itself, allowing performance bottlenecks to remain hidden. User-space and custom kernel instrumentation was used to break the black-box abstraction of HDFS and observe the interactions between Hadoop and storage.

As shown in this thesis, these black-box framework components can have a significant impact on the overall performance of a MapReduce framework. Many performance bottlenecks are not directly attributable to user-level application code as previously thought, but rather are caused by the task scheduler and distributed filesystem underlying all Hadoop applications. For example, delays in the task scheduler result in compute nodes waiting for new tasks, leaving the disk to sit idle for significant periods. A variety of techniques were applied to this problem to reduce the task scheduling latency and frequency at which new tasks need to be scheduled, thereby increasing disk utilization to near 100%.

The poor performance of HDFS goes beyond scheduling bottlenecks. A large part of the performance gap between MapReduce and parallel databases can be attributed to challenges in maintaining Hadoop portability across different operating systems and filesystems, each with their own unique performance characteristics and expectations. For example, disk scheduling and filesystem allocation algorithms are frequently designed in native operating systems for general-purpose workloads, and not optimized for data-intensive computing access patterns. Hadoop, running in Java, has no way to impact the behavior of these under-

lying systems. Fortunately, HDFS performance under concurrent workloads was significantly improved through the use of HDFS-level I/O scheduling while preserving portability. Further improvements by reducing fragmentation and cache overhead are also possible, at the expense of reducing portability. However, maintaining Hadoop portability whenever possible will simplify development and benefit users by reducing installation complexity.

Optimizing HDFS will boost the overall efficiency and performance of MapReduce applications in Hadoop. While this may or may not change the ultimate conclusions of the MapReduce versus parallel database debate, it will certainly allow a fairer comparison of the actual programming models. Further, greater efficiencies can reduce cluster power and cooling costs by reducing the number of computers required to accomplish a fixed quantity of work.

In addition to improving the performance of MapReduce computation, this thesis also focused on improving its flexibility. MapReduce and Hadoop were designed (by Google, Yahoo, and others) to marshal all the storage and computation resources of a dedicated cluster computer. Unfortunately, such a design limits this programming model to only the largest users with the financial resources and application demand to justify deployment. Smaller users could benefit from the MapReduce programming model too, but need to run it on a cluster computer shared with other applications through the use of virtualization technologies.

The traditional Hadoop storage architecture tightly couples storage and com-

putation resources together in the same node. This is due to a design philosophy that it is better to move the computation to the data, than to move the data to the computation. Unfortunately, this architecture is unsuitable for use in a virtualized environment. In this thesis, a new architecture for persistent network-based HDFS storage is proposed. This new design breaks the tight coupling found in the traditional architecture in favor of a new model that co-locates storage and computation at the same network switch, not in the same node. This is made possible by exploiting the high bandwidth and low latency of modern datacenter network switches. Such an architectural change greatly increases the flexibility of the cluster, and offers advantages in terms of resource provisioning, load balancing, fault tolerance, and power management. The new remote architecture proposed here was designed with virtualization in mind, thereby increasing the flexibility of MapReduce and encouraging the spread of this parallel computing paradigm.

8.1 Future Work

After contributing to the storage architecture of MapReduce and Hadoop, a wide variety of interesting projects remain as future work. With regards to the traditional local storage architecture, further improvements could be made in the areas of task scheduling and startup. For example, task prefetching could be implemented, or JVM instances started up in parallel with requesting new tasks. Both methods could reduce scheduling latency with fewer tradeoffs than the mecha-

nisms evaluated to date.

The use of Java as the implementation language for Hadoop could be re-visited as future work, regardless of the storage architecture employed. The benefit of Java for Hadoop is portability, specifically in simplifying the installation process by providing a common experience across multiple platforms and minimizing the use of third party libraries. The cost of Java is partially in overhead, but more significantly in loss of feature support. Testing with synthetic Java programs shows that Java code is able to achieve full local disk bandwidth and full network bandwidth, at the expense of a slight increase (less than 3%) in processor overhead compared with native programs written in C. Considering that data-intensive computing applications are often storage bound, not processor bound, this extra overhead is unlikely to pose a significant problem. The bigger drawback with Java is its least-common-denominator design. In the Java language, a feature is not implemented unless it can be provided by the Java Virtual Machine running on all supported platforms. The tradeoff here is that Java is unable to support platform-specific optimizations. Hadoop could benefit, for example, by pre-allocating space for an entire HDFS block to reduce fragmentation. Ideally, this would be conditionally enabled based on platform support (*i.e.*, the ext4 or XFS filesystem), but Java does not provide any mechanism to do so beyond the use of the Java Native Interface or other ad-hoc, inconvenient methods. Instead of using Java, it is possible to imagine a new HDFS framework where the DataNode service is written in C and

takes advantage of a library such as the Apache Portable Runtime to benefit from platform-specific optimizations like block pre-allocation. A C-language implementation could also employ O_DIRECT to bypass the OS page cache and transfer data directly into user-space buffers, something that is not possible in Java and would reduce processor overhead.

