RDMA-based Plugin Design and Profiler for Apache and Enterprise Hadoop Distributed File system

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By
Adithya Bhat, B.E.
Graduate Program in Computer Science and Engineering

The Ohio State University
2015

Master’s Examination Committee:
Dr. D.K. Panda, Advisor
Dr. Feng Qin
Abstract

International Data Corporation states that the amount of global digital data is doubling every year. This increasing trend necessitates an increase in the computational power required to process the data and extract meaningful information. In order to alleviate this problem, HPC clusters equipped with high speed InfiniBand Interconnects are deployed. This is leading to many Big Data applications to be deployed on HPC clusters. The RDMA capabilities of InfiniBand have shown to improve the performance of Big Data applications like Hadoop, HBase and Spark on HPC systems. Since Hadoop is Open Source software, many vendors offer their own distribution with their own optimization or added functionalities. This restricts easy portability of any enhancement across Hadoop distributions.

In this thesis, we present RDMA-based plugin design for Apache and Enterprise Hadoop distributions. Here we take existing RDMA-enhanced designs for HDFS write that is tightly integrated into Apache Hadoop and propose a new RDMA based plugin design. The proposed design utilizes the parameters provided by Hadoop to load client and server RDMA modules. The plugin is applicable to Apache, Hortonworks and Cloudera’s Hadoop distribution. We develop a HDFS profiler using Java instrumentation. The HDFS profiler evaluates performance benefit or bottleneck in HDFS. This profiler can work across Hadoop distribution and does not require any Hadoop source code modification.

Based on our experimental evaluation, our plugin ensures the expected performance of up to 3.7x improvement in TestDFSIO write, associated with the
RDMA-enhanced design, to all the distributions. We also demonstrate that our RDMA-based plugin can achieve up to 4.6x improvement over Mellanox R4H (RDMA for HDFS) plugin.

Using HDFS profiler we show that RDMA-enhanced HDFS designs improve the performance of Hadoop applications that have frequent HDFS write operations.
Dedication

Dedicated to my family and friends
Acknowledgments

I would like to express my very great appreciation to my advisor, Dr. D. K Panda, for his guidance throughout the duration of my M.S. study. I am deeply indebted to him for his support. I would like to thank Dr. Feng Qin for agreeing to serve on my Masters Examination committee.

I am also thankful for the National Science Forum for the financial support they provided for my graduate studies and research.

I would also like to acknowledge and thank Dr. Xiaoyi Lu for his valued and constructive suggestions towards this thesis and also for being a great mentor.

I would also like to thank my HiBD colleagues and friends Nusrat Islam, Wasiur (M. W.) Rahman and Dipti Shankar for all the guidance and support they gave me.

I also want to thank all my NOWLAB friends Hari Subramoni, Khaled Hamidouche, Jonathan Perkins, Mark Arnold, Akshay Venkatesh, Jie Zhang, Albert, Jian Lin, Ammar Ahmad Awan, Ching-Hsiang Chu, Sourav Chakraborty and Mingzhe Li for all the support.

I would like to thank my family members for all their love and support.
Vita

2011 .................................................B.E. Computer Science & Engineering
Visveswaraiah Technological University,
Belgaum, India

2011 – 2013 .............................................Software Engineer, Subex Ltd

2013 – 2015 .............................................M.S. Computer Science and Engineering,
The Ohio State University

2014 – 2015 .............................................Graduate Research Associate,
The Ohio State University

Publications

Approach to Exploit RDMA Benefits for Apache and Enterprise HDFS"
The Sixth workshop on Big Data Benchmarks, Performance Optimization, and
Emerging Hardware (BPOE 6), Aug 2015.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Abstract .............................................................................................................................. ii

Dedication ....................................................................................................................... iv

Acknowledgments ............................................................................................................. v

Vita ..................................................................................................................................... vi

List of Tables .................................................................................................................... x

List of Figures .................................................................................................................. xi

Chapter 1: INTRODUCTION ......................................................................................... 1

1.1 InfiniBand - An Overview .......................................................................................... 2

1.2 Overview of Apache and Enterprise Hadoop distribution ........................................ 2

1.3 Overview of RDMA-enhanced HDFS Design ........................................................... 4

1.4 Problem Statement .................................................................................................... 4

1.5 Organization of the Thesis ....................................................................................... 5

Chapter 2: BACKGROUND ............................................................................................ 7

2.1 HDFS Concepts ......................................................................................................... 8

2.1.1 Blocks ................................................................................................................... 9

2.1.2 NameNode ........................................................................................................... 9

