JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.16, NO.6, DECEMBER, 2016
https://doi.org/10.5573/JSTS.2016.16.6.880

ISSN(Print) 1598-1657
ISSN(Online) 2233-4866

Adaptable I/O System based I/O Reduction for Improving the Performance of HDFS

Jung Kyu Park, Jaeho Kim, Sungmin Koo, and Seungjae Baek*

Abstract—**In this paper, we propose a new HDFS-AIO framework to enhance HDFS with Adaptive I/O System (ADIOS), which supports many different I/O methods and enables applications to select optimal I/O routines for a particular platform without source-code modification and re-compilation. First, we customize ADIOS into a chunk-based storage system so its API semantics can fit the requirement of HDFS easily; then, we utilize Java Native Interface (JNI) to bridge HDFS and the tailored ADIOS. We use different I/O patterns to compare HDFS-AIO and the original HDFS, and the experimental results show the design feasibility and benefits. We also examine the performance of HDFS-AIO using various I/O techniques. There have been many studies that use ADIOS, however our research is expected to help in expanding the function of HDFS.**

Index Terms—**HDFS, ADIOS, JNI, HADOOP, GFS**

I. INTRODUCTION

With the advent of the Big Data era, an overwhelming amount of data can be generated in our daily life by a wide range of computing facilities, from smart phones and wearable computing devices to high-end scientific computing clusters and giant data centers enabling world-wide media and social networking services [14]. To extract meaningful knowledge and economic value from massive-scale data, MapReduce has evolved as the main processing framework since its introduction by Google in around 2004 [1]. Inspired by the map and reduce functions commonly used in functional programming language, the Google MapReduce programming model inherits the parallelism characteristic 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), the distributed storage system inside the framework [2]. Its simple yet expressive interfaces, efficient scalability, and strong fault tolerance have motivated a growing number of organizations to build their services on the MapReduce framework.

The success of Google MapReduce in the Big Data era motivates the development of Hadoop MapReduce, the most popular open-source implementation of MapReduce, and Hadoop Distributed File System (HDFS), the counterpart of GFS [3]. 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 users define according to particular analysis requirements.

Large-scale scientific applications (e.g. global warming modeling and combustion simulation programs) often generate an extremely massive volume of data [15, 16]. The gap between the I/O speed and computing power of high-end clusters motivates many research efforts on the improvement of storage techniques [5]. However, these techniques are often based upon underlying system supports; hence, they are not always compatible with each other. Therefore, the application using one particular I/O technique has to be modified when ported to another platform, and the cost to change

Manuscript received Oct. 10, 2016; accepted Dec. 12, 2016
Seungjae Baek is with the Korea Institute of Ocean Science and Technology
E-mail : baeksj@kiost.ac.kr

the long-term developed and optimized scientific program might be high. This issue is closely related to what we just stated about HDFS. To address this, Adaptive I/O System (ADIOS) has been designed [4]. ADIOS, as 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 ADIOS interfaces for applications to use are as simple as POSIX ones, although not compatible; new storage systems or techniques can be hooked into them very easily. ADIOS has been widely adopted in the 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 HDFS-AIO framework 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 to enable HDFS 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 HDFS-AIO and the original HDFS. For the current system prototype, the data-writing performance can be improved by up to 10%. In addition, we analyze the performance of HDFS-AIO configured with different I/O methods (e.g. POSIXIO and MPI-IO) to evaluate if HDFS can benefit from the edibility of ADIOS.

The rest of the paper is organized as follows. Chapter II provides the background for this work. We then describe customizing ADIOS and integrating it with HDFS via JNI in Chapter III. Chapter IV analyzes the experimental results. Finally, we conclude the paper.

II. RELATED WORKS

In this chapter, we describe the background of this work. First, 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 of the runtime mechanism of HDFS for data reading and writing, we introduce ADIOS in terms of its architecture and data file structure. Finally, we discuss Java Native Interface (JNI), which is used in our system to integrate HDFS and ADIOS.

1. Hadoop

The Hadoop framework is designed for data-intensive distributed applications. Essentially, it implements the computational model MapReduce, in which each job is divided into many parallel tasks assigned to a cluster of nodes [1]. These tasks are categorized into two types: MapTask and ReduceTask. These are responsible for the execution of user-defined map and reduce functions to process data in an embarrassingly parallel manner. Loss of data and computation failure due to system glitches are common in large-scale distributed computing scenarios. Therefore, to make Hadoop easy to program, the reliability issues of both computation and data are handled within the framework transparently and hidden from the application programmers.

