The Hadoop Distributed Filesystem: Balancing Portability and Performance

Jeffrey Shafer, Scott Rixner, and Alan L. Cox
Rice University
Houston, TX
Email: {shafer, rixner, alc}@rice.edu

***Abstract*—Hadoop is a popular open-source implementation of MapReduce for the analysis of large datasets. 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. This paper analyzes the performance of HDFS and uncovers several performance issues. First, *architectural bottlenecks* exist in the Hadoop implementation that result in inefficient HDFS usage due to delays in scheduling new MapReduce tasks. Second, *portability limitations* prevent the Java implementation from exploiting features of the native platform. Third, HDFS implicitly makes *portability assumptions* about how the native platform manages storage resources, even though native filesystems and I/O schedulers vary widely in design and behavior. This paper investigates the root causes of these performance bottlenecks in order to evaluate tradeoffs between portability and performance in the Hadoop distributed filesystem.**

I. Introduction

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 [2]. The MapReduce programming model has emerged as a scalable way to perform data-intensive computations on commodity cluster computers [8], [9]. The success of MapReduce has inspired the creation of Hadoop, a popular open-source implementation. Written in Java for cross-platform portability, 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.

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 in experiments when compared to similar programs using parallel databases [18], [22]. However, this work did not perform the profiling necessary to distinguish the fundamental performance of the MapReduce programming model from a specific implementation. We find that it is actually the implementation of the Hadoop storage system that degrades performance significantly.

This paper is the first to analyze the interactions between Hadoop and storage. We describe how the user-level Hadoop filesystem, instead of efficiently capturing the full performance potential of the underlying cluster hardware, actually degrades application performance significantly. The specific bottlenecks in HDFS can be classified into three categories:

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. Instead of using the disk in a streaming manner, the access pattern is periodic. Further, even when tasks are available for computation, the HDFS client code, particularly for file reads, serializes computation and I/O instead of decoupling and pipelining those operations. Data prefetching is not employed to improve performance, even though the typical MapReduce streaming access pattern is highly predictable.

Portability Limitations — 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.

Portability Assumptions — 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 underlying software layers, specifically the OS I/O scheduler and native filesystem allocation algorithm.

Here, we quantify the impact and significance of these HDFS bottlenecks. Further, we explore potential solutions and examine how they impact portability and performance. These solutions include improved I/O scheduling, adding pipelining and prefetching to both task scheduling and HDFS clients, preallocating file space on disk, and modifying or eliminating the local filesystem, among other methods.

MapReduce systems such as Hadoop are used in large-scale deployments. Eliminating HDFS bottlenecks will not only boost application performance, but also improve overall cluster efficiency, thereby reducing power and cooling costs and allowing more computation to be accomplished with the same number of cluster nodes.

In this paper, Section II describes Hadoop and its distributed

filesystem, while Section III characterizes its current performance. Section IV discusses potential performance improvements to Hadoop and their portability implications. Section V discusses related work, and Section VI concludes this paper.

II. Background

Hadoop [3] is an open source framework that implements the MapReduce parallel programming model [8]. Hadoop 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 — both components are written in Java and only require commodity hardware.

A. MapReduce Engine

In the MapReduce model, computation is divided into a *map* function and a *reduce* function. The map function takes a key/value pair and produces one or more intermediate key/value pairs. The reduce function 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, also exposing significant parallelism.

In Hadoop, a centralized *JobTracker* service is responsible for splitting the input data into pieces for processing by independent map and reduce tasks, scheduling each task on a cluster node for execution, and recovering from failures by re-running tasks. On each node, a *TaskTracker* service runs MapReduce tasks and periodically contacts the JobTracker to report task completions and request new tasks. By default, when a new task is received, a new JVM instance will be spawned to execute it.

B. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) provides global access to files in the cluster [4], [23]. For maximum portability, 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 [11]. First, the data files accessed are large, typically tens to hundreds of gigabytes in size. Second, these files are manipulated via 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 (based on Hadoop rack-awareness, but optimally the same node) and requests a specific block ID. That HDFS block is returned over the same connection, and the data delivered to the application.

