HadioFS: Improve the Performance of HDFS by Off-loading I/O to ADIOS

by

Xiaobing Li

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 14, 2013

Keywords: High Performance I/O, ADIOS, HDFS, Hadoop

Abstract

Hadoop Distributed File System (HDFS) is the underlying storage for the whole Hadoop stack, which includes MapReduce, HBase, Hive, Pig, etc. Because of its robustness and portability, HDFS has been widely adopted, often without using the accompanying subsystems. However, as a user-level distributed filesystem designed for portability and implemented in Java, HDFS assumes the standard POSIX I/O interfaces to access disk, which makes it difficult to take most of the platform-specific performance-enhancing features and high performance I/O techniques that have already been very mature and popular in HPC community, such as data staging, asynchronous I/O and collective I/O, because of their incompatibility to POSIX. Although it is feasible to re-implement the disk access functions inside HDFS to exploit the advanced features and techniques, such modification of HDFS can be time-consuming and error-prone. In this paper, we propose a new framework HadioFS to enhance HDFS with Adaptive I/O System (ADIOS), support many different I/O methods and enable the upper application to select optimal I/O routines for a particular platform without source code modification and re-compilation. Specifically, we first customize ADIOS into a chunk-based storage system so that the semantics of its APIs can fit the requirement of HDFS easily; then we utilize Java Native Interface (JNI) to bridge HDFS and the tailored ADIOS together. We use different I/O patterns to compare HadioFS and the original HDFS, and the experimental results show the feasibility and benefits of the design. We also shed light on the performance of HadioFS using different I/O techniques. To the best of our knowledge, this is the first attempt to leverage ADIOS to enrich the functionality of HDFS.

Acknowledgments

This author would like to express great gratitude to his advisor, Dr. Weikuan Yu, for his generous support, insightful comments, passionate encouragement and limitless tolerance. It is indispensable for this author's career route having been recruited by Dr. Yu and joined in his amazing and productive laboratory - Parallel Architecture and System Laboratory (PASL). The author similarly would very much like to acknowledge Dr. Xiao Qin and Dr. Sanjeev Baskiyar as vital members of the graduate committee. Without the combined expertise and insights from these individuals, to whom the author is indebted very much, this work would not have been possible.

This project is enabled by a contract provided by Dr. Scott Klasky of Oak Ridge National Laboratory. The author is very grateful for Dr. Klasky's passion for excellence in research and unceasing diligence.

This author is also very thankful for the fellow students in PASL. They are Yandong Wang, Zhuo Liu, Bin Wang, Cong Xu, Xinyu Que, Teng Wang, Cristian Cira and Yizheng Jiao, with whom this author has spent a very enjoyable and fruitful period of graduate study. Particularly, the author appreciates Yandong Wang's guidance and engagement greatly during the research project on Big Data and Cloud Computing. The experience accumulated through those projects helps the author complete this work very much.

Finally, this author would like to thank the Department of Computer Science and Software Engineering and Auburn University for providing such great resources and facilities, with which this work was accomplished.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of Big Data era, an overwhelming amount of data can be generated in our daily life by a wide range of computing facilities, ranging from smart phones, wearable computing devices, to high-end scientific computing clusters and giant data centers enabling the world-wide media and social networking services. To extract meaningful knowledge and economic value from the massive-scale data, MapReduce [24] has evolved as the backbone processing framework since its introduction by Google around 2004. Inspired by the map and reduce functions commonly used in functional programming language, the Google MapReduce programming model inherits the parallelism nature and is equipped with a scalable and reliable runtime system to parallelize the analysis job to process extremely large datasets, which are kept in Google File System (GFS) [26], the distributed storage system inside the framework. The simple yet expressive interfaces, efficient scalability and strong fault-tolerance have attracted a growing number of organizations to build their services on top of MapReduce framework.

The success of Google MapReduce in Big Data era motivates the development of Hadoop MapReduce [2], the most popular open source implementation of MapReduce, and Hadoop Distributed File System (HDFS) [2], the counterpart of GFS. Hadoop MapReduce includes two categories of components: a JobTracker and many TaskTrackers. The JobTracker commands TaskTrackers to process data through the two functions, i.e., map and reduce, which are left to users to define according to particular analysis requirement. Before being launched to run by Hadoop, each MapReduce job includes the definition of the two functions and corresponding configuration options, such as the number of tasks to execute the map and reduce functions, job priority and user authentication. On each TaskTracker, there are a certain number of map slots and reduce slots as resource containers to run tasks derived from the MapReduce job. Task to run map (reduce) function is called MapTask (ReduceTask) and assigned to idle map (reduce) slot. For convenient parallelization and scalable data processing, Hadoop MapReduce divides input data into many splits stored in HDFS; and MapTasks process such splits in parallel. ReduceTasks output the results to HDFS as well. To date, Hadoop has evolved into a huge ecosystem, inside which a lot of tools have been developed to make Big Data analytics more convenient, efficient and insightful. Hadoop Distributed File System (HDFS) is the underlying storage system for the whole Hadoop Ecosystem. Because of its robustness and portability, HDFS has been widely adopted, even without using the accompanying subsystems at times. For instance, HAQW [15], an efficient SQL-engine built for analysis on large dataset, is quite different from the other data processing tools, such as Hive [6] and Pig [14] that translate a user query into a DAG (Directed Acyclic Graph) of MapReduce jobs. HAQW allows user to issue queries to a large amount of data stored in HDFS directly, bypassing the whole MapReduce runtime.

The high rate of information growth and demand for fast analysis drive system engineers to deploy Hadoop onto HPC clusters [21]. However, as a user-level distributed filesystem designed for portability and implemented in Java, HDFS assumes the standard POSIX I/O interfaces to access disk, which makes it difficult to take most of the platform-specific performance-enhancing features and high performance I/O techniques that have already been very mature and popular in HPC community, like data staging, asynchronous I/O and collective I/O, most of which are incompatible to POSIX standard. Although it is feasible to re-implement the disk access functions inside HDFS to exploit the advanced features and techniques, the time-consuming and error-prone modification of HDFS is inevitable.

Large-scale scientific applications, such as global warming modeling and combustion simulation programs, often generate extremely massive volume of data. The gap between the I/O speed and computing power of the high-end clusters motivates many research efforts on the improvement of storage techniques. However, these techniques are often based upon the underlying system supports; and hence not always compatible to each other. So the application using one particular I/O technique has to be modified when ported to another platform; and the cost might be very huge to change the long-term developed and optimized scientific program. This issue is closely related to what we just stated about HDFS. To address it, Adaptive I/O System (ADIOS) [30] has been designed. ADIOS, as a middleware, supports many different I/O methods, data formats and parallel file systems. Most importantly, it enables the upper application to select optimal I/O routines for a particular platform without source code modification and re-compilation. The interfaces of ADIOS for applications to use are as simple as POSIX ones, although not compatible; and new storage systems or techniques can be hooked into it very easily. ADIOS has been widely adopted in HPC community due to its simplicity, extensibility and efficiency.

Therefore, to enable HDFS to fully utilize the power of HPC clusters, we propose a new framework **HadioFS** to enhance HDFS with ADIOS, so that the platform-specific performance-enhancing features and various high performance I/O techniques can be leveraged by HDFS without the cost incurred by source code modification. Specifically, on the one hand, we customize ADIOS into a chunk-based storage system and implement a set of POSIX-compatible interfaces for it; on the other hand, we use JNI [7] to make HDFS able to use the functions of the tailored ADIOS through this new set of POSIX APIs. To investigate the feasibility and advantages of our design, we conduct a set of experiments to compare HadioFS and the original HDFS. For current system prototype, the performance on data writing can be improved by up to 10%. In addition, we also analyze the performance of HadioFS configured with different I/O methods, such as POSIX-IO, MPI-IO and so on, to evaluate if HDFS can benefit from the flexibility of ADIOS.

To the best of our knowledge, this is the first attempt to leverage ADIOS to enrich the functionality of HDFS. Overall, we have made five contributions in this work:

- We have clarified the extensibility limitation of HDFS in terms of the utilization of diverse I/O techniques.

- We have investigated the feasibility to leverage ADIOS to enhance HDFS and quantified the performance potential could be achieved.
- We have customized ADIOS into a chunk-based storage system and encapsulated it into a POSIX-compatible I/O system.
- We have proposed a new framework HadioFS to enhance the extensibility and performance of HDFS by using our tailored ADIOS.
- We have carried out a systematic evaluation on HadioFS in comparison to the original HDFS, and the experimental results verified the feasibility of our method.

The remainder of the thesis is organized as follows. Chapter 2 provides the background for this work. We then describe the motivation of this project in Chapter 3, followed by Chapter 4 that details our way to customize ADIOS and integrate it with HDFS via JNI. Chapter 5 elaborates the experimental results. Chapter 6 reviews the related work. Finally, we conclude the thesis and discuss the future work in Chapter 7.

Chapter 2
Background

In this chapter, we elaborate the background of this work. First of all, we present the general framework of the Hadoop Ecosystem; then, we focus on HDFS, which is modified and enhanced in this work. After the explanation for the runtime mechanism of HDFS on data reading and writing, we introduce ADIOS in terms of its architecture and data file structure. In the end, we discuss Java Native Interface (JNI), which is used in our system to integrate HDFS and ADIOS together.

2.1 Hadoop

Hadoop framework is designed for data-intensive distributed applications. Essentially, it implements the computational model MapReduce [24], in which each job is divided into many parallel tasks assigned to a cluster of nodes. These tasks are categorized into two types: MapTask and ReduceTask. These two are responsible for the execution of user-defined map and reduce functions to process data in an embarrassingly parallel manner. Loss of data and failure of computation due to system glitches are common in large scale distributed computing scenarios. Therefore, to make Hadoop straight to program, the reliability issue of both computation and data is handled within the framework transparently and hidden to the application programmers.

To achieve the required core function and ease of programmability, several subsystems are provided inside the whole Hadoop Ecosystem [2] as shown in Figure 2.1.