To achieve the required core function and ease of programmability, several subsystems are provided within the whole Hadoop Ecosystem. The subsystem, Hadoop MapReduce, implements the data-processing framework, which encapsulates the computational model MapReduce. One JobTracker and many TaskTrackers are present in this layer. To be specific, the JobTracker accepts a job from a client, divides the job 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 slave node, take full control of the node-local computing resource via slot abstraction. Two kinds of slots are defined: map slots and reduce slots. On each TaskTracker, the numbers of both slots are configurable. Additionally, they can be regarded as static resource containers for executing corresponding tasks: MapTask or ReduceTask. YARN (MRv2) 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 [3]. We discuss its structure in detail here and its runtime feature in the next section. HDFS consists of one NameNode and several DataNodes. The NameNode is responsible for building and managing 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 location, but a set of locations because the file is broken into a list of equal-sized blocks that are perhaps assigned to different DataNodes. Furthermore, on the DataNode, each block is kept as a single file, with a few replicas dispersed on other DataNodes to ensure high data reliability.

2. Hadoop Distributed File System (HDFS)

HDFS plays a critical role in the Hadoop Ecosystem [13]. 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 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, it will acquire the information of a set of DataNodes, which 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 obtain the block. During the read process, no intervention from NameNode is needed, avoiding potential performance bottleneck. In addition, HDFS supports the random seek operation for reads.

To write a file, the client first asks the NameNode to allocate space from the storage cluster to keep the user file. It will receive a list of DataNodes for each file block. Additionally, 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 them to the first DataNode in the pipeline; this DataNode persistently stores each packet and mirrors it in 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 received, the DataNode will notify the upstream DataNode of the success of receiving the 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 the current block are received by all the DataNodes in the pipeline. In contrast with the read operations, HDFS only supports sequential write operations. 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 on the local rack; and the last one on a remote rack.

3. Adaptive I/O System (ADIOS)

ADIOS is highly configurable and lightweight I/O middleware [4]. It is not a runtime system, but a library. Applications need to embed the APIs exposed by this middleware to access the data on the disk. By making use of ADIOS, the application can switch among different I/O methods and tune parameters that might impact I/O performance without source-code modification and recompilation. 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.

However, 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 requirements. On the other hand, to achieve wide adoption in the HPC community, ADIOS supports many advanced I/O methods (e.g. synchronous MPI-IO, collective MPI-IO, and asynchronous I/O) using the DataTap system [8]; many high-performance file systems (e.g. GPFS [9] and Lustre [10]); as well as several different data formats (e.g. NetCDF, HDF-5, and its native BP format [4]), which will be described later. However, it should be noted that though ADIOS APIs are as simple as POSIX ones, they are not POSIX-compatible, which makes this work very challenging.

III. Design and Implementation

In this section, we start with the description of our new framework, HDFS-AIO, which is designed to enhance HDFS by offloading its disk I/O to ADIOS. Then, we propose an 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

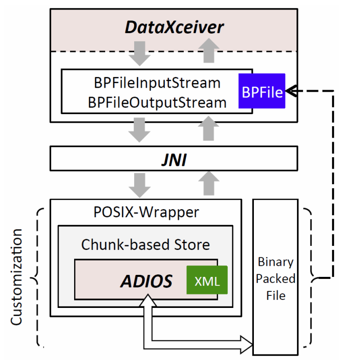

Fig. 1. Architecture of HDFS-AIO.

this chapter, we summarize the constituent components, which HDFS needs to leverage the ADIOS functions.

Previous work also used ADIOS framework for improving the performance of HDFS [17]. However, the difference of work is that the study divided big data file to several files and then has read the file with I/O stream. We used the scheme that enables HDFS to store data blocks into BP file as variables.

1. The Architecture of HDFS-AIO

Fig. 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.

The layer inside HDFS includes the components encapsulating the BP file and its I/O streams. In the existing implementation of HDFS, the file for storing the data block is abstracted as a plain byte stream. 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 a BP file via a new File(), the operations, such as obtain file size or seek in file, will behave abnormally. Additionally, the 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 the BP file.