To write data to HDFS, client applications see the HDFS file as a standard output stream. Internally, however, stream data is first fragmented into HDFS-sized blocks (64MB) and then smaller packets (64kB) by the client thread. Each packet is enqueued into a FIFO that can hold up to 5MB of data, thus decoupling the application 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

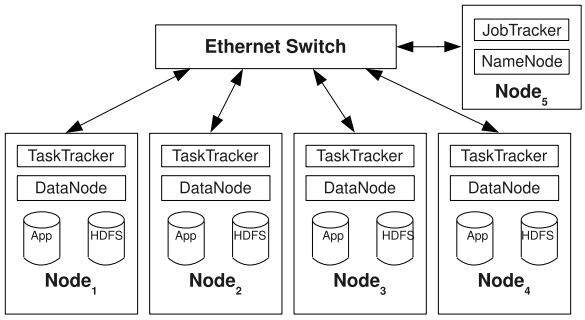


Fig. 1. Cluster Setup

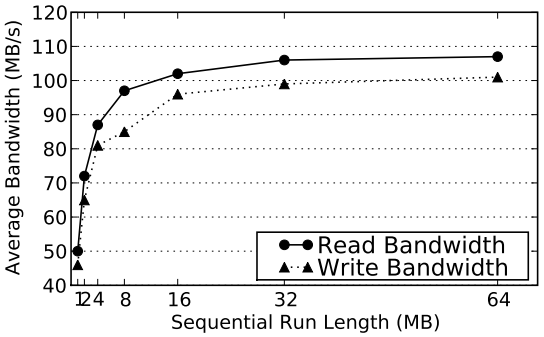


Fig. 2. Raw Hard Drive Read and Write Bandwidth from AIO Test With Random Seek Every *n* Megabytes

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.

C. HDFS Replication

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 block 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.

III. Performance Characterization

In this section, the Hadoop distributed filesystem is evaluated in order to identify bottlenecks that degrade application performance.

A. Experimental Setup

For performance characterization, a 5-node Hadoop cluster was configured, as shown in Figure 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, 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. Unless otherwise stated, Hadoop replication was disabled.

To characterize the Hadoop framework, a variety of test applications were installed as shown in Table I. 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. Hadoop is still a young platform, and the few complex applications used in industry are proprietary and thus unavailable. 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.

B. Raw Disk Performance

To place an upper bound on Hadoop performance, the raw bandwidth of the commodity hard drive used in the cluster was measured. To quantify the performance impact of seeks, the AIO program (running on a raw disk, not inside Hadoop) 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 2, 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 reasonable and, assuming that the filesystem maintains file contiguity, should enable high disk bandwidth.

C. Software Architectural Bottlenecks

Hadoop application performance suffers due to architectural bottlenecks in the way that applications use the Hadoop filesystem. 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 computation requirements do not

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 search of text data for rare string
AIO-Wr	Synthetic Write	10GB / node	Native C Program - Asynchronous I/O
AIO-Rd	Synthetic Read	10GB / node	Native C program - Asynchronous I/O

TABLE I
APPLICATION TEST SUITE

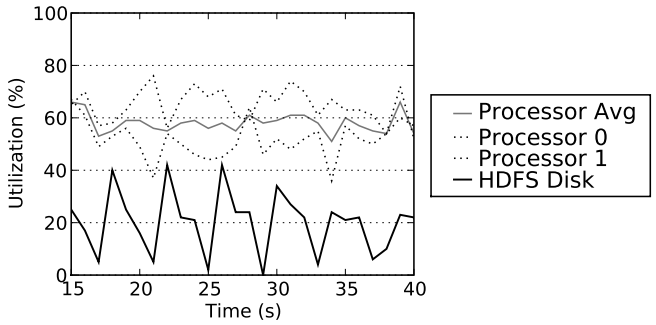


Fig. 3. Simple Search Processor and Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

achieve this ideal operating mode. Instead, they utilize the disk in a periodic fashion, decreasing performance.