The subsystem, Hadoop MapReduce, implements the data processing framework, which encapsulates the computational model MapReduce. One JobTracker and many TaskTrackers are present at this layer. To be specific, the JobTracker accepts job from a client, divides job

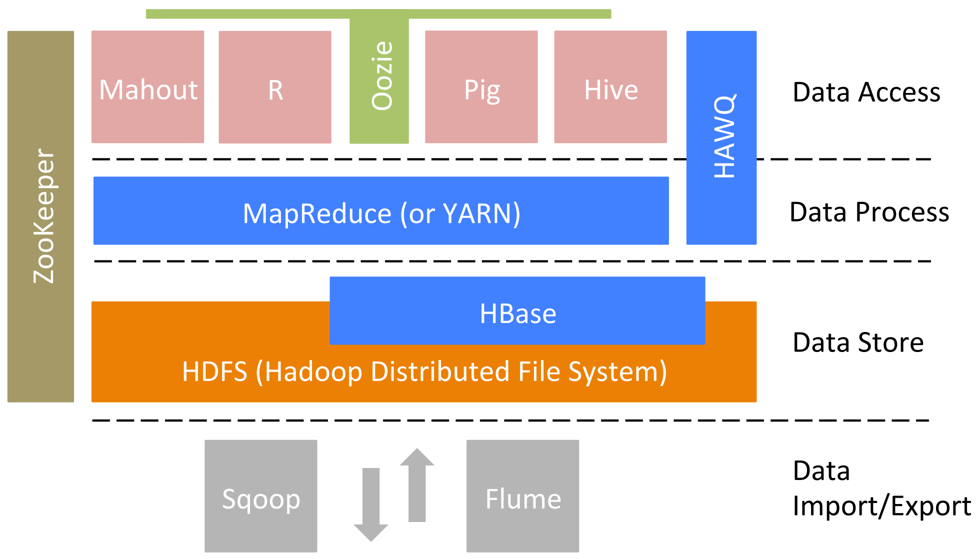


Figure 2.1: Hadoop Ecosystem

into tasks according to the input splits stored within HDFS, and assigns them to TaskTrackers with the awareness of data-locality. In the meantime, TaskTrackers, one per each slave node, take full control of the node-local computing resource via slot abstraction. Two kinds of slots are defined: map slot and reduce slot. On each TaskTracker, the number of both slots are configurable. And they can be regarded as static resource containers to execute corresponding tasks: MapTask or ReduceTask. YARN (MRv2) [10] is the second generation of the Hadoop framework, which splits the resource management and job scheduling functions into different components. In contrast, these functions are closely tangled inside JobTracker in the first generation.

Under the processing framework is the storage subsystem: HDFS [2]. We discuss its structure in detail here, with its runtime feature in the next section. HDFS consists of one NameNode and several DataNodes. The NameNode is responsible to build and manage the file system name space, which is used to map each file name to the locations of corresponding file data. It is not a single but a set of locations because the file is broken into a list of equal-sized blocks that are assigned to perhaps different DataNodes. And on DataNode, each block is kept as a single file, with a few replicas dispersed on other DataNodes to ensure high data reliability.

Based upon the data storage and processing subsystems, many convenient tools have been developed for analysts and programmers to exploit the processing power of Hadoop much more easily. Hive [6] is a SQL-like engine. Analysts can use it to query the data stored in Hadoop compatible file systems. A Hive job is translated into a DAG (Directed Acyclic Graph) of MapReduce jobs, which are scheduled according to the dependency defined in the DAG. HAWQ [15], attempting to replace Hive, is also a SQL engine. But it is designed to access the underlying data directly, bypassing the MapReduce framework to achieve better performance. Pig [14] is a counterpart of Hive and HAWQ. It is also used to extract knowledge from large dataset stored in Hadoop, but modeled with procedural language not declarative language like SQL. Oozie [13] is a workflow definition tool, which supports many types of jobs, like MapReduce job, Pig job, Shell scripts or Java executables. Mahout is developed for machine learning; R connector [17] for statistics; Flume and Sqoop [3, 18] for data movement; and HBase [4] for semi-structured data storage. In addition, considering the complexity and wide use of coordination service within distributed applications, ZooKeeper [19] is developed to expose a simple set of primitives for such service. To sum up, Hadoop is more than MapReduce and becoming more and more popular in today's big data community; HDFS, one of its core components, has been actively studied recently.

2.2 Hadoop Distributed File System (HDFS)

HDFS plays a critical role in the Hadoop Ecosystem as shown in Figure 2.1. In this section, we focus on its runtime features. When accessing data, the HDFS clients only communicate with NameNode for necessary metadata. After that, most of the subsequent operations are performed between clients and DataNodes directly.

To read a file, the client inquiries NameNode for the location of each block belonging to the file. If permitted to access, it will acquire the information of a set of DataNodes, which

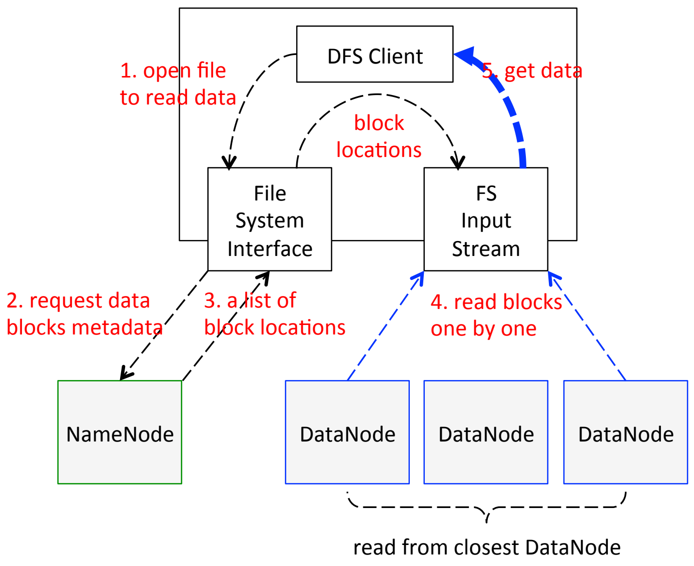

DFS Client
1. open file to read data
5. get data
block locations
File System Interface
FS Input Stream
2. request data blocks metadata
3. a list of block locations
4. read blocks one by one
NameNode
DataNode
DataNode
DataNode
read from closest DataNode

Figure 2.2: Read from HDFS

keep the file blocks. Because of replication, each block might reside on several DataNodes, and the client will select the nearest one, in terms of network hops, to get the block. During the read process, no intervention from NameNode is needed, avoiding potential performance bottleneck. In addition, HDFS supports random seek operation for reads.

To write a file, the client firstly asks NameNode to allocate space from the storage cluster to keep the user file. It will receive a list of DataNodes for each file block. And a replication pipeline is built with this set of DataNodes to store the block. The client then splits the block into small packets; and transmits these packets to the first DataNode in the pipeline; this DataNode stores each packet persistently and mirrors it to the downstream DataNode. The 'store and mirror' action is executed by all the DataNodes in the pipeline; when the acknowledgement from the downstream DataNode is recevied, the DataNode will notify the upstream DataNode the success of receiving packet, and finally the first DataNode in the pipeline will notify the client. The next block will not be written until all the packets from current block are received by all the DataNodes in the pipeline. Different from the read operations, HDFS only supports sequential write operations.

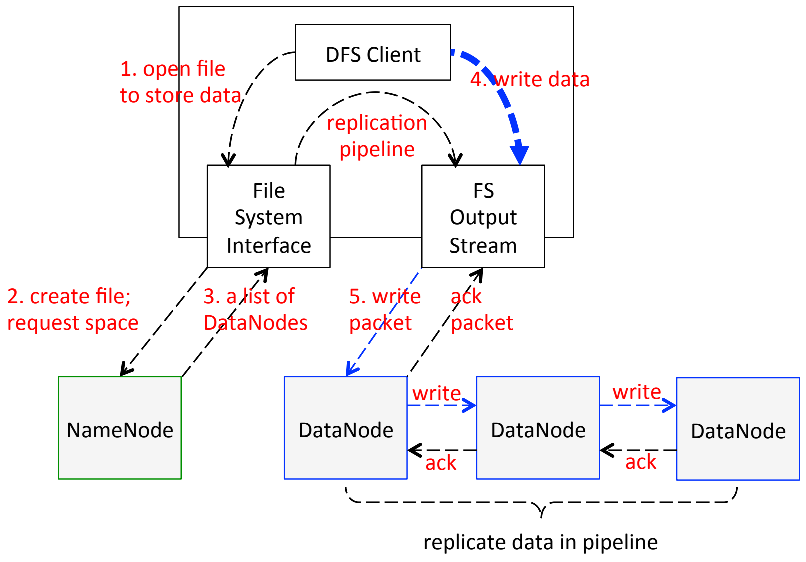


Figure 2.3: Write to HDFS

By default, each block has three replicas. Thus, to balance data reliability and access throughput, HDFS writes the first replica on the local node if the client runs on DataNode; the second one the local rack; and the last one to a remote rack.

2.3 Adaptive I/O System (ADIOS)

ADIOS [30] is a highly configurable and lightweight I/O middleware. It is not a runtime system but a library. Applications need to embed the APIs exposed by this middleware to access the data on disk. By taking use of ADIOS, application can switch among different I/O methods and tune parameters that might impact I/O performance, without source code modification and re-compilation. These features are particularly beneficial to scientific applications, which often require a long time to develop, optimize and verify; and is hard to port to different platforms. But with ADIOS, the scientific application can be regarded as a block-box, when ported or tuned according to the specific characteristics of underlying platforms and even high-level requirement. And on the other hand, to achieve wide adoption in HPC community, ADIOS supports a lot of advanced I/O methods, such as synchronous MPI-IO [11], collective MPI-IO and asynchronous I/O using DataTap system [20]; many high performance file systems, like GPFS [32] and Lustre [9]; as well as several different data formats, such as NetCDF [12], HDF-5 [5] and its native BP format [30], which will be detailed later. However, it should be noted that though ADIOS APIs are as simple as POSIX ones, they are not POSIX compatible, which renders this work very challenging. The details about how we handle the incompatibility will be stated in Chapter 4.