However, it is not sufficient to use 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 a byte stream-based file, because the runtime states, like r/w pointers, are supposed to be maintained in a manner matching 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 native ADIOS APIs into POSIX-compatible ones, which enable the implementation of the Java-side relevant objects mentioned above. The gap of the interfaces’ semantics between ADIOS and POSIX makes it challenging to accomplish this transformation. We use the write operation as an example to describe this issue.

In the POSIX standard, the write interface is specified as “*ssize t write (int fd, const void *buf, size t count),*” which means writing up to count bytes from the buffer pointed by buf to the file referred to by the file descriptor fd [11]. The amount of bytes to store cannot be known before the interface is called. Additionally, 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)*” [1]. hdlr is a handler pointing to an internal data structure, which includes the context information (e.g. 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 the *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 the *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.

The read operation has a similar difference (i.e. POSIX has an explicit reading pointer, while ADIOS locates content by variable names). In fact, simply speaking, our method for handling the semantics gap is to tailor ADIOS to work like it has 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

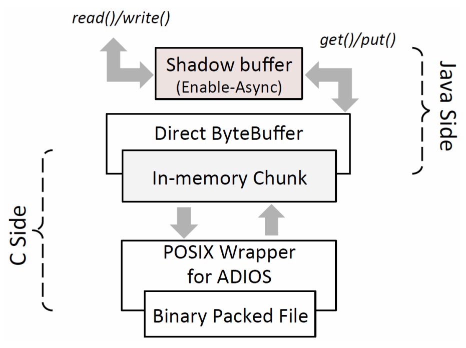

Fig. 2. Implementation details of HDFS-AIO.

support the 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 given in Section 3.2.

2. Implementation Details

We have implemented the aforementioned two layers and made use of JNI to bridge them together, as shown in Fig. 1 [12]. The enhanced HDFS can access data via ADIOS. To leverage high-performance I/O methods (e.g. 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.

As shown in Fig. 2, we use the JNI direct buffer to wrap the in-memory chunk allocated at the C side in a Java-side ByteBuffer object, so that the C-side pointer and Java-side ByteBuffer object refer to the same memory region. In other words, 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. However, the memory region allocated at the C side is outside the JVM heap. Significant overhead will be incurred if the Java program copies data into or out of this region. Therefore, putting (obtaining) small data segments into (from) the direct ByteBuffer object synchronously is very time-consuming. By introducing the shadow buffer, as shown in Fig. 2, we accumulate small data segments into a large one and execute the through-JVM copy asynchronously to eliminate this performance bottleneck from the critical path of HDFS block reading and writing.

All the chunks are also stored and loaded in a dedicated thread asynchronously. When the ADIOS close function is called, all the data still within the memory buffer has to be flushed onto the disk; then, the metadata is updated and stored in a BP file as well. This often makes the ADIOS close function very slow. To reduce the turnaround time of the client, who starts to write the second block only after receiving the acknowledgement that the first block is successfully stored in HDFS, we also asynchronously close the BP file.

The 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 the BlockReceiver, BlockSender, and FSDataset classes, where FileOutputStream and FileInputStream objects are originally constructed and execute the disk access work, respectively. In our implementation, we hybridize them together in such a way that the ADIOS enhancement can be disabled or enabled by the user without changing any source codes.

IV. EVALUATION

1. Experimental Setup

Cluster setup: All the experiments are conducted on a cluster of five 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 one Western Digital SATA hard-drive featuring 250-GB storage space.

Hadoop setup: In all experiments, we use Hadoop version 1.1.2, from which HDFS-AIO 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 experiments.

2. Analysis with Different I/O Patterns

In this section, we investigate the performance of

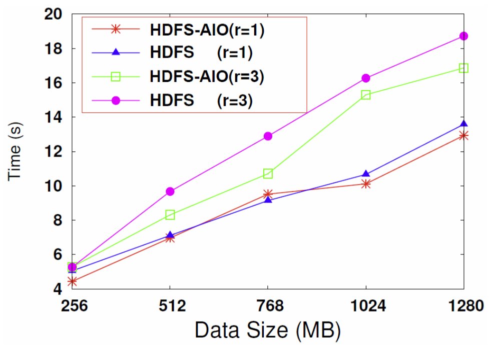

Fig. 3. Performance of single writer.