The new persistent network storage architecture for HDFS in a virtualized datacenter motivates further research into scheduling algorithms. In this new architecture, MapReduce computation should be executed on compute nodes located in the same rack as the persistent storage nodes. Otherwise, data in the hierarchical network will be transferred over cross-switch links, increasing the potential for network congestion. The cluster-wide node scheduler (for example, in Eucalyptus) needs to be modified to take the location of storage nodes into account when assigning virtual machine images to specific hosts. To ensure good MapReduce performance, it may be desirable or necessary to migrate other applications away from racks containing persistent storage nodes, in order to make room for MapReduce computation.

Persistent network storage for Hadoop has many benefits related to design flexibility that could also be investigated as future work. First, rack-level load balancing could be evaluated as a way to reduce cluster provisioning cost. Racks could be provisioned for the average aggregate I/O demand per rack, rather than the peak I/O demand per node. Compute nodes can consume more or less I/O resources on

demand. Second, the benefits related to fault isolation could be more clearly identified and evaluated. In a network storage architecture, a failure in a compute node only affects computation resources, and a failure in a storage node only affects storage resources. This is in contrast to a failed node in the traditional local storage architecture, which impacts both computation and storage resources. Third, fine-grained power management techniques that benefit from stateless compute nodes could be investigated. For example, a cluster could be built with a mix of compute nodes, some employing high-power/high-performance processors, and others employing low-power/low-performance processors. The active set of compute nodes could be dynamically varied depending on application requirements for greater energy efficiency.

Bibliography

[1] *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley Publishing, Inc., 2003.

[2] Hadoop. `http://hadoop.apache.org`, 2008.

[3] HBase: Hadoop database. `http://hadoop.apache.org/hbase/`, 2008.

[4] HDFS (hadoop distributed file system) architecture. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`, 2008.

[5] Netgear prosafe 48-port gigabit L3 managed stackable switch data sheet - GSM7352S, 2008.

[6] Scaling hadoop to 4000 nodes at Yahoo! `http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html`, 2008.

[7] 2009 linux kernel summit: How google uses linux. `http://lwn.net/Articles/357658/`, 2009.

[8] AoE (ATA over ethernet). `http://support.coraid.com/documents/AoEr11.txt`, 2009.

[9] The diverse and exploding digital universe. http://www.emc.com/digital_universe, 2009.

[10] Eucalyptus open-source cloud computing infrastructure - an overview. Technical report, Eucalyptus, Inc., August 2009.

[11] HDTach. http://www.simplisoftware.com, 2009.

[12] Amazon web services. http://aws.amazon.com, January 2010.

[13] Eucalyptus community. http://open.eucalytpus.com, January 2010.

[14] Hadoop quick start guide. http://hadoop.apache.org/common/docs/current/quickstart.html, March 2010.

[15] KVM - kernel based virtual machine. http://www.linux-kvm.org, 2010.

[16] Virtio para-virtualized drivers. http://wiki.libvirt.org/page/Virtio, 2010.

[17] Xen - Xen hypervisor, the powerful open source industry standard for virtualization. http://www.xen.org, 2010.

[18] R.K. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 113–124, Dec 1990.

[19] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth*

international conference on Architectural support for programming languages and operating systems, pages 81–91, New York, NY, USA, 1998. ACM.

[20] N. Ali and M. Lauria. SEMPLAR: high-performance remote parallel I/O over SRB. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 366–373, Washington, DC, USA, 2005. IEEE Computer Society.

[21] N. Ali and M. Lauria. Improving the performance of remote I/O using asynchronous primitives. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 218–228, 0-0 2006.

[22] Darrell C. Anderson, Jeffery S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. In *OSDI'00: Proceedings of the 4th Symposium on Operating System Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2000. USENIX Association.

[23] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, New York, NY, USA, 1995. ACM.

[24] Keith Bell, Andrew Chien, and Mario Lauria. A high-performance cluster storage server. In *HPDC '02: Proceedings of the 11th IEEE International Sympo-*

sium on High Performance Distributed Computing, page 311, Washington, DC, USA, 2002. IEEE Computer Society.

[25] Randal E. Bryant. Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon University, May 2007.

[26] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. In *35th International Conference on Very Large Data Bases (VLDB)*, 2009.

[27] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 217–228, New York, NY, USA, 2009. ACM.

[28] John A. Chandy. A generalized replica placement strategy to optimize latency in a wide area distributed storage system. In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 49–54, New York, NY, USA, 2008. ACM.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06:*

Proceedings of the 7th symposium on Operating systems design and implementation, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[30] R.-I. Chang, W.-K. Shih, and R.-C. Chang. Deadline-modification-scan with maximum-scannable-groups for multimedia real-time disk scheduling. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 40, Washington, DC, USA, 1998. IEEE Computer Society.