2.1.3 DataNode ............................................................................................................ 10
2.2 HDFS Write Operation ................................................................. 10

2.3 Existing RDMA enhanced HDFS Designs........................................ 12
  2.3.1 RDMA based HDFS Design [11] .............................................. 12
  2.3.2 Parallel Replication for HDFS Write [12].................................. 13
  2.3.3 SOR-HDFS [13] ................................................................. 15
  2.3.4 Triple-H Design [15]............................................................ 16

Chapter 3: RDMA-based HDFS Plugin Design ..................................... 18
  3.1 Introduction .............................................................................. 18
  3.2 RDMA-based Plugin Design for Apache and Enterprise HDFS .......... 18
  3.3 Performance Evaluation .......................................................... 23
    3.3.1 Experimental Setup .......................................................... 23
    3.3.2 Evaluation with Apache Hadoop Distributions ......................... 24
    3.3.3 Evaluation with Enterprise Hadoop Distributions ..................... 28
  3.4 Summary .................................................................................. 31

Chapter 4: HDFS Profiler .................................................................. 32
  4.1 Introduction .............................................................................. 32
  4.2 Proposed Design for HDFS Profiler ............................................ 32
    4.2.1 Instrumentation Engine ...................................................... 33
    4.2.2 Collection Engine ............................................................. 34
    4.2.3 Parser .............................................................................. 35
    4.2.4 Visualization Engine ........................................................ 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.5 MapReduce-based Parallel Parser</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Profiling Results</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1 Evaluation of Different Parsers</td>
<td>37</td>
</tr>
<tr>
<td>4.3.2 Evaluation of RDMA and IPoIB HDFS Profiler</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>47</td>
</tr>
<tr>
<td>Chapter 5: Conclusion and Future Work</td>
<td>48</td>
</tr>
<tr>
<td>Bibliography</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables

Table 1 Performance comparison of different Parsers--------------------------------------37
List of Figures

Figure 1 Hadoop Ecosystem........................................................................................................... 7
Figure 2 HDFS write operation...................................................................................................... 11
Figure 3 Overview of RDMA based HDFS Design ........................................................................ 13
Figure 4 Overview of Parallel Replication Scheme ......................................................................... 14
Figure 5 Overview of HDFS Write Stages ...................................................................................... 15
Figure 6 RDMA-based HDFS Plugin Overview ........................................................................... 19
Figure 7 HDFS Client side approaches ........................................................................................ 21
Figure 8 Implementation and deployment ....................................................................................... 22
Figure 9 Evaluation of HDFS plugin with Hadoop 2.6 using TestDFSIO ........................................... 25
Figure 10 Evaluation of HDFS plugin with Apache Hadoop 2.6 using data generation benchmarks ........................................................................................................................................ 26
Figure 11 Evaluation of SOR-HDFS RDMA plugin with Apache Hadoop 2.5 using TestDFSIO........................................................................................................................................ 27
Figure 12 Evaluation of Triple-H RDMA plugin with HDP-2.2 using TestDFSIO 29
Figure 13 Evaluation of Triple-H RDMA plugin with HDP-2.2 using different benchmarks ........................................................................................................................................ 30
Figure 14 Performance comparison between Triple-H RDMA plugin and R4H .. 30
Figure 15 Overview of HDFS profiler ............................................................................................ 33
Figure 16 Overview of MapReduce-based Parallel Parser ............................................................... 36
Figure 17 Block Transmission Time for RDMA and IPoIB ............................................................... 38
Figure 18 HDFS Block Write time .................................................................................................. 39
Figure 19 Block Write Time Breaktime at DataNode ........................................ 41
Figure 20 Block Distribution across DataNodes ............................................. 42
Figure 21 Block Distribution across Storage Types ..................................... 43
Figure 22 CPU Usage on DataNodes .......................................................... 44
Figure 23 Memory Usage on DataNodes ..................................................... 45
Figure 24 Disk Usage on DataNodes .......................................................... 46
Chapter 1: INTRODUCTION

The advances in technology and human lifestyle has caused an increase in data generation at an unprecedented rate. This rapid data generation is outpacing computation capabilities in many scientific domains adding data intensive computing to their list [1]. The International Data Corporation (IDC) [2] defines digital universe comprising of data from diverse sources, data being as simple as images and videos from an average person’s phone to complex scientific data gathered from Large Hadron Collider at CERN [3]. But only a fraction of this digital universe has been explored for analytic value as it is getting hard to scale computing performance using enterprise class servers. This is leading Big Data applications to being deployed on HPC configuration. High Performance Data Analytics (HPDA) [4], the term coined by IDC, represents such applications that have sufficient data volumes and algorithmic complexity requiring HPC class systems. HPC systems equipped with InfiniBand offer high bandwidth to meet the communication requirements to transfer large volumes of data across different nodes. InfiniBand has emerged as popular high performance interconnect with more than 50% of the top 500 supercomputers in the world based on it, as of July’15 [5].
1.1 InfiniBand - An Overview