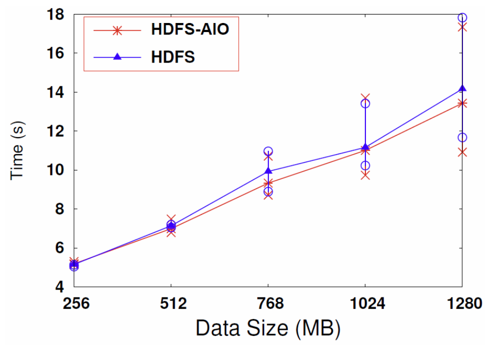

Fig. 4. Performance of multiple writers.

HDFSAIO in comparison with the original HDFS by using different I/O patterns. For the subsequent experiments, the HDFS block size is set to 512 MB; and the I/O method for ADIOS to MPI.

Firstly, we evaluate the writing performance of HDFS-AIO. During experimentation, each pattern has a specific number of writing processes, dataset size, 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 Fig. 3, the performances of HDFS-AIO and HDFS are very close and increase linearly with the size of the dataset. However, HDFS-AIO outperforms HDFS slightly (4%) when the data size increases to 1.2 GB. This is because the asynchronous close operation, designed for HDFS-AIO, can enable the DataNode to notify the client of the completion of writing before the data is even stored onto the 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 Fig. 3, the improvement of HDFS-AIO is enlarged with the increment of replication level. It achieves 10%

Table 1. Detailed profiling for writing

Data Size	Client	Server		
		Write	Close	Store
512 MB	6.530s	0.652s	1.418s	8.602s
1024 MB	10.091s	1.146s	2.279s	20..443s

Table 2. Detailed profiling for reading

Data Size	Client	Server		
		Write	Close	Store
512 MB	7.203s	3.627s	0.427s	2.988s
1024 MB	13.194s	7.597s	4.520s	4.520s

acceleration when writing 1.2 GB of data. This is also attributed to the asynchronous close operation. With it, the downstream DataNode can acknowledge the upstream one prior to finishing the flushing of data buffered in the 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, the more speedup can be gained by HDFS-AIO.

Subsequently, we analyze the "multiple writers" pattern. In contrast with 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 Fig. 4. The average time is calculated among the four parallel writers. As shown, the performance of HDFS-AIO is slightly better than that of HDFS. The contention incurred by the metadata management at the single NameNode offsets the improvement from the asynchronous close operation. For both, this contention also leads to dramatic variance of the execution time among these parallel writers when the dataset size increases.

Next, we evaluate the reading performance of HDFS-AIO. The experiment configuration is similar to that for the writing tests. The chunk size is set to 256 MB. We conduct the "single reader" test. As shown in Fig. 5, the performance of HDFS-AIO is much worse than that of HDFS. To determine why HDFS-AIO is so slow for

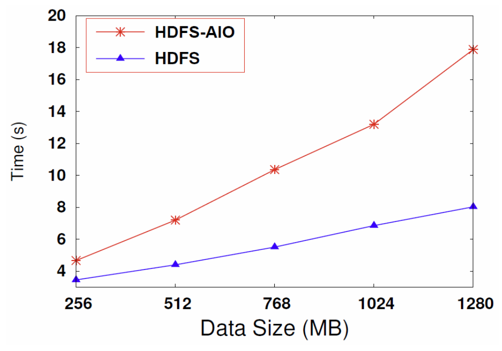

Fig. 5. Performance of single reader.

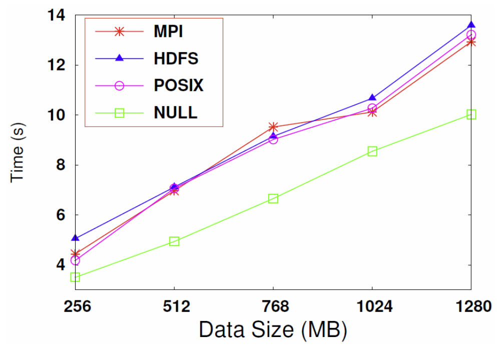

Fig. 6. Writing performance of different I/O methods.

reading, we dissect its execution time in Table 1, as well as its writing execution time.