[31] Steve C. Chiu, Wei keng Liao, Alok N. Choudhary, and Mahmut T. Kandemir. Processor-embedded distributed smart disks for I/O-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput.*, 64(3):427–446, 2004.

[32] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[33] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.

[34] George Davidson, Kevin Boyack, Ron Zacharski, Stephen Helmreich, and Jim Cowie. Data-centric computing with the netezza architecture. Technical Re-

port SAND2006-3640, Sandia National Laboratories, April 2006.

[35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[38] Kenton DeLathouwer, Alan Y Lee, and Punit Shah. Improve database performance on file system containers in IBM DB2 universal database v8.2 using concurrent I/O on AIX. Technical report, IBM, 2004.

[39] Yuhui Deng. RISC: A resilient interconnection network for scalable cluster storage systems. *J. Syst. Archit.*, 54(1-2):70–80, 2008.

[40] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[41] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. A decentralized algorithm for erasure-coded virtual disks. In *DSN*

'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, page 125, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[43] Phillip B. Gibbons. Data rich computing: Where it's at. In *Data-Intensive Computing Symposium*, March 2008.

[44] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 92–103, New York, NY, USA, 1998. ACM.

[45] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–284, New York, NY, USA, 1997. ACM.

[46] Robert Grossman and Yunhong Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 920–927, New York, NY, USA, 2008. ACM.

[47] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[48] Steven R. Hetzler. The storage chasm: Implications for the future of HDD and solid state storage. Technical report, http://www.idema.org/, 2008.

[49] Andy D. Hospodor. Interconnection architectures for petabyte-scale high-performance storage systems. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 273–281, 2004.

[50] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[51] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP '01:*

Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, pages 117–130, New York, NY, USA, 2001. ACM.

[52] Shenze Chen John, John A. Stankovic, James F. Kurose, and Don Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Journal of Real-Time Systems*, 3:307–336, 1991.

[53] Sujatha Kashyap, Bret Olszewski, and Richard Hendrickson. Improving database performance with AIX concurrent I/O: A case study with oracle9i database on AIX 5L version 5.2. Technical report, IBM, 2003.

[54] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, 1998.

[55] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM.

[56] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, New York, NY, USA, 1996. ACM.

[57] Christopher R. Lumb and Richard Golding. D-SPTF: decentralized request distribution in brick-based storage systems. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 37–47, New York, NY, USA, 2004. ACM.

[58] John Maccormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *Trans. Storage*, 3(4):1–43, 2008.

[59] Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 95, Washington, DC, USA, 2001. IEEE Computer Society.

[60] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158, New York, NY, USA, 2009. ACM.

[62] Baila Ndiaye, Xumin Nie, Umesh Pathak, and Margaret Susairaj. A quantitative comparison between raw devices and file systems for implementing oracle databases. Technical report, Oracle / Hewlett-Packard, 2004.

[63] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[64] Owen O'Malley and Arun C. Murthy. Winning a 60 second dash with a yellow elephant. Technical report, Yahoo!, 2009.

[65] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 205–216, New York, NY, USA, 1998. ACM.

[66] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[67] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches

to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 165–178, New York, NY, USA, 2009. ACM.

[68] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[69] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10 Gb/s using safe and transparent network interface virtualization. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 61–70, New York, NY, USA, 2009. ACM.

[70] A. L. N. Reddy, Jim Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 1(1):37–59, 2005.

[71] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM.

[72] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[73] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM.

[74] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[75] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 48–58, New York, NY, USA, 2004. ACM.

[76] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 44–55, New York, NY, USA, 1998. ACM.

[77] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[78] Wittawat Tantisiriroj, Swapnil Patil, and Garth Gibson. Data-intensive file systems for internet services: A rose by any other name. Technical report, Carnegie Mellon University, 2008.

[79] Vijay Vasudevan, Jason Franklin, David Andersen, Amar Phanishayee, Lawrence Tan, Michael Kaminsky, and Iulian Moraru. FAWNdamentally Power-efficient Clusters. In *HotOS '09: Proceedings of the 12th Workshop on Hot Topics in Operating Systems*. USENIX, 2009.

[80] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, 2006.

[81] Jun Wang, Rui Min, Yingwu Zhu, and Yiming Hu. UCFS - a novel user-space, high performance, customized file system for web proxy servers. *IEEE Transactions on Computers*, 51(9):1056–1073, Sep 2002.

[82] Simon Wardley, Etienne Goyer, and Nick Barcet. Ubuntu enterprise cloud architecture. Technical report, Canonical, August 2009.

[83] C. Wu and R. Burns. Improving I/O performance of clustered storage systems by adaptive request distribution. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 207–217, 0-0 2006.

[84] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 723–734, 2007.