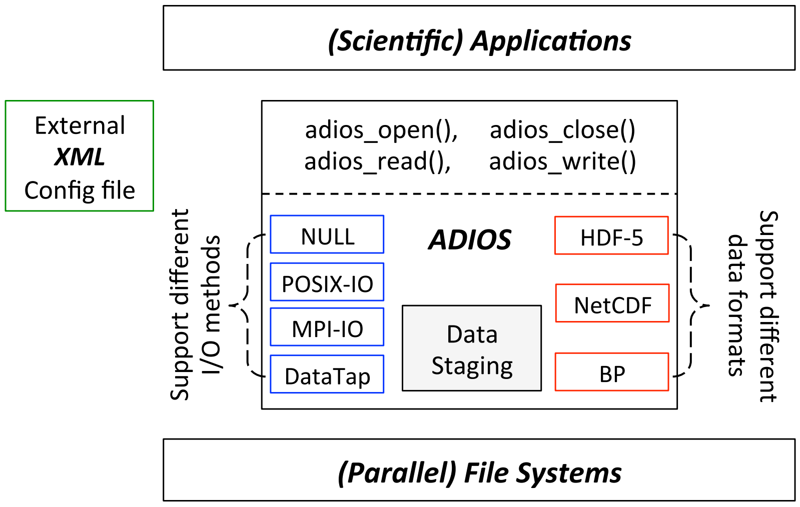

(Scientific) Applications
External XML Config file
adios_open(), adios_close()
adios_read(), adios_write()
Support different I/O methods
NULL
POSIX-IO
MPI-IO
DataTap
ADIOS
Data Staging
HDF-5
NetCDF
BP
Support different data formats
(Parallel) File Systems

Figure 2.4: The Architecture of ADIOS

As shown in Figure 2.4, ADIOS supports a lot of I/O methods, but only exposes a small set of unified APIs, such as *adios_open()*, *adios_write()*, *adios_close()*, etc. Application programmer can utilize these unified APIs to compose the processing logic while decoupling it from concrete I/O systems. In this way, the programmer is freed from many complicated issues, such as how to program with a particular I/O system; which I/O routine fits a specific platform best; and how to achieve portability related to the I/O components. Inside the ADIOS framework, the specific I/O method selected to access disk, e.g. MPI-IO, is determined by modifying the associated XML file before the scientific application starts. In addition to the adaptive selection of methods, parameters critical to performance, such as the size of I/O buffer and its allocation time, can also be tuned via the XML file. Furthermore, considering scientific application might generate very large result set and do so across multiple platforms, ADIOS allows programmer to partition result set into small groups and tune each group independently. Last but not least, to overlap computing and I/O better, ADIOS also provides APIs and corresponding parameters inside XML file for programmer to inform the underlying I/O system of the I/O patterns that the application carries, so that the schedule component inside the ADIOS framework can do necessary coordination.

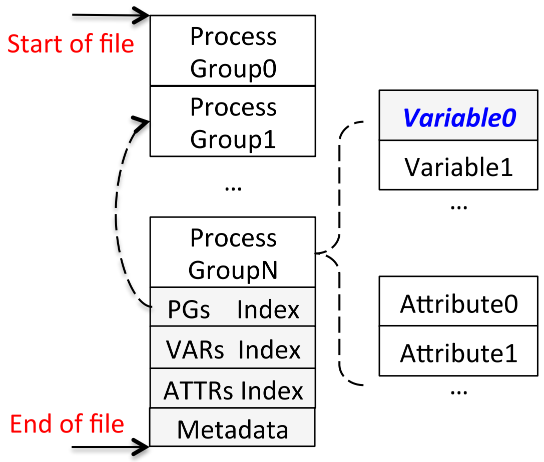


Figure 2.5: The Structure of BP file

As just stated, ADIOS supports several different data formats. But it also has its native data format called Binary Packed (BP). There are two design goals behind BP file: convertibility to other data formats, e.g. HDF-5 and NetCDF; and optimization to I/O performance for both serial and parallel file systems. The former one is achieved by rich data annotation; the later one by relaxing certain consistency requirements. The detailed format of BP file is shown in Figure 2.5. At the end of the file is the metadata to locate progress groups as well as variables and attributes inside each group. Each group is manipulated by one process, and inherently, all these process groups can be accessed in parallel. The synchronization point is file open and close, when the data offset information is exchanged among processes to ensure consistency. With direct index to variables and attributes, the upper applications can employ more diverse and random querying methods. Generally, *variable* is a structure to store concrete data in BP file. Our design enables HDFS to store data blocks into BP file as variables. Details will be discussed in Chapter 4.

2.4 Java Native Interface (JNI)

JNI [7] allows Java programs that run in Java Virtual Machine (JVM) to leverage functions encapsulated in components implemented in other programming languages, such as C or C++; and applications in other languages to use Java objects. This cross-language interaction is needed when platform-dependent features is required but Java cannot supply them; or a library in another language exists and is worth reusing; or time-critical module is required and needs implementing in a lower-level language, like assembly. ADIOS is realized in C and provides the features we need to enhance HDFS, which is in Java. Thus, this work falls into the second category.

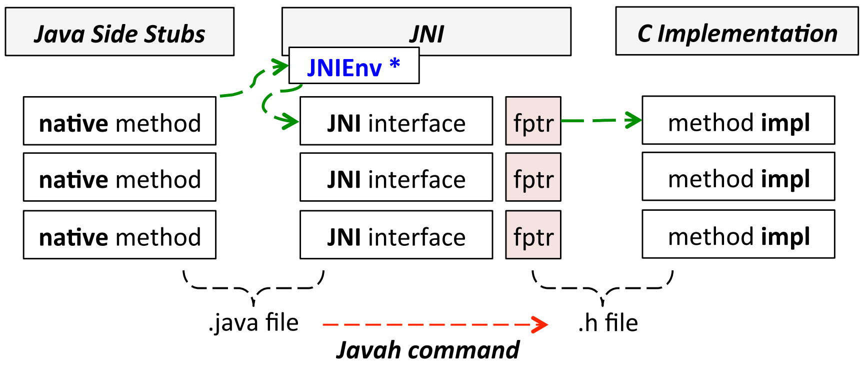


Figure 2.6: The Mechanism of JNI

With Figure 2.6, we explain how JNI enables Java program to utilize C functions. When JNI is used in Java program, some methods are declared by *native* keyword and have no concrete implementation. Compiling them leaves stubs in Java program but generates a header file related to these *native* declared functions. A programmer needs to implement the functions declared inside the header file and build a shared library for them. But now, s/he is free to use any existing C libraries. When the Java program starts, it will load the shared library; and if it calls a *native* declared method, the stub will use the JNI context, i.e. JNIEnv as in Figure 2.6, to locate the corresponding implementation in the shared library; execute it; and return values if required. Nowadays, JNI techniques are especially prevalent in Android mobile application development.

Chapter 3

Motivation

In this chapter, we start with a detailed description of the existing HDFS implementation, aiming to highlight its limitation on using various high performance I/O methods. Then, to motivate this work, we discuss the flexibility of ADIOS, followed by the comparison of disk access performance between HDFS and ADIOS.

3.1 POSIX-IO based HDFS

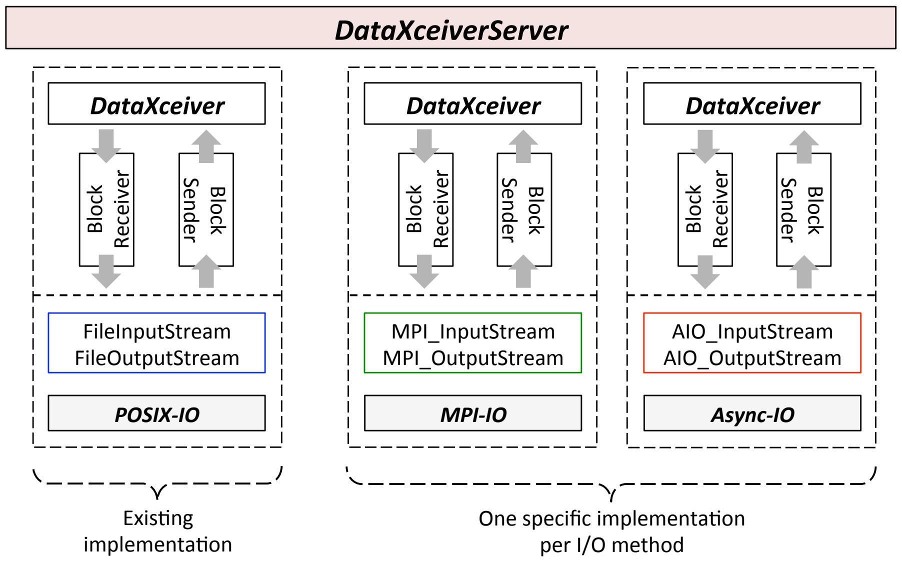


Figure 3.1: The Architecture of HDFS and its Lack of Extensibility

In this section, we present the implementation details of the disk I/O methods in the existing HDFS, and explain the challenges to extend this framework to leverage new I/O techniques. Though a NameNode needs to access disk storage to load and store the file system name space, i.e. *FSImage*, these disk operations are not frequent because the data (metadata from the perspective of the whole distributed file system) is retained in memory during the execution of HDFS, except for periodically checkpointing. Thus, in this work, we primarily focus on the disk operations that happen inside DataNode, where the file data is stored and retrieved frequently.

As shown in Figure 3.1, several integral components within DataNode work together to offer storage service to the clients. DataXceiverServer is the frontend, which keeps listening to the incoming connection requests. When a request from DFSClient is received and accepted by DataXceiverServer, an independent DataXceiver thread is spawned to serve the specific client. Once connected with DataNode, the client will send out data request, *read* or *write*, to DataXceiver, which offers particular service accordingly by leveraging BlockSender or BlockReceiver. And the unit of each data request is a block, whose size can be configured before starting HDFS. It is common to set the block size as 512MB or 1GB to balance metadata overhead and storage throughput.

BlockReceiver is created to store a series of packets belonging to the incoming block, which might be from client or upstream DataNode. If current DataNode is not the last one in the pipeline, it has to mirror the packets to the downstream DataNode. Once receiving a complete packet, BlockReceiver stores it onto disk by standard POSIX write operation. BlockSender is used to read a block belonging to a desired file. It cuts a block into chunks, retrieves each from disk and sends out in one packet. Similarly, it is POSIX read operation that is utilized to get data from disk. While read or write is being conducted, checksum is used to ensure the integrity of the block data. Strictly speaking, although POSIX-IO is de facto and most platforms support it, it is relied on the concrete implementation of Java and specific platform whether to use POSIX-IO or not. HDFS achieves portability because of Java, but it does not gain extensibility, as we are going to state next.