The total execution time instrumented by the client is placed in the Client column. The execution time of the operation that is executed on the DataNode is recorded in the Server column. Specifically, the asynchronous data storing and loading times are in the Store and Load columns, respectively. As shown in Table 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 the disk asynchronously when it becomes full. However, as shown in Table 2, the time spent on the read operation, which obtains data from the shadow buffer, is longer than that of data prefetching.

This means the normal data reading flow is blocked by the asynchronous chunk loading flow often if not always, even though 256-MB chksize can provide the best chance for pipelining. Obtaining 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, as the BP file needs to be closed to commit the completion of the read operation before the variable value can be used. This inherent restriction makes each chunk loading very time-consuming, which finally leads to inefficient HDFS block retrieval. We will address this restriction in our future work to improve the reading performance.

3. Analysis with Different I/O Methods

An application using ADIOS is capable of switching I/O methods without re-compilation. Additionally, one goal of HDFS-AIO is to provide HDFS with this capability. Therefore, in this section, we investigate the flexibility of HDFS-AIO in the utilization of different I/O methods. ADIOS can use POSIX-IO functions to access the disk as well. However, in contrast with the original HDFS, which also uses this I/O method by default, HDFS-AIO can obtain performance benefits from the asynchronous store and close operations.

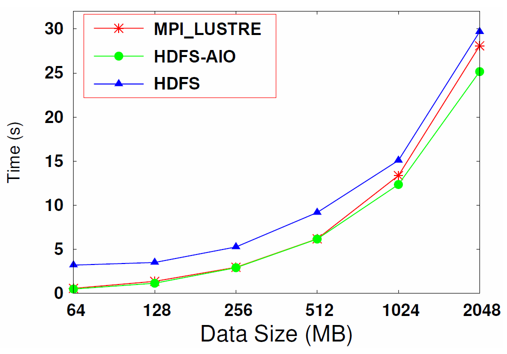

Fig. 7. Writing performance on outside the cluster.

Fig. 6 shows that the performance of POSIX-based HDFS-AIO is better than that of HDFS and very close to that of the MPI-based HDFS-AIO. 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 performance upper bound. ADIOS also supports other I/O systems or techniques (e.g. Dataspaces, DIMES, DataTap, and MPI-AIO [6]). However, most are not available in the public version of ADIOS. In a future work, we would like to evaluate if HDFS-AIO can support most of them.

We performed experiments to see the overall result. We execute the writing process on the node outside the storage cluster to remove the possibility that data is stored locally. As shown in Fig 7, HDFS0-AIO outperforms HDFS while the data size is changed. Although the improvement ratio decreases as the data

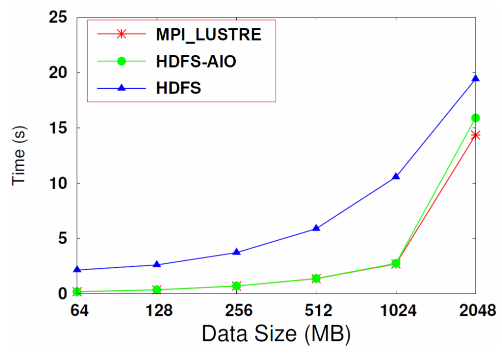

Fig. 8. Reading Performance on outside the cluster.

size grows, HDFS-AIO can still achieve up to 32.8% improvement when writing 512 MB data. This result shows that our scheme is to replace the disk I/O modules within DataNode with our customized ADIOS while, DataNode stores and retrieves data in block unit.

And then we run the reading process on the node outside the storage cluster. The result shows that HSFS-AIO is faster than HDFS by as much as 70%. The reason why the read performance of HDFS-AIO 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 HDFS-AIO when it reads large dataset outside the cluster, as shown in Fig. 8.

V. Conclusion

Hadoop is a successful open-source implementation of the MapReduce programming model. Thus, we proposed HDFS-AIO to enhance HDFS with the Adaptive I/O System (ADIOS), which supports many high-performance I/O techniques (e.g. 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 recompilation.

Accordingly, we customized ADIOS into a chunk-based storage system, encapsulate it to expose POSIX-compatible APIs, and utilize JNI to integrate HDFS and the tailored ADIOS. Overall, our method is feasible and can improve the performance of HDFS by using advanced I/O methods.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0190-15-2012,High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and also supported by SK telecom.