InfiniBand (IB) architecture [6] provides high throughput and low latency with RDMA capabilities. It is used to establish connection between processing nodes, as well as between processing nodes and storage nodes via point-to-point bi-directional serial link. Every node that connects to InfiniBand subnet has one or more Host Channel Adapter (HCA). These HCAs are connected to host node via PCI Express slot and have InfiniBand network ports that connect to InfiniBand subnet. InfiniBand provides Remote Direct Memory Access (RDMA) technology. Using RDMA, an application in one node can send data to an application in another node without any OS intervention and with minimal CPU usage on both sides. User level protocol offered by InfiniBand provides interface to use IB without any change at the application layer. Transmitting IP packets over IB (IPoIB) is one such user level protocol. Here IP packets are encapsulated over InfiniBand packet via network interface.

1.2 Overview of Apache and Enterprise Hadoop distribution

Hadoop is an open source software written in Java used for distributed data processing and storage. Hadoop Distributed File System (HDFS) is the primary storage of a Hadoop cluster and the basis of many large-scale storage systems that is designed to span several commodity servers, Hadoop Distributed File System (HDFS) serves as efficient and reliable storage layer that provides high throughput access to application data. Hadoop 2.x has made fundamental architectural changes to its storage platform, with the introduction of HDFS federation, NameNode high availability, Zookeeper based fail-over controller, NFS support, and support for file
system snapshots. While improvements to HDFS speed and efficiency are being added on an ongoing basis, different releases of the Apache Hadoop 2.x distribution have introduced significantly different versions of HDFS. For instance, Apache Hadoop 2.5.0, which is a minor release in the 2.x release includes some major features, such as specification for Hadoop compatible file system, support for POSIX-style file system attributes, while the Apache Hadoop 2.6.0 includes transparent encryption in HDFS along with a key management server, work-preserving restarts of all YARN daemons, and several other improvements.

Hadoop has become the platform of choice for most businesses that are looking to harness the potential of Big Data. Enterprise versions like Hortonworks [7], Cloudera [8], and others have emerged to simplify working with Hadoop. Hortonworks Data Platform [7] (HDP) provides an enterprise ready data platform for a modern data architecture. This open Hadoop data platform incorporates new HDFS data storage innovations in its latest version HDP 2.2, including heterogeneous storage, encryption, and some operational security enhancements. Cloudera Hadoop (CDH) [8] by Cloudera Inc., is a popular Hadoop distribution that provides a unified solution for batch processing, interactive SQL and search, along with role-based access controls, to streamline solving real business problems. Although the core components of these distributions are based on Apache Hadoop, the proprietary management and installation integrated into these distributions make it hard for users to consider trying out other HDFS enhancements. In addition to these integrated solutions, vendors such as Mellanox [9] have introduced solutions to accelerate HDFS, using RDMA-based plugin known as RDMA for HDFS (R4H) [10], that works alongside other HDFS communication layers.
1.3 Overview of RDMA-enhanced HDFS Design

In the literature, there are many works that focus on improving the performance of HDFS operations on modern clusters. In [11], [12], we propose an RDMA-based design of HDFS over InfiniBand that improves HDFS communication performance using pipelined and parallel replication schemes. In these works, the Java-based HDFS gets access to the native communication library via JNI that significantly improves the communication performance while keeping the existing HDFS architecture and APIs intact. SOR-HDFS [13] re-designs HDFS to maximize the overlapping among different stages of HDFS operations using a SEDA [14] based approach. This design proposes schemes to utilize the network and I/O bandwidth efficiently for maximal overlapping. These RDMA-enhanced designs work with different Apache Hadoop versions from 0.20.2 to 2.6.0 and guarantee performance benefits over default HDFS. In order to take advantage of the heterogeneous storage devices available on HPC clusters, we propose a new scheme [15] that accelerates HDFS I/O performance by intelligent placement of data via a hybrid approach. In this work, we propose enhanced data placement policies with hybrid replication and eviction/promotion of HDFS data based on the usage pattern. All these designs are done on top of Apache Hadoop codebase.