Inside BlockReceiver and BlockSender, FileOutputStream and FileInputStream objects are constructed and execute the disk access work, respectively. While, FileInputStream and FileOutputStream use JNI to call the native platform-dependent disk access functions essentially. For different platforms, the native functions might be different. Therefore, this maximizes the portability of Java program. However, many high performance I/O methods, such as collective MPI-IO, data staging, and asynchronous I/O, are not the system default configuration and even incompatible to the default interfaces. In order to take advantage of them, subclasses extending FileInputStream and FileOutputStream should be implemented to call these high performance I/O methods via JNI. But this needs to modify the source code of applications each time a new I/O method is to be supported. A better solution is to collect the common modules together into a middleware and expose unified APIs to upper applications to decouple the diverse processing logic with concrete underlying I/O systems. As stated earlier, ADIOS is one of the existing solutions. The factors motivating us to choose ADIOS are detailed in next sections.

3.2 The Flexibility of ADIOS

As stated in Section 2.3, ADIOS is lightweight, extensible and highly configurable.

ADIOS is a thin layer between the processing logic and I/O systems, because it is just a C library without gigantic runtime environment that most middleware tools have. And its core only includes the XML parser, function pointer based extension framework, and the buffer management components. To keep it lightweight, ADIOS itself does not have any functional I/O components. All the disk access work is packed and forwarded to the underlying I/O system that is determined by the external XML file.

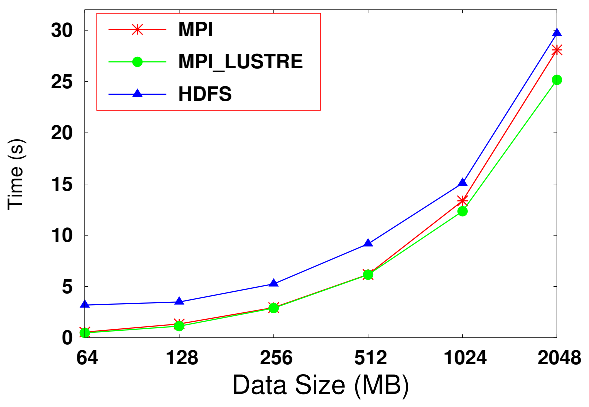
It is straightforward to hook new I/O systems into ADIOS. Its open extension framework is built upon a mapping table [30], in which the key is the I/O method name, such as *POSIX*, *MPI*, *MPI_LUSTRE*, *NC4* and so on. At the same time, the value is a list of functions corresponding to the unified ADIOS APIs. For instance, the list for *MPI* includes *adios_mpi_open()*, *adios_mpi_write()*, *adios_mpi_close()*, etc. Whenever to add a new I/O system, we need to implement the corresponding functions and add an entry into the mapping table so as to find the new list of I/O methods by name. And ADIOS will assign to the set of exposed unified APIs, namely *adios_open()*, *adios_write()*, *adios_close()*, etc., the selected list of functions, when ADIOS context is initialized based on the external configuration file. When the upper application calls the unified APIs, the selected I/O system will take over the disk access work.

As mentioned in Section 2.3, ADIOS supports not only adaptive selection of I/O methods but also tuning of performance critical parameters; and partitions the whole result set into small groups so as to configure each of them independently. And this high configurability is accomplished by maintaining only one single XML file. The details on the configuration file is stated in Chapter 4, where we customize it to make the semantics of ADIOS APIs fit the requirements of HDFS' disk access modules.

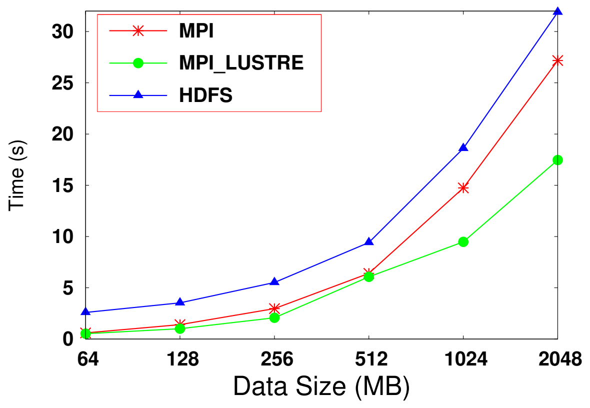
3.3 Comparison between ADIOS and HDFS

Here we discuss the advantages of ADIOS from the perspective of disk I/O performance. We conduct a group of experiments on a cluster of 6 nodes, each of which has one SATA hard-driver and 8GB memory. All these nodes are connected with 1 Gigabit Ethernet. As to the deployment of HDFS, the NameNode runs on one dedicated node, with DataNodes on another four nodes; and the remaining one acts as a client outside the storage cluster. Besides, the number of replicas is set to 1; HDFS block size is set to 512MB. To compare with HDFS that uses POSIX-IO, we configure ADIOS to leverage MPI-IO and MPI_LUSTRE methods to access disk, respectively. To enable MPI_LUSTRE, we have also deployed Lustre file system [9] in a similar way, i.e., one dedicated metadata server and four object storage servers. Compared with MPI-IO and POSIX-IO, MPI_LUSTRE is optimized particularly for Lustre when storing data. We run each case five times in a row and compute the average result to eliminate the system initialization time and occasional results.

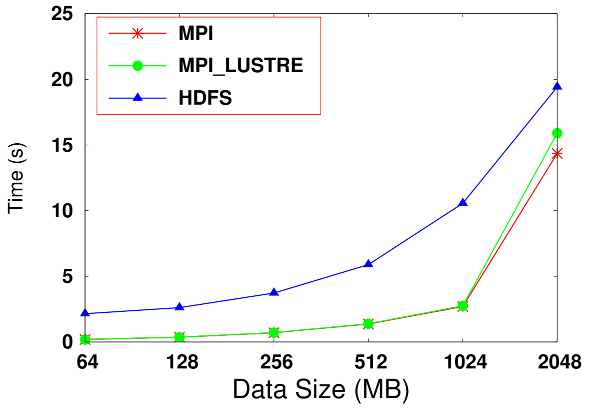
To investigate the write performance, we firstly execute the writing process on the node outside the storage cluster to remove the possibility that data is stored locally. As shown in Figure 3.2 (a), ADIOS outperforms HDFS while the data size is changed. Although



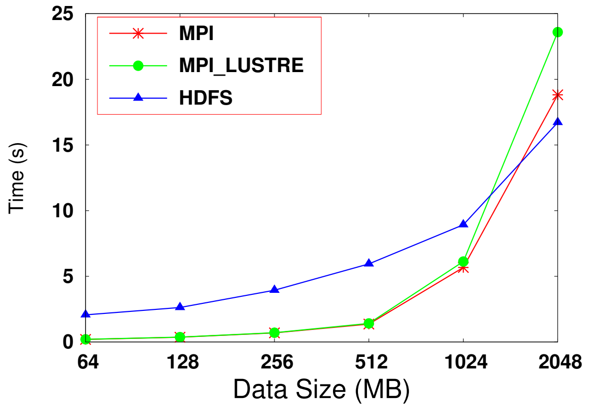
(a) Writing data outside the cluster



(b) Writing data inside the cluster



(c) Reading data outside the cluster



(d) Reading data inside the cluster

Figure 3.2: Comparison of different I/O methods

the improvement ratio decreases as the data size grows, ADIOS can still achieve up to 32.8% improvement when writing 512MB data. This is encouraging because our design goal is to replace the disk I/O modules within DataNode with our customized ADIOS; while, DataNode stores and retrieves data in block unit, which is commonly set around 512MB as we stated in Section 3.1. The nearly constant absolute improvement brought by ADIOS is resulted from its wise buffering mechanism, via which small data segments are accumulated before being flushed to disk. We then migrate the writing process onto a storage node within the cluster, and observe even larger improvement, with up to 49% when writing 1GB data as shown in Figure 3.2 (b). Furthermore, writing large dataset inside the storage cluster accelerates ADIOS but slows HDFS down. This is because, in addition to the more effective memory management, ADIOS can also take good use of the underlying parallel file system to balance storage overhead. With specific optimization for Lustre file system, MPI_LUSTRE method achieves best writing performance in both scenarios.

Different from write operations, the read performance of ADIOS is not always better that HDFS. As shown in Figure 3.2 (d), HDFS completes earlier than ADIOS when reading 2GB data from disk. However, as we explained above, it is around 512MB that our interest resides. In that previous case, ADIOS is faster than HDFS by as much as 70%. The reason why the read performance of ADIOS degrades so quickly when data size grows large is that it tries to reserve sufficient memory before accessing disk for data; while, co-locating client with Lustre storage daemons leads to severe contention on memory resource, which impacts the performance very negatively. Therefore, we can also observe better performance of ADIOS when it reads large dataset outside the cluster, as shown in Figure 3.2 (c).

Based on the aforementioned analysis and observation, we design and implement an approach, as detailed in the next chapter, to make HDFS able to gain benefit from the flexibility and better disk I/O performance of ADIOS.

Chapter 4

Design and Implementation

In this chapter, we start with the description of our new framework, called **HadioFS**, designed to enhance HDFS by off-loading its disk I/O to ADIOS. Then, we propose the approach to customize ADIOS. Based on the customization, we wrap the original ADIOS APIs into a set of POSIX compatible ones, upon which the modification within HDFS is dependent. At the end of this chapter, we summarize the constituent components, which HDFS needs to leverage the functions of ADIOS.

4.1 The Architecture of HadioFS

Figure 4.1 illustrates the general architecture of our design. We keep the HDFS APIs intact, while enhancing HDFS to utilize the efficient and flexible ADIOS. Two extra layers are introduced between HDFS and ADIOS to integrate them both together.

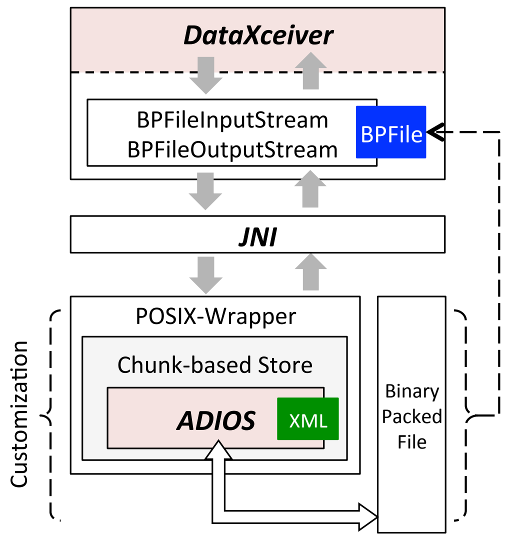


Figure 4.1: The Architecture of HadioFS

The layer inside HDFS includes the components encapsulating the BP file and its I/O streams. In the existing implementation of HDFS, the file to store data block is abstracted as plain byte stream. Recalled in Section 3.1, *File*, *FileInputStream* and *FileOutputStream* are utilized to access these disk files. There is no such hierarchical structure within them as in BP files, whose structure is presented in Section 2.3. If we construct an ordinary file object upon BP file via new File(), the operations, like to get file size or to seek in file, will behave abnormally. Additionally, BP file only supports the open modes 'w' (write-only), 'r' (read-only), and 'a' (write-only without erasing existing content). Therefore, we design and implement a dedicated file object, i.e. *BPFile*, according to the specific structure of BP file. However, it is not sufficient using only an abstraction of the static file structure. It will not work as expected to access the BP file via *FileInputStream* or *FileOutputStream*, which is designed for byte stream based file, because the runtime states, like r/w pointers, are supposed to be maintained in a way matching with the underlying file abstraction. Therefore, we also realize the I/O streams corresponding to the BP file, i.e. *BPFileInputStream* and *BPFileOutputStream*.