The behavior of the disk and processor utilization over time for the simple search benchmark is shown in Figure 3. Disk utilization was measured as the percentage of time that the disk had at least one I/O request outstanding. This profiling did not measure the relative efficiency of disk accesses (which is influenced by excessive seeks and request size), but simply examined whether or not the disk was kept sufficiently busy with outstanding service requests. Here, the system is not accessing the 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, not shown here.

The overall system impact of this periodic behavior is shown in Figure 4, 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 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 [17], [18] — 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 III-E.

To more directly attack the performance bottleneck, Hadoop can be configured to re-use the same JVM for multiple tasks instead of starting a new JVM each time. In the search test, this increased performance by 27%, although disk utilization was still below 50%. Further, the amount of work done by each map task can be adjusted. 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%.

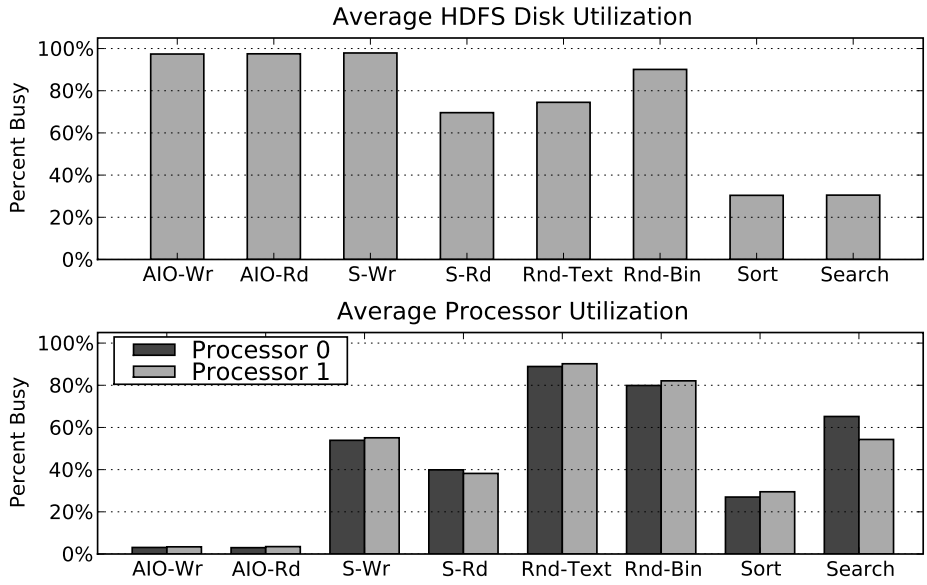


Fig. 4. Average Processor and HDFS Disk Utilization (% of Time Disk Had 1 or More Outstanding Requests)

These quick tests show that HDFS — the focus of this paper — is not the cause of this performance bottleneck. Rather, further work in the rest of the Hadoop framework is needed. Solutions such as pipelining and prefetching tasks from the JobTracker in advance may help hide scheduling latency.

Even when tasks are available for processing and each task is operating over multiple 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 Section II, 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 request.

D. 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. These are referred to as *Portability Limitations*.

An example of the total overhead incurred is shown in Figure 4. 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 Section II, 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, while the filesystem page cache consumes both processor and memory resources to manage.

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. 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. Kernel callgraph profiling was used to attribute overhead to specific OS functions.

As shown in Table II, 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,

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 II
PROCESSOR OVERHEAD OF DISK AS RAW DEVICE VERSUS DISK WITH FILESYSTEM AND PAGE CACHE (FS+CACHE)

and higher if additional or faster disks were added to the cluster.

E. 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.

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 demonstrate poor scheduling by the operating system, 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 5(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 2.