1.4 Problem Statement

As the gap between BigData and HPC is blurring, many Hadoop clusters are deployed on HPC configuration with InfiniBand interconnect. From Sections 1.2 and 1.3 we see the following issues:
1. There are many Hadoop vendors that are free to add their own optimization or new features to their Hadoop distribution leading to inconsistencies.

2. Any enhancement designed for one Hadoop distribution cannot be directly utilized by another distribution without code level changes.

3. The benefits of enhanced design may vary across Hadoop distribution due to differences in default environment.

This leads us to the following questions:

1. Can we propose an RDMA-based plugin for HDFS that can bring benefits of efficient hybrid, RDMA-enhanced HDFS designs, to different Hadoop distributions?

2. Can this plugin ensure similar performance benefits to Apache Hadoop as the existing enhanced designs of HDFS, such as [11], [13] and [15]?

3. Can different enterprise Hadoop distributions, such as HDP and CDH also observe performance benefits for different benchmarks?

4. What is the possible performance improvement over existing HDFS plugins such as Mellanox R4H [10]?

5. Can we propose an HDFS profiler that evaluates any performance benefits or bottleneck in HDFS?

6. Can the proposed HDFS profiler work across Hadoop distribution?

We address the above mentioned issues and try to provide our evaluations in the following Chapters.

1.5 Organization of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we provide an overview of HDFS and HDFS write operation along with existing work towards
optimizing HDFS write using RDMA enhanced designs. In Chapter 3, we propose an RDMA-based plugin design for Apache and Enterprise Hadoop distribution and show experimental results for various benchmark using this plugin. In Chapter 4, we give details on design and implementation of HDFS profiler and results of the profiler. Finally we summarize our conclusion and possible future work in Chapter 5.
Chapter 2: BACKGROUND

Hadoop [16] is an open source software project written in Java. The main goal of Hadoop is to store and process large data in distributed fashion using computer clusters along with providing high scalability and high degree of fault tolerance. The following figure shows main components of Hadoop and its ecosystem.

![Hadoop Ecosystem](image)

Figure 1 Hadoop Ecosystem

The main components of Hadoop are Hadoop common, HDFS, YARN and MapReduce.
• Hadoop common – These are general libraries that are core to Hadoop’s working and are utilized by other components of Hadoop.

• HDFS – Hadoop Distributed File System is the core part of Hadoop. This stores large data across multiple storage nodes and serves this data when requested by application.

• YARN (Yet Another Resource Negotiator) – YARN is responsible for managing computing resources in terms of CPU and Memory. These resources are utilized for scheduling users’ application.

• MapReduce – MapReduce is an application for distributed data processing that reads data from HDFS and launches Map/Reduce tasks using YARN to process this data. Even though MapReduce is one of the core components, from the introduction of YARN, it is just an application running on Hadoop.

Along with these core components, many other applications can be installed to make use of HDFS and MapReduce programming paradigm. These installed applications along with Hadoop core components make Hadoop Ecosystem.

2.1 HDFS Concepts

Hadoop Distributed File system is the core distributed storage part of the Hadoop. It has of two types of nodes: NameNode and DataNode. The nodes can be compared to master-worker model with NameNode being the master and DataNode the worker. The following section explains various HDFS components.
2.1.1 Blocks

Files in HDFS are broken into smaller size chunks called Block. The default block size varies across Hadoop versions. Hadoop 2x has default block size of 128 MB. These blocks are replicated across multiple nodes to provide fault tolerance. The replication also helps in improving application read throughput.

2.1.2 NameNode

NameNode is critical component of HDFS. It manages directory tree of all the files in the file system. It has mapping to keep track of all the blocks in a particular file and DataNodes that has these blocks. So if a client has to read an existing file or write a new file to HDFS, it has to contact NameNode first. NameNode responds with an appropriate list of DataNodes servers. As NameNode manages metadata for the entire file system, it is also the single point of failure. Hadoop 2x introduces HDFS high-availability which solves the problem of NameNode being the single point of failure. In high-availability scenario, there are multiple NameNodes. One NameNode acts as active and the rest as standby NameNodes. In the event of failure of an active NameNode, one of the standby NameNodes becomes active and starts serving client requests without any significant interruptions. NameNode stores the metadata for file system in its memory. So, this is another bottleneck as the NameNode might run out of memory when HDFS has large number of files. To avoid this, Hadoop 2x introduces HDFS federation. In HDFS federation, multiple NameNodes can be configured to maintain portion of file system namespace. For example, one NameNode can manage all the files under /order and a second NameNode can maintain all the files under /purchase.
2.1.3 DataNode

DataNode is where blocks are stored for a file. It periodically reports all the blocks it contains to NameNode. Clients can directly talk to DataNode once it receives DataNode server list as a response from NameNode to any file system operation. DataNode's servers can talk to each other to re-balance the block replication. DataNode also sends periodic heart beat messages to NameNode that helps NameNode to keep track of active DataNodes.