The layers that encapsulate ADIOS consist of the components to transform the ADIOS native APIs into POSIX compatible ones, which enable the implementation of the Java side relevant objects stated above. The gap of the interfaces' semantics between ADIOS and POSIX renders it challenging to accomplish this transformation. We use the write operation as an example to elaborate this issue.

In POSIX standard, the write interface is specified as 'ssize_t **write** (int fd, const void *buf, size_t count)' [16], which means writing up to *count* bytes from the buffer pointed by *buf* to the file referred to by the file descriptor *fd*. The amount of bytes to store cannot be known before the interface is called. And an explicit writing pointer is maintained, which is incremented automatically after each execution of this operation. However, the write interface provided by ADIOS is 'int **adios_write** (int64_t hdlr, char *var_name, void *var_value)' [30]. *hdlr* is a handler pointing to an internal data structure, which includes the context information, such as the file descriptor of the BP file, I/O buffers and a set of offsets for variables and attributes. The content to write is in a contiguous memory space pointed by *var_value*. The *var_name* passed to this function should correspond to the name in the variable definition, which is listed in the XML file. Within the variable definition, the size of *var_value* is also configured beforehand. Because the variables are defined before the application runs, their offsets in the file can be calculated even before writing is executed, which enables ADIOS to store these variables via their names, without maintaining an explicit writing pointer.

Read operation has similar difference, i.e., POSIX has an explicit reading pointer, while ADIOS locates contents by variable names. In fact, simply speaking, our method to handle the semantics gap is to tailor ADIOS to work like that it has an automatically maintained explicit r/w pointers. The other issues, (1) that ADIOS needs to initialize and finalize the context before and after disk access work; and (2) that it does not support read-write mode when opening the BP file, are also handled by this layer. The approaches are elaborated in the next section.

In addition to the two primary layers, there are several other JNI related modules that bridge them together in order for HDFS to utilize the ADIOS functions. Their details will be explained in Section 4.3.

4.2 The Customization of ADIOS

In this section, we present how to customize ADIOS and wrap it up to achieve the POSIX compatible APIs. Accordingly, we partition this task into two subtasks. The first one is to tailor the configuration file of ADIOS to restrict it into a storage system, which accesses disk in chunk unit; the second one is to add the r/w pointer mechanism into the tailored ADIOS via a wrapper component in order to expose the POSIX compatible APIs for the upper layer.

Figure 4.2 shows the details of an external XML file, which ADIOS uses to configure itself when starting. Before elaborating the customization approach, we firstly explain the meanings of the items within the XML file [30]. The *adios-config* is a global container. The programming language used by the application is specified here. For instance, we set it as C, in which the wrapper components are implemented. Within *adios-config*, there can be multiple *adios-groups*, each of which can be configured independently. Every group is named and contains a set of variables and attributes, as well as the I/O method it will use. In our example, there is a group called *hdfsblock*, which contains many variables with type information, such as the long integer variable *chksize*, and will be stored and retrieved by MPI method. Some variable also has an extra attribute called *dimensions* to indicate that this is an array, whose size is determined by the product of this attribute value and the size of the element type. Besides, *buffer*, setting the memory space used to stage data, has attributes, like size and allocation time. ADIOS supports some other configuration items as well, but they are not necessary to this work.