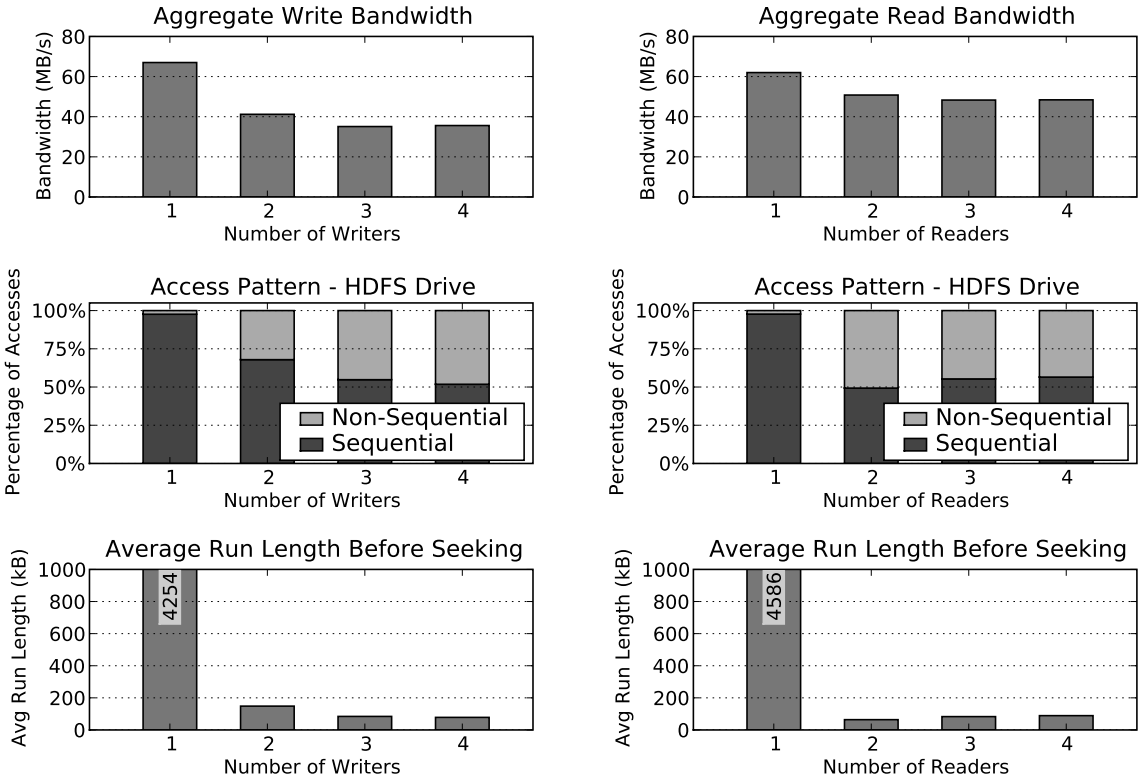
A similar performance issue occurs when HDFS is sharing the disk between concurrent readers. To demonstrate this, the same synthetic test program was used. First, a single writer was used per node to write 4 separate 10GB HDFS files. A single writer process creates data that is highly contiguous on disk, 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 5(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. If the system had used a more sophisticated scheduler that minimizes seeks, such as the Anticipatory Scheduler, this problem may have been masked, and the limitations of the portable HDFS implementation hidden. The Anticipatory Scheduler attempts to reduce seeks by waiting a short period after each request to see if further sequential requests are forthcoming [13]. If they are, the requests can be serviced without extra seeks; if not, the disk seeks to service a different client.

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

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



(a) Concurrent Writers

(b) Concurrent Readers

Fig. 5. Impact of Concurrent Synthetic Writers and Readers on HDFS Drive Access Patterns

by the general-purpose filesystem when making disk allocation decisions.

To measure 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 should be able to access it with a minimum of seeks; otherwise, the file must be fragmented on disk.

The results from this experiment are shown in Figure 6. 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 approximately 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 6 for the fragmentation test are almost twice as long as the multiple writers test shown in Figure 5(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.