References

[1] J. Dean and S. Ghemawat, “Mapreduce: simplied data processing on large clusters,” *OSDI’04*, vol.6, pp.10-10, 2004.
[2] S. Ghemawat, H. Gobioff, and S. Leung, “The google file system,” *SOSP’03*, pp.29-43, 2003.
[3] The apache hadoop project, http://hadoop.apache.org/.
[4] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki and C. Jin, “Flexible io and integration for scientific codes through the adaptable io system (adios),” *CLADE’08*, pp.15-25, 2008.
[5] Y. Tian, Z. Liu, S. Klasky, B. Wang, H. Abbasi, S. Zhou, N. Podhorszki, T. Clune, J. Logan, and W. Yu, “A lightweight i/o scheme to facilitate spatial and temporal queries of scientific data analytics,” *MSST’13*, 2013.
[6] Adios users manual, http://users.nccs.gov/pnorbert/ ADIOS-UsersManual-1.5.0.pdf.
[7] Mapreduce 2.0 (yarn), http://hadoop.apache.org/docs/current/hadoop-yarn/ hadoop-yarn-site/YARN.html.
[8] H. Abbasi, M. Wolf, and K. Schwan., “Live data workspace: A fexible, dynamic and extensible platform for petascale applications,” *Cluster’07*, 2007.
[9] F. Schmuck and R. Haskin, “Gpfs: A shared-disk file system for large computing clusters,” *FAST’02*, 2002.
[10] Lustre file system, http://www.lustre.org.
[11] Posix write, http://linux.die.net/man/2/write.
[12] Java native interface JNI, http://docs.oracle.com/javase/6/docs/technotes/ guides/ jni.
[13] J. Shafer, S. Rixner, and . L. Cox, “The hadoop distributed filesystem: Balancing portability and performance,” *ISPASS’10*, 2010.

[14] Y. Wang, X. Que, W. Yu, D. Goldenberg and D. Sehgal, "Hadoop acceleration through network levitated merge," *SC'11*, 2011.

[15] Z. Liu, B. Wang, T. Wang, Y. Tian, C. Xu, Y. Wang, W. Yu, C. A. Cruz, S. Zhou, T. Clune, and S. Klasky, "Profiling and improving i/o performance of a large-scale climate scientific application," *ICCCN'13*, 2013.

[16] J. Appavoo, V. Uhlig, A. Stoess, J. Waterlandy, B. Rosenburgy, R. Wisniewskiy, D. D. Silvay, E. V. Hensbergeny and U. Steinberg, "Providing a cloud network infrastructure on a supercomputer," *HPDC'10*, 2010.

[17] Xiaobing, "HadioFS: Improve the Performance of HDFS by Off-loading I/O to ADIOS," *Auburn University*, 2013.

Jung Kyu Park received the M.S. and Ph.D. degrees in computer engineering from Hongik University in 2002 and 2013, respectively. He has been a research professor at the Dankook University since 2014. In 2016, he joined the Research scientist of School of Electrical and Computer Engineering at the UNIST. His research interests include operating system, new memory, embedded system and robotics theory and its application.

Jaeho Kim received the BS degree in information and communications engineering from Inje University, Gimhae, Korea, in 2004, and the MS and PhD degrees in computer science from the University of Seoul, Seoul, Korea, in 2009 and 2015, respect-tively. He is currently a postdoctoral researcher in the School of Electrical and Computer Engineering at UNIST (Ulsan National Institute of Science and Technology), Ulsan, Korea. His research interests include storage systems, operating systems, and computer architecture.

Sung Min Koo received the B.S. degree in the Department of Computer Engineering from Dankook University, Korea, in 2016. He is a student of unified master's and Doctor's course in Dankook University. His research interests include operating system, file system and flash memory.

Seungjae Baek received the BS, MS, and PhD degrees in computer engineering from Dankook University, Yongin, South Korea, in 2005, 2007, and 2010, respectively. He joined Peromnii Inc., Seoul, South Korea, in 2010, as a start-up member, and contributed to the definition and development of instant booting technique. He was a post-doctoral research associate at the University of Pittsburgh, Pennsylvania, prior to joining the faculty of the Dankook University in 2014. He is currently a senior research scientist at Korea Institute of Ocean Science and Technology, from 2016. His research interests include file system, storage device, and operating system itself.