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="hdfsblock" coordination-communicator="comm">
        <var name="ttlsize"  type="long" />
        <var name="chkcnt"   type="long" />
        <var name="lastchk" type="long" />
        <var name="lastpos" type="long" />
        <var name="chksize" type="long" />
        <var name="chunk0"  type="byte" dimensions="chksize" />
        <var name="chunk1"  type="byte" dimensions="chksize" />
        ...
        <var name="chunk9"  type="byte" dimensions="chksize" />
    </adios-group>
    <method group="hdfsblock" method="MPI"/>
    <buffer size-MB="64" allocate-time="now"/>
</adios-config>
```

Figure 4.2: The Customization of ADIOS configuration file

Now, we elaborate how to make a chunk-based storage system based upon ADIOS. First of all, we define some chunks in the configuration file. As shown in Figure 4.2, ten chunks, each of which is an array of *chksize* bytes, are defined. The total amount of chunks can be changed, but the product of *chksize* and the number of chunks should not be smaller than the size of one HDFS block. That is because, in our current design, one BP file stores one HDFS block. For the other variables, $ttlsize = chkcnt * chksize$; and the (*lastchk*, *lastpos*) tuple is the last writing position when the BP file is closed. As we just stated, *ttlsize* should be larger than HDFS block size. Configured in this way, ADIOS can only read or write data belonging to one HDFS block in chunk unit, because the ADIOS APIs use predefined variable name and size, i.e., *chunk#* and *chksize*, to access BP file.

Then, based upon this chunk-based storage system, we design the POSIX compatible r/w operations. We start with the write operation. Its interface is declared as 'ssize_t **write** (int fd, const void *buf, size_t count)'. We maintain an explicit writing pointer, which is initialized as 0 when the BP file is opened with 'w' mode. Then, the first chunk, i.e., *chunk0*, is created and retained in memory. Data passed to the write operation above is copied into chunk0; meanwhile, we increment the writing pointer by the number of bytes successfully copied. Once the first chunk is fully filled, we invoke **adios_write** to store it into BP file via its name and create a second in-memory chunk, i.e., *chunk1*. Chunk index and writing pointer is supposed to be maintained correspondingly so that data later can be retrieved correctly by read operation given the offset parameter. This procedure is iterated until all of the data belonging to one HDFS block is stored; then, the metadata is updated and written to BP file. Current design does not support seek operation for writing. In fact, random write is not necessary for our framework, because the HDFS block is written to disk only in sequential manner. As to the read operation, a reading pointer is also maintained explicitly. It is incremented after each execution of read; and can also be set randomly by seek operation. The chunk is loaded and cached when the pointer is set into a chunk that has not been in memory yet. Then, the data is copied from the cache space into the user's buffer. When cache space is used up, loading new chunks will evict old chunks. Append operation is supported as well. The stored metadata (*lastchk*, *lastpos*) is used to resume the writing pointer before storing any new data.

During the process of all these operations, there is at least one chunk kept in memory. As to the write operation, this chunk accumulates small data segments before flushed to disk; for the read operation, this chunk is cached for subsequent data requests. Therefore, the *chksize* parameter is very critical to the whole system performance. Its performance impact will be discussed in detail in Section 5.2.

4.3 Implementation Details

We have implemented the aforementioned two layers and taken use of JNI [7] to bridge them together, as shown in Figure 4.1. The enhanced HDFS can access data via ADIOS. To leverage high performance I/O methods, like MPI-IO, asynchronous I/O or data staging, we now can only change the method setting item within the external XML file, without any source code modification or re-compilation. Additionally, the HDFS APIs are kept intact; and applications can enable or disable the ADIOS enhancement by just changing the configuration file of HDFS.

The details of JNI techniques are elaborated in Section 2.4. While, in practice, it is important to avoid memory copies between the Java program and C library. As shown in Figure 4.3, we use JNI direct buffer [8] to wrap the in-memory chunk allocated at C side into a Java side ByteBuffer object, so that the C side pointer and Java side ByteBuffer object refer to the same memory region. In other word, this is shared memory between C and Java. Therefore, no cross-language data copies occur when the in-memory chunk is manipulated by ADIOS functions, such as **adios_read** and **adios_write**. But the memory region allocated at C side is outside the JVM heap. Much overhead will be incurred if the Java program copies data into or out of this region. Therefore, putting (getting) small data segments into (from) the direct ByteBuffer object synchronously is very time-consuming. By introducing the shadow buffer as shown in figure, we accumulate small data segments into large one and execute the through-JVM copy asynchronously to eliminate this performance bottleneck from the critical path of HDFS block reading and writing. Besides, all the chunks are also stored and loaded in a dedicated thread asynchronously. When **adios_close** function is called, all the data still within the memory buffer has to be flushed onto disk; then metadata is updated and stored in BP file as well. This often makes **adios_close** function very slow. To reduce the turnaround time of client, who starts to write the second block only after receiving the acknowledgement that the first block is successfully stored into HDFS, we also asynchronously close the BP file.

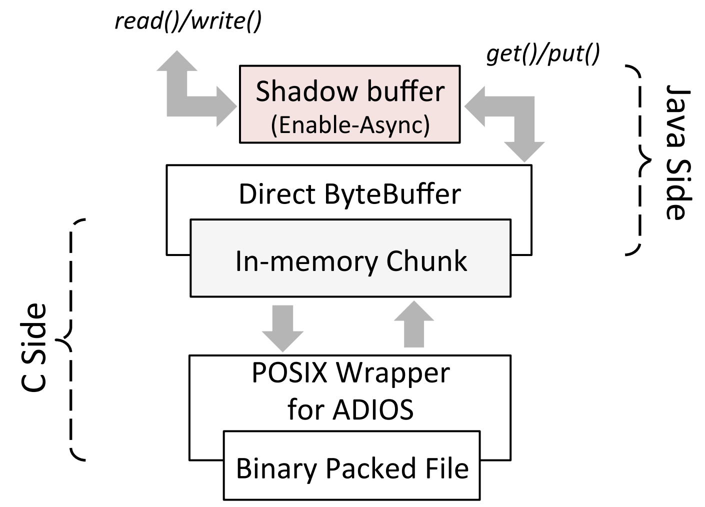


Figure 4.3: The Implementation Details of HadioFS

BPFile object and its I/O streams, i.e., *BPFileInputStream* and *BPFileOutputStream*, are implemented based upon the Java side JNI stubs. The places to hook these ADIOS related objects into HDFS are inside BlockReceiver, BlockSender and FSDataset classes, where originally FileOutputStream and FileInputStream objects are constructed and execute the disk access work, respectively. In our implementation, we hybrid them together in such a way that the ADIOS enhancement can be disabled or enabled by user without changing any source code.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of HadioFS in comparison to the original HDFS. We begin with parameter tuning. The disk I/O performance of HadioFS is sensitive to the chunk size, which controls the size of memory buffer directly related to the disk access operations. We firstly determine the optimal value for this parameter before launching the subsequent evaluation experiments. Then, we compare the writing performance between HadioFS and HDFS with two different patterns. The first one is 'single writer', where one process inside the storage cluster issues writing requests; the second one is 'multiple writers', where writers on all DataNodes store a specific amount of data to HDFS simultaneously. The reading efficiency of current HadioFS system is restricted by the characteristics of ADIOS' read operation [35]. Details about why HadioFS is slower than HDFS on reads are explained to motivate the future work. After that, we investigate the flexibility of HadioFS via different I/O methods supported by ADIOS.

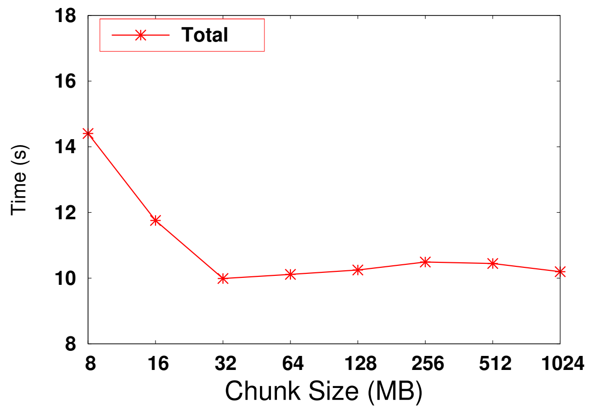
5.1 Experimental Setup

Cluster setup: All the experiments are conducted on a cluster of 5 nodes, which are connected with 1 Gigabit Ethernet. Each node is equipped with four 2.00 GHz hex-core Intel Xeon E5405 CPUs, 8 GB memory, and 1 Western Digital SATA hard-drivers featuring 250GB storage space.

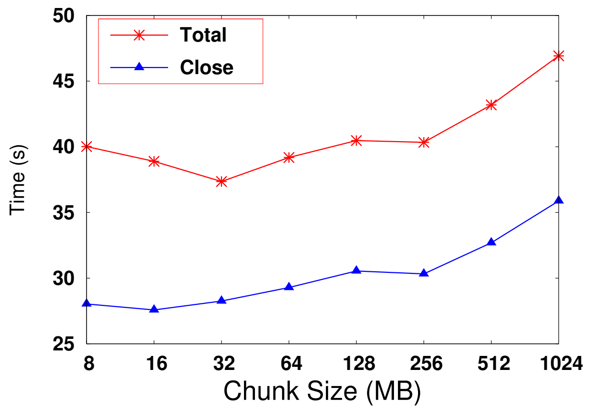
Hadoop setup: In all experiments, we use Hadoop version 1.1.2, from which HadioFS is implemented. As to the deployment of HDFS, the NameNode runs exclusively on one node; with DataNodes on another four nodes. The number of replicas and HDFS block size will be changed for specific experiment.

5.2 Tuning of Chunk Size

In this section, we conduct a set of experiments by just changing the size of chunk as defined in Section 4.2, so that we can determine the optimal value for this performance-critical parameter, called *chksize*, which controls the size of buffer directly related to the disk access operations inside HadioFS.



(a) Using Asynchronous close



(b) Using Synchronous close

Figure 5.1: Writing Performance with different Chunk Sizes

To begin with, we analyze how this parameter affects the performance of data writing. For all of these experiments, we write 1GB data by launching a single process on one DataNode. The number of replicas is set to one; and the HDFS block size to 1GB. Besides, ADIOS uses MPI as the I/O method. This configuration aims to mitigate the interference coming from operations, such as replication pipelining, in order that we can focus on the performance analysis of *chksize*.

In Section 4.3, we discuss the performance bottleneck caused by file close operation, i.e., **adios_close**, which flushes the data left inside memory onto disk before closing the BP file. Waiting for its completion, client will suffer from long turnaround time. Thus, we relax the data consistency requirement and decouple the time-consuming close operation out of the critical path of HDFS block writing via asynchronization. Shown in Figure 5.1 (a) is the performance of HadioFS using asynchronous close operation. The optimal performance occurs when *chksize* is set to 32MB. And as it decreases, the execution time increases very fast. Smaller chunk size means larger amount of chunks. And both values determine the duration and frequency of **adios_write**, which is used to store a fully filled in-memory chunk. When this function is called, overhead is incurred to maintain relevant data structures and copy data into the staging area inside ADIOS. (If this area becomes full, data within it has to be flushed to the BP file.) Although it is decoupled out of the critical path of HDFS block writing, storing chunk to BP file, if issued too frequently, can also lead to dramatic performance degradation through interfering the normal data writing flow. And high frequency is much more harmful to the whole performance than long duration of each execution of chunk storing. Therefore, setting chunk larger than 32MB does not deteriorate the writing performance too much.