F. 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 experimentation, 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

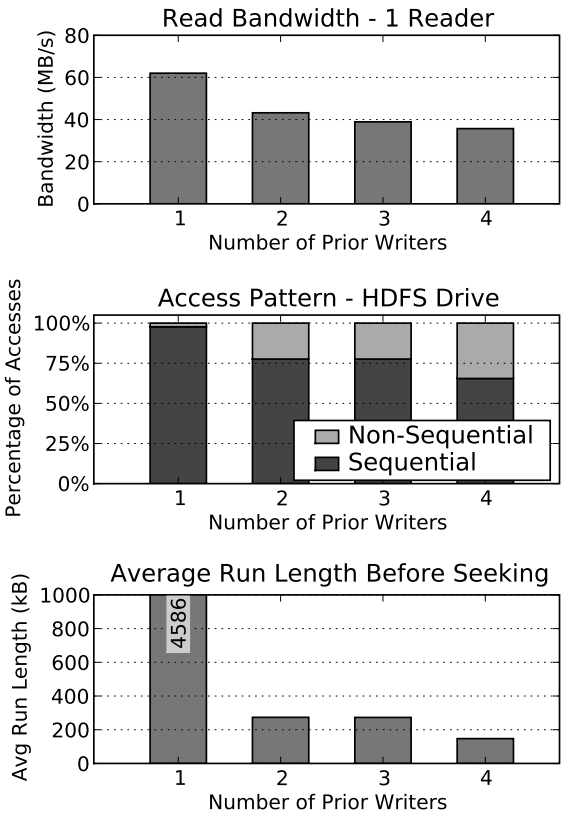


Fig. 6. One Hadoop Synthetic Reader Program Accessing Data From One Synthetic Writer. (Data was Previously Generated With 1-4 Concurrent Writers)

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 Section III-C.

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 III show that

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

TABLE III
DISK ACCESS CHARACTERISTICS FOR SYNTHETIC WRITE AND READ APPLICATIONS WITH REPLICATION ENABLED

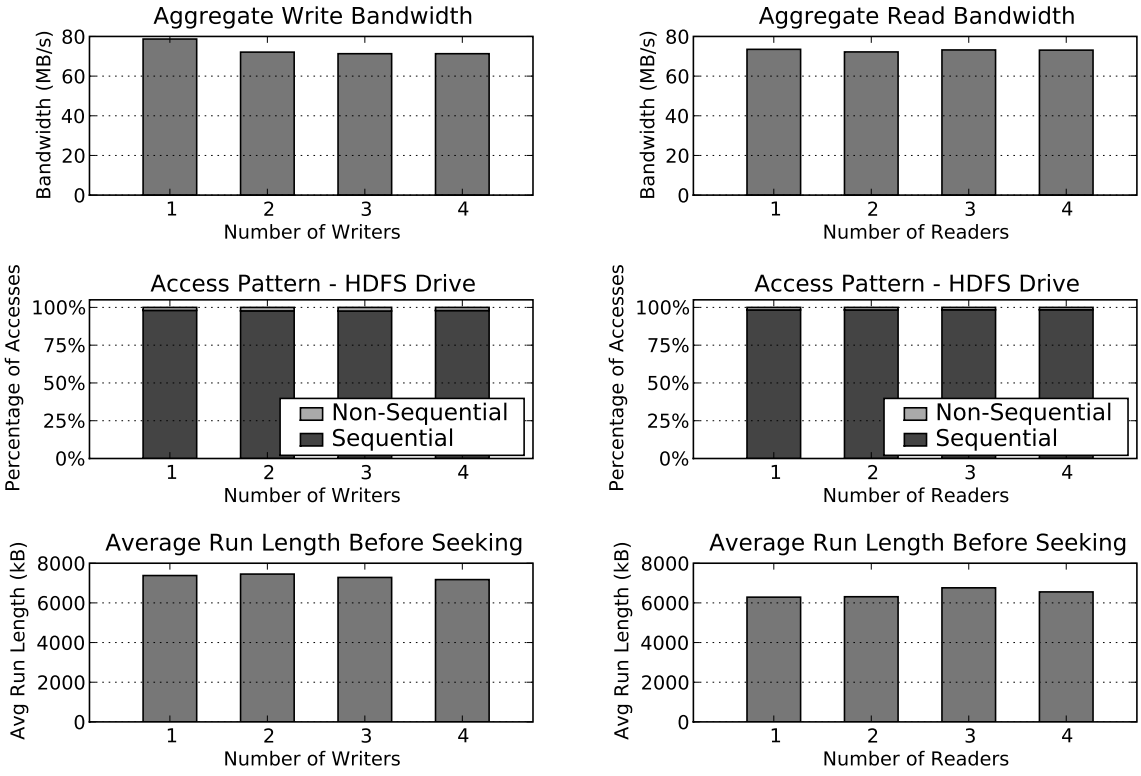
replication is a trivial way to produce concurrent access. The behavior of the synthetic writer with replication enabled is highly similar to the behavior of 2 concurrent writers, previously shown in Figure 5(a). The mix of sequential and random disk accesses is similar, as is the very small average run length before seeking. Similar observations for the read test can be made against the behavior of 2 concurrent readers, previously shown in Figure 5(b). Thus, the performance degradation from concurrent HDFS access is present in every Hadoop cluster using replication.

G. Other Platforms – Linux and Windows

The primary results shown in this paper 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, but space limitations necessitate a brief discussion of results here.

HDFS on Linux suffers from the same performance problems as on FreeBSD, although the degree varies by filesystem and test. 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 in the same test. This compares to a 47% drop in FreeBSD as shown in Figure 5(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 shown in Figure 5(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 shown in Figure 6.

Hadoop in Windows 7 relies on a Unix emulation layer such as Cygwin to function. Write bandwidth to disk was acceptable (approximately 60MB/s), but read bandwidth was very low (10MB/s or less) despite high disk utilization (in excess of 90%). Although the cause of this performance degradation was not investigated closely, the behavior is consistent with disk access patterns using small I/O requests (2kb-4kB) instead of large requests (64kB and up). Because of these performance limitations, Hadoop in Windows is used only for non-performance-critical application development. All large-scale deployments of Hadoop in industry use Unix-like operating systems such as FreeBSD or Linux, which are the focus of this paper.



(a) Concurrent Writers

(b) Concurrent Readers

Fig. 7. Impact of Application Disk Scheduling on Concurrent Synthetic Writers and Readers

IV. Performance versus Portability

As characterized in Section III, the portable implementation of HDFS suffers from a number of bottlenecks caused by lower levels of the software stack. These problems include:

Disk scheduling — The performance of concurrent readers and writers suffers from poor disk scheduling, as seen in Section III-E1. 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 III-E2, 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 III-D. 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.

A. Application 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 application-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