The performance of HadioFS with synchronous close operation is shown in 5.1 (b). Meanwhile, the cost of close operation is dissected out. The best value for *chksize* is also 32MB. But the performance trend is quite different from that of HadioFS with asynchronous close operation. With smaller chunk, the execution time increases due to high frequency of chunk storing via **adios_write** as well. But as the chunk size grows larger, the performance also degrades dramatically. This is due to the close operation, which costs more time to flush a larger chunk. As shown in 5.1 (b), when the chunk size is increased from 32MB to 1024MB, the execution time grows 20.38%; while, the cost of close grows 21.24%. And within the whole execution time, the percentage of the close operation cost are 75.63% for 32MB and 76.46% for 1024MB, respectively.

Then, we investigate the influence of parameter *chksize* in terms of reading performance. Similarly, for this set of experiments, we read 1GB data out of HDFS by launching a single process on one DataNode. The number of replicas is one; the HDFS block size is 1GB; and MPI is set as the I/O method for ADIOS. As shown in Figure 5.2, the lowest execution time can be achieved when *chksize* is set to 256MB. At this point, the HDFS block reading flow and the asynchronous chunk prefetching flow can overlap with each other best. Recalled, if

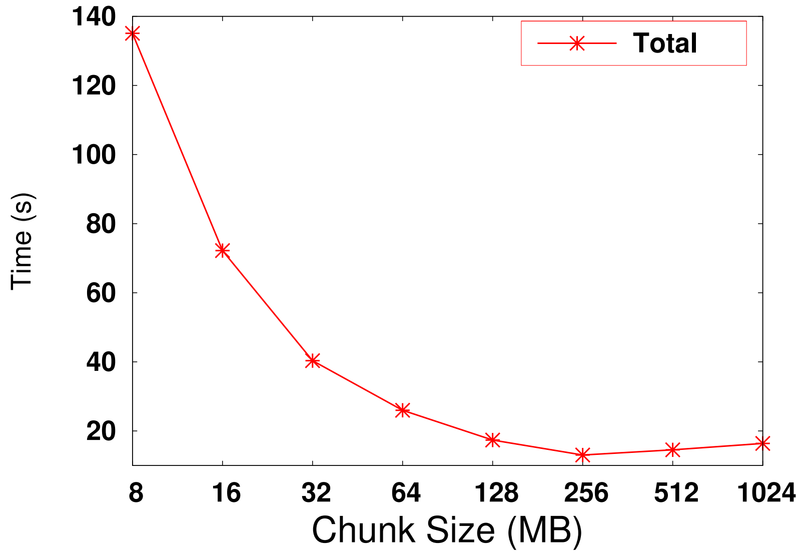


Figure 5.2: Reading Performance with different Chunk Sizes

the chunk size is so small that chunk storing is issued too frequently, the writing performance is degraded very dramatically. The reason is that the asynchronized **adios_write** operations interfere the normal HDFS block writing flow *indirectly*. However, as to reading, if the data is not loaded in time, the normal reading flow is blocked *directly*. This is why the execution time grows when the chunk size is set smaller or larger than 256MB, as show in Figure 5.2. Specifically, if the chunk is too small, in-memory data is consumed so fast that the data loading thread cannot catch up with the block reading thread; while, if the chunk is too large, loading the first chunk stalls the whole reading flow for a long time.

In the rest of our experiments, we use 32MB as the value of parameter *chksize* on data writing; and 256MB on data reading. And in our future work, this parameter will be set for different operations adaptively.

5.3 Analysis with different I/O patterns

In this section, we investigate the performance of HadioFS in comparison with the original HDFS by using different I/O patterns. For the subsequent experiments, the HDFS block size is set to 512MB; and the I/O method for ADIOS to MPI.

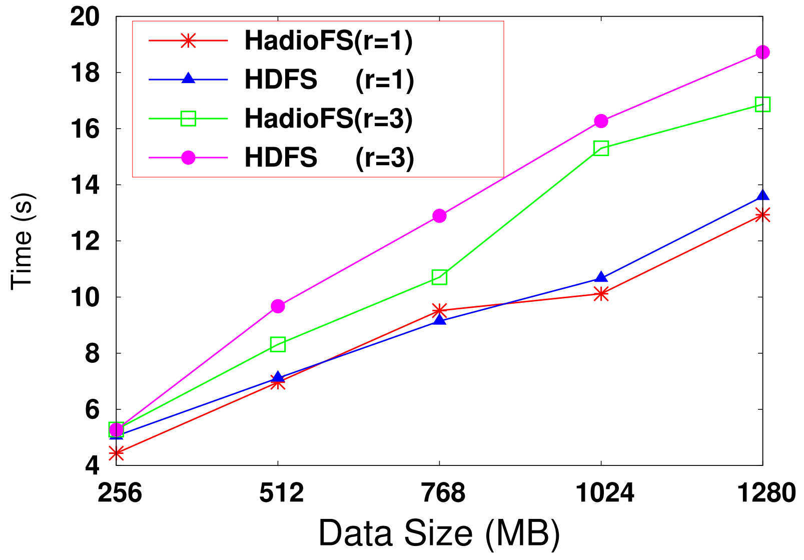


Figure 5.3: Performance of Single Writer

Firstly, we evaluate the writing performance of HadioFS. During the experimentation, each pattern has specific number of writing processes, size of dataset and replication level. We begin with the 'single writer' pattern, i.e., one process on a DataNode issues writing requests. When the replication number is one, most of the data is stored locally. As shown in Figure 5.3, the performance of HadioFS and HDFS are very close and increase linearly to the size of dataset. But HadioFS outperforms HDFS a little (4%) when the data size grows up to 1.2GB. This is because the asynchronous close operation, designed for HadioFS, can enable DataNode to acknowledge client the completion of writing even before the data is stored onto disk.

Data blocks are pipelined and stored onto three different DataNodes for high reliability when the replication number is configured as three. As shown in Figure 5.3, the improvement of HadioFS is enlarged with the increment of replication level. It achieves 10% acceleration when writing 1.2GB data. This is also attributed to the asynchronous close operation. With it, downstream DataNode can acknowledge the upstream one prior to finishing flushing data buffered in memory. While, every DataNode in the original HDFS replication pipeline has to wait for the completion of the last POSIX write function call before notifying its upstream. In theory, the longer the replication pipeline is, the more speedup can be gained by HadioFS.

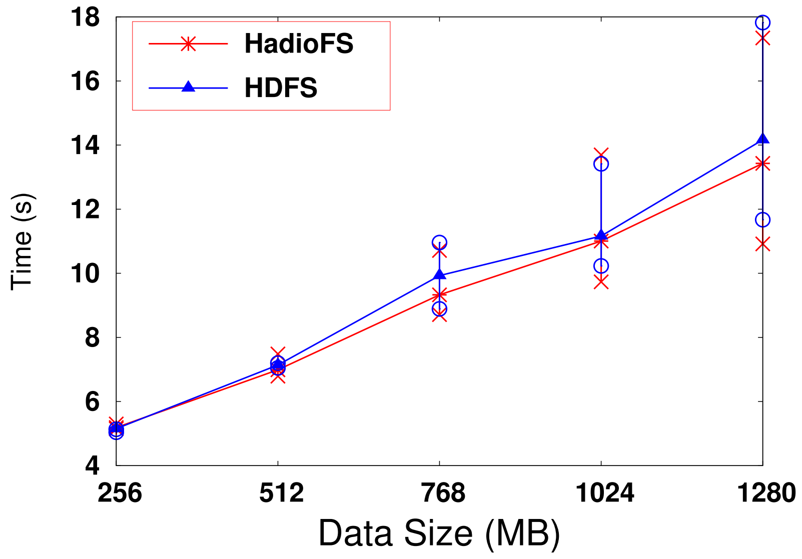


Figure 5.4: Performance of Multiple Writers

Subsequently, we analyze the 'multiple writers' pattern. Different from the 'single writer' pattern configured with three replicas, which can also activate multiple DataNodes during writing, this pattern launches four writing processes, one per DataNode, simultaneously and does not introduce interdependence among DataNodes. In addition, the replication number is set as one in this experiment. The minimum, maximum and average execution times for each data size are all plotted in Figure 5.4. The average time is calculated among the four parallel writers. As shown, the performance of HadioFS is just slightly better than that of HDFS. The contention incurred by the metadata management at the single NameNode offsets the improvement brought by the asynchronous close operation. And for both, this contention also leads to dramatic variance of the execution time among these parallel writers when the size of dataset increases.

Next, we evaluate the reading performance of HadioFS. The experiment configuration is similar with what for the writing tests. The chunk size is set to 256MB based upon the analysis in Section 5.2. We conduct the 'single reader' test. As shown in Figure 5.5, the performance of HadioFS is much worse than HDFS. To find out why HadioFS is so slow on reading, we dissect its execution time as in Table 5.2, as well as the dissection on writing execution time.

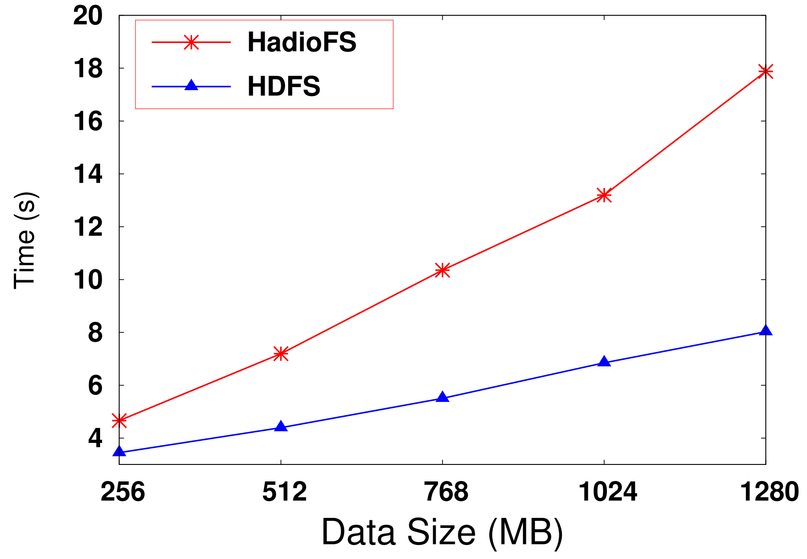


Figure 5.5: Performance of Single Reader

Table 5.1: Detailed Profiling for Writing

Data Size	Client	Server		
		Write	Close	Store
512MB	6.529s	0.651s	1.417s	8.601s
1024MB	10.090s	1.145s	2.278s	20.442s

Table 5.2: Detailed Profiling for Reading

Data Size	Client	Server		
		Read	Close	Load
512MB	7.202s	3.626s	0.426s	2.087s
1024MB	13.194s	7.596s	4.519s	4.519s

The total execution time instrumented by client is placed in the **Client** column. The execution time of the operation that is executed on DataNode is recorded in the **Server** column. Specifically, the asynchronously data storing and loading times are in the **Store** and **Load** columns, respectively. As shown in Table 5.1, the time cost by write operation is very small, because the data just needs moving to the shadow buffer. Then, another dedicated thread will put the data into the in-memory chunk and store the chunk to disk asynchronously when it becomes full. However, as shown in Table 5.2, the time spent by read operation, which gets data out of the shadow buffer, is longer than that of data prefetching. This means the normal data reading flow is blocked by the asynchronously chunk loading flow often if not always, even though 256MB *chksize* can provide the best chance for pipelining as stated in Section 5.2. Getting data from the memory region outside JVM is one reason for the slow data prefetching, but the root cause is the reading mechanism of ADIOS, that the BP file needs to be closed to commit the completion of read operation before the variable value can be used. This inherent restriction renders each chunk loading very time-consuming, which finally leads to the inefficient HDFS block retrieval. We will address this restriction in our future work in order to improve the reading performance.

5.4 Analysis with different I/O methods

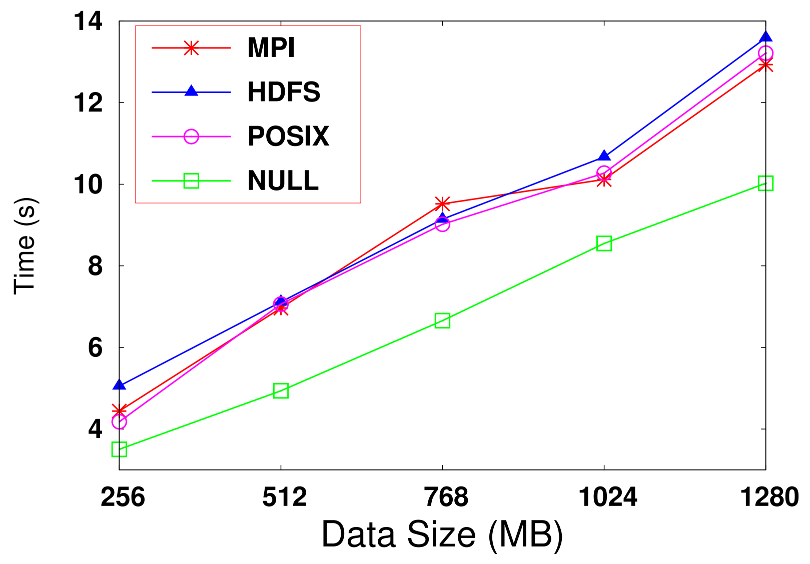


Figure 5.6: Writing Performance of different I/O Methods

Application using ADIOS is capable to switch I/O methods without re-compilation. And one goal of HadioFS is to provide HDFS this capability. Therefore, in this section, we investigate the flexibility of HadioFS on the utilization of different I/O methods.

ADIOS can use POSIX-IO functions to access disk as well. However, different from the original HDFS, which also uses this I/O method as default, HadioFS can achieve performance benefit from the asynchronous store and close operations. Figure 5.6 shows that the performance of POSIX-based HadioFS is better than that of HDFS; and very close to that of the MPI-based HadioFS. *NULL* is a special method. With it, ADIOS drops the data to be written immediately without touching the disk. The performance of *NULL* can be regarded as the upper bound of performance.

ADIOS also supports some other I/O systems or techniques, such as Dataspaces, DIMES, DataTap, MPI-AIO and so on [1]. But most of them are not available in the public version of ADIOS. In the future work, we would like to evaluate if HadioFS can take most of them.

Chapter 6
Related Work

In this chapter, we elaborate some notable research endeavors closely related to this work either on the issues tackled or on the methods applied. Specifically, in Section 6.1, we describe the research efforts conducted to improve the Hadoop stack, including HDFS and the MapReduce processing engine; and Section 6.2 briefly summarizes the research works, which improve ADIOS and leverage it to optimize the overall I/O performance of applications.

6.1 Hadoop Improvement

As the storage backend for the Hadoop stack, HDFS has been studied actively recently to improve its performance, scalability and hence the upper data processing components, like the MapReduce engine, Hive and etc. Islam *et al.* [27] proposed to improve the performance of HDFS with Remote Direct Memory Access (RDMA) over InfiniBand via JNI interface. Their solution was to enhance the suboptimal communication implementation based upon Java-socket interface. And they suggested leveraging SSD to ensure that the storage modules within HDFS can catch up with the InfiniBand-enabled transmission modules. While, our work, utilizing the high performance I/O methods to accelerate the storage functions within HDFS via ADIOS, is complementary to their work. Shafer *et al.* [33] evaluated the tradeoffs between portability and performance inside the Hadoop distributed filesystem profoundly. They claimed that HDFS, implemented in Java, could not exploit performance-enhancing features of the native platform, such as bypassing the kernel page cache, direct I/O and exploiting OS-specific calls to manipulate file lock at the inode level. However, HadioFS, by offloading the disk I/O workload to ADIOS, which can be tuned or extended flexibly according to the underlying platform, is capable to take advantage of these system-dependent features to boost performance. Another branch of research looks at modifying the architecture of HDFS to improve its scalability. As stated by Konstantin V. Shvachko in [34], the single NameNode becomes a performance bottleneck for Hadoop cluster linear scaling. And the way to tackle this issue is to distribute the name space service onto multiple nodes, like what has been applied to Ceph [39], Lustre [9] and recently extended GFS [26]. There are also researchers trying to integrate some of these more scalable file systems into Hadoop stack directly [31].