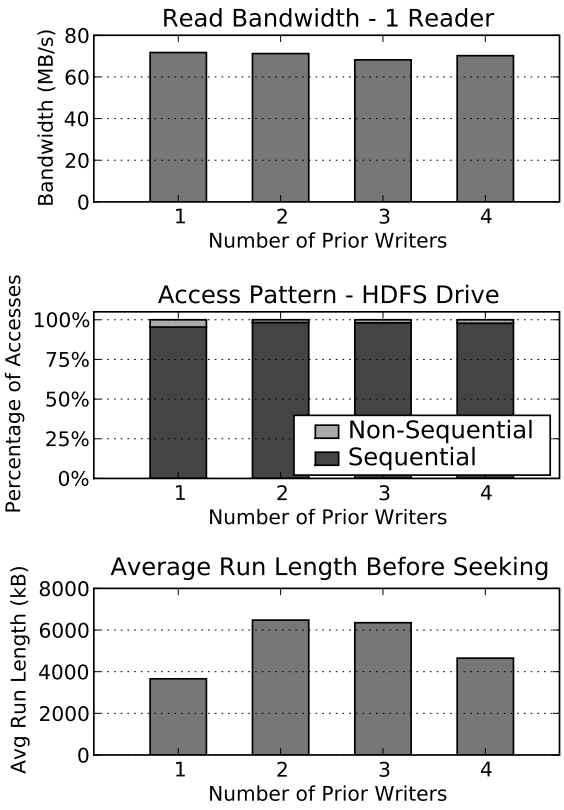


Fig. 8. Impact of Application Disk Scheduling on Data Fragmentation

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 2. 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 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 7(a) and Figure 7(b).

Compared to the previous concurrent writer results in Figure 5(a), the improved results shown in Figure 7(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 Figure 5(b). In addition to improving performance under concurrent workloads, application-level disk scheduling also significantly decreased the amount of data fragmentation created. Recall that, as shown in Figure 6, files created with 2 concurrent writers were split into fragments of under 300kB. However, when retesting the same experiment with the modified DataNode, the fragmentation size exceeded 4MB, thus enabling much higher disk bandwidth as shown in Figure 8.

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.

B. Non-Portable Solutions

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 conjunction 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.

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. 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 2.

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 application-level scheduling on a conventional filesystem (See Figure 7). 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 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.

V. Related Work

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 [20]. 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 [5], [7], [14], [19], [20], [21]. 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 [18]. 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 [10], [15]. 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], [12], [16]. 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 [6], [24]. 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.

Some optimizations proposed here for Hadoop may be present in the Google-developed MapReduce implementation that is not publicly available. 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 [9].

VI. Conclusions

The performance of MapReduce, and Hadoop in particular, has been called into question recently. For example, in some experiments, applications using Hadoop performed poorly compared to similar programs using parallel databases [18], [22]. While such differences are typically blamed on the MapReduce paradigm, this paper shows that the underlying filesystem can have a significant impact on the overall performance of a MapReduce framework. Optimizing HDFS as described in this paper will boost the overall efficiency

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.

Furthermore, the performance impacts of HDFS are 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. This paper is the first to characterize the interactions between Hadoop and storage. Here, we explained how many performance bottlenecks are not directly attributable to application code (or the MapReduce programming style), but rather are caused by the task scheduler and distributed filesystem underlying all Hadoop applications.

The poor performance of HDFS can be attributed to challenges in maintaining portability, including disk scheduling under concurrent workloads, filesystem allocation, and filesystem page cache overhead. HDFS performance under concurrent workloads can be significantly improved through the use of application-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, thus encouraging the spread of this parallel computing paradigm.

REFERENCES

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

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

[3] Hadoop. http://hadoop.apache.org, 2009.

[4] HDFS (hadoop distributed file system) architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html, 2009.

[5] R. 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.

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

[7] 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.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

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

[10] K. DeLathouwer, A. Y. Lee, and P. 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.

[11] S. Ghemawat, H. Gobioff, and S.-T. 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.

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

[13] S. Iyer and P. 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.

[14] S. C. John, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Journal of Real-Time Systems*, 3:307–336, 1991.

[15] S. Kashyap, B. Olszewski, and R. 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.

[16] B. Ndiaye, X. Nie, U. Pathak, and M. Susairaj. A quantitative comparison between raw devices and file systems for implementing oracle databases. Technical report, Oracle / Hewlett-Packard, 2004.

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

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. 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.

[19] A. L. N. Reddy and J. 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.

[20] A. L. N. Reddy, J. 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.

[21] P. J. Shenoy and H. 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.

[22] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[23] W. Tantisiriroj, S. Patil, and G. Gibson. Data-intensive file systems for internet services: A rose by any other name. Technical report, Carnegie Mellon University, 2008.

[24] M. Zukowski, S. Héman, N. Nes, and P. 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.