There has been a great amount of work to optimize the MapReduce data processing framework since the release of Hadoop [2], an open source implementation of the MapReduce programming model [24]. Most of the endeavors are concerned to speedup the MapReduce pipelining, or ensure the fairness and efficiency of the task scheduling. Recalled, Hadoop MapReduce has three phases, i.e. map, shuffle and reduce. The first two phases can overlap with each other very well. But the final phase is blocked by the previous one, because a ReduceTask cannot start the reduce function until all its input data segments are fetched and sorted locally. Condie *et al.* [22] proposed MapReduce-Online to overlap the last two phases. The intermediate data generated by MapTasks are pushed aggressively to the ReduceTasks, which begin processing even before the completion of all the MapTasks. Though all three phases are overlapped, the early returned result is just estimation of the final one. Wang *et al.* [36] proposed Hadoop-A to pipeline the shuffle and reduce phases by leveraging an RDMA-enabled network-levitated merge algorithm. Though data shuffling cannot begin until all the MapTasks complete, the JVM-Bypass techniques [38] designed by them can enable very fast data movement over high performance networking, like InfiniBand or 10 Gigabit Ethernet. We are inspired by the JVM-Bypass techniques when gluing ADIOS and HDFS together. To better utilize the resources in Hadoop cluster, researchers have proposed many novel task scheduling techniques. Delay scheduler [40] accelerates the execution of MapReduce jobs by prioritizing the assignment of MapTasks with better data-locality, which means the input of the task is stored on or close to the node where it runs. Wang *et al.* [37] observed that long running ReduceTask might occupy the reduce slot on TaskTracker but does nothing, hence wasting the cluster resources. Thus, they have proposed Fast Completion Scheduling [37] to mitigate this monopolizing behavior. And if some task is preempted too many times, it will be prioritized to avoid starvation issue. Having clarified task interference and excessive disk I/O issues within the existing Hadoop MapReduce framework, Li and Wang *et al.* [28] designed CooMR to speedup the disk and network I/O operations related to the intermediate data management. Specifically, they have implemented three main features in C libraries and hooked them into the MapReduce layer via JNI. We borrow much experience from this work when implementing the JNI-related modules.

6.2 ADIOS Application and Improvement

ADIOS has emerged as a popular I/O middleware in High Performance Computing field recently because of its simplicity, extensibility and efficiency. Liu *et al.* [29], based upon the analysis of the communication and I/O issues that keep scientific applications from fully utilizing the storage capability, proposed to leverage ADIOS to parallelize the I/O framework inside GEOS-5, a representative climate and earth modeling application. They eliminated the network aggregation operation by letting each participating process write its own data to a shared file in the parallel file system and took use of the asynchronous I/O of ADIOS to improve the parallel data dumping. Because of the utilization of ADIOS, the enhanced I/O modules of GEOS-5 can be optimized for specific platforms without source code modification and recompilation. Cummings *et al.* [23] proposed to develop an End-to-End framework for fusion simulation. ADIOS, as one of the cornerstones within this framework, provides not only high I/O performance and configurability but also rich data annotation for post-processing. Specifically, DART [25], implemented with RDMA to enable efficient asynchronous I/O, was set as the I/O method for ADIOS. Extending ADIOS is straightforward due to its open extension framework. But most efforts have concentrated on the writing issues of ADIOS. Although efficient at data writing, ADIOS could not enhance the reading performance very well. But recently, many research efforts have been conducted to improve its reading functions, which is particularly important to accelerate the data post-processing stage. Tian *et al.* [35] proposed STAR scheme to enable high speed data query to the large dataset generated by scientific application, which commonly consists of vast amounts of small data elements along various spatial and temporal dimensions. STAR scheme is applied to data elements on the fly. Before being stored, variables are re-orgnaized according to their spatial and temporal relationship. STAR incurs negligible overhead on writing performance by leveraging the staging features of ADIOS. The reorganized data can support efficient and diverse queries, enabling fast post-processing. As our future work, we would like to characterize the read operation of HDFS and optimize the reading mechanism of ADIOS accordingly.

Chapter 7
Conclusion and Future Work

Hadoop is a successful open source implementation of the MapReduce programming model. As the computation capacity of modern commodity machines continues to grow, for Hadoop to fully utilize this power, it is much-needed to accelerate the storage backend of the massive data processing cluster – Hadoop Distributed File System (HDFS). Thus, we propose HadioFS to enhance HDFS with Adaptive I/O System (ADIOS), which supports many high performance I/O techniques, such as data staging, asynchronous I/O and collective I/O; and enables HDFS to select optimal I/O routines and parameter values for a particular platform without source code modification and re-compilation. Accordingly, we customize ADIOS into a chunk-based storage system, encapsulate it to expose POSIX-compatible APIs and utilize JNI to integrate HDFS and the tailored ADIOS together. Overall, our method is feasible and can improve the performance of HDFS via using the advanced I/O methods.

Current system is a prototype to verify our conceptual design to leverage ADIOS to migrate Hadoop onto high performance cluster. Although the general framework is essentially completed, to make it fully functional, a great deal of additional work remains to be performed. Specifically, the reading performance should be accelerated; and parallel I/O capacity within one process should be enabled. In addition, the other high performance I/O methods, such as Dataspaces, DIMES, DataTap and MPI-AIO supported by current version ADIOS, should be investigated within the new framework. After all, HadioFS is helpful to drive the research work on the fusion of Big Data and scientific computing fields further.

Bibliography

[1] Adios users manual. http://users.nccs.gov/ pnorbert/ADIOS-UsersManual-1.5.0.pdf.

[2] The apache hadoop project. http://hadoop.apache.org/.

[3] Flume. http://flume.apache.org.

[4] Hbase. http://hbase.apache.org.

[5] Hdf-5. http://www.hdfgroup.org/HDF5.

[6] Hive. http://hive.apache.org.

[7] Java native interface JNI. http://docs.oracle.com/javase/6/docs/technotes/guides/jni.

[8] Jni directbytebuffer.

[9] Lustre file system. http://www.lustre.org.

[10] Mapreduce 2.0 (yarn). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[11] Mpi and mpi-io. http://beige.ucs.indiana.edu/I590/node52.html.

[12] netcdf. http://www.unidata.ucar.edu/software/netcdf.

[13] Oozie. http://oozie.apache.org.

[14] Pig latin. http://pig.apache.org.

[15] Pivotal hd: Hawq. http://www.emc.com/about/news/press/2013/20130225-04.htm.

[16] Posix write. http://linux.die.net/man/2/write.

[17] R connector. https://blogs.oracle.com/R/entry.

[18] Sqoop. http://sqoop.apache.org.

[19] Zookeeper. http://zookeeper.apache.org.

[20] Hasan Abbasi, Matthew Wolf, and Karsten Schwan. Live data workspace: A flexible, dynamic and extensible platform for petascale applications. In *2007 IEEE International Conference on Cluster Computing*, Cluster'07. IEEE, 2007.

[21] Jonathan Appavoo, Volkmar Uhlig, Amos Stoess, Jan Waterlandy, Bryan Rosenburgy, Robert Wisniewskiy, Dilma Da Silvay, Eric Van Hensbergeny, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10. ACM, 2010.

[22] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[23] Julian Cummings, Jay Lofstead, Karsten Schwan, Alexander Sim, Arie Shoshani, Ciprian Docan, Manish Parashar, Scott Klasky, Norbert Podhorszki, and Roselyne Barreto. Effis: an end-to-end framework for fusion integrated simulation. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP'10. IEEE, 2010.

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

[25] Ciprian Docan, Manish Parashar, and Scott Klasky. Dart: A substrate for high speed asynchronous data io. In *The International ACM Symposium on High-Performance Parallel and Distributed Computing*, HPDC'08. ACM, 2008.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP'03. ACM, 2003.

[27] Nusrat S. Islam, Md W. Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'12. ACM, 2012.

[28] Xiaobing Li, Yandong Wang, Yizheng Jiao, Cong Xu, and Weikuan Yu. Coomr: Cross-task coordination for efficient data management in mapreduce programs. In *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13. ACM, 2013.

[29] Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu, Carlos A. Cruz, Shujia Zhou, Tom Clune, and Scott Klasky. Profiling and improving i/o performance of a large-scale climate scientific application. In *International Conference on Computer Communications and Networks*, ICCCN'13, 2013.

[30] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In

Proceedings of the 6th international workshop on Challenges of large applications in distributed environments, CLADE'08, pages 15–24. ACM, 2008.

[31] Carlos Maltzahn, Esteban Molina-Estolano, Amandeep Khurana, Alex Nelson, Scott Brandt, and Sage A. Weil. Ceph as a scalable alternative to the hadoop distributed file system. ;login'10, 2010.

[32] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. FAST'02. USENIX Association, 2002.

[33] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'10. IEEE, 2010.

[34] Konstantin V. Shvachko. Hdfs scalability: The limits to growth. ;login'10, 2010.

[35] Yuan Tian, Zhuo Liu, Scott Klasky, Bin Wang, Hasan Abbasi, Shujia Zhou, Norbert Podhorszki, Tom Clune, Jeremy Logan, and Weikuan Yu. A lightweight i/o scheme to facilitate spatial and temporal queries of scientific data analytics. In *2013 IEEE Symposium on Massive Storage Systems and Technologies*, MSST'13. IEEE, 2013.

[36] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[37] Yandong Wang, Jian Tan, Weikuan Yu, Xiaoqiao Meng, and Li Zhang. Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, June 2013.

[38] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'13. IEEE, 2013.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'06. ACM, 2006.

[40] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 265–278, New York, NY, USA, 2010. ACM.