2.2 HDFS Write Operation

In this section we explain HDFS write operation as RDMA enhanced HDFS designs target HDFS write. Figure 2 shows the HDFS write operation with three DataNodes.

1. To create file in HDFS, client calls create() on DistributedFileSystem.

2. DistributedFileSystem contacts NameNode. NameNode verifies that file does not already exist and creates an entry for this new file in it’s filesystem metadata.

3. Client gets an instance of DFSOutputStream once the call to create() is successful. DFSOutputStream has methods for communicating with NameNode and DataNode. Client starts writing to file by calling write() on DFSOutputStream.
4. As client writes data, DFSOutputStream splits it into packets of 64KB size and populates it into data queue. DataStreamer thread inside DFSOutputStream consumes data queue. At the beginning of each block DataStreamer contacts NameNode to get the list of DataNode where this new block should be written. The returned DataNode list forms the pipeline. DataStreamer sends the packet to the first DataNode in the pipeline, which stores it and forwards it to the next DataNode in the pipeline. DataStreamer removes the sent packet from data queue and adds it to ack queue.

5. The last DataNode in the pipeline sends acknowledgement to second DataNode for the packet it received. Second DataNode receives the acknowledgement and sends acknowledgement to first DataNode and first
DataNode sends acknowledgement to client. Once the packet is acknowledged by all the DataNodes in the pipeline, it is removed from the ack queue.

6. After all the block of the files are written, client calls close on DFSOutputStream.

7. Once the file is closed, NameNode is notified of the file write completion which returns as long as the blocks in the file are replicated at least once.

2.3 Existing RDMA enhanced HDFS Designs

In this section we take a look at the existing RDMA enhanced designs for HDFS.

2.3.1 RDMA based HDFS Design [11]

This is the first design of RDMA based HDFS write using InfiniBand. Here authors propose [11] a Hybrid design that supports both socket and RDMA based communication. Figure 3 shows multi layer design of RDMA based HDFS write. Client RDMA module is responsible for converting incoming data to packets and sending it to first DataNode in the pipeline. Server RDMA module is responsible for receiving the packet and replicating it to next DataNode in the pipeline if any. RDMA module at both client and server buffer RDMA connection they establish with the DataNode.
It also makes calls to UCR [17] library using JNI interface. UCR library is responsible for IB level communication. Another design optimization that authors have made is instead of acknowledging every packet received at DataNode, only last packet is acknowledged by every DataNode in the pipeline. This is because UCR establishes a RC connection between the nodes and InfiniBand RC protocol guarantees message delivery.

The experimental results reported in [11] show that, for 5GB HDFS file writes, the above design reduces the communication time by 87% and 30% over 1Gigabit Ethernet (1GigE) and IP-over-InfiniBand (IPoIB), respectively, on InfiniBand QDR platform (32Gbps).

2.3.2 Parallel Replication for HDFS Write [12]

From Section 2.2, we saw that HDFS write replicates data to DataNodes in a sequential manner. Figure 4 shows the design of parallel replication for HDFS.
write. In this design DataStreamer writes simultaneously to all the DataNodes returned by the NameNode for block creation. This design applies to socket based design of HDFS explained in Section 2.2 and RDMA based design of HDFS explained in Section 2.3.1. In RDMA based parallel replication, the design uses single connection object per HDFS client but different End Points (EPs) for each DataNode in the replication list. This single connection reduces polling overhead for acknowledgement sent from any of the DataNodes in the replication list.

![Diagram of Parallel Replication Scheme]

Figure 4 Overview of Parallel Replication Scheme

The experimental result reported in [12] show 16% reduction in the execution time of TeraGen benchmark with parallel replication. For TestDFSIO benchmark, 10% and 12% throughput improvement were observed for IPoIB and RDMA respectively.
2.3.3 SOR-HDFS [13]

![Figure 5 Overview of HDFS Write Stages](image)

In this paper [13], the authors propose a new SEDA [14] (Staged Event Driven Architecture) architecture to improve HDFS write performance. This design incorporates previous designs mentioned in Sections 2.3.1 and 2.3.2. Figure 5(a) shows the stages of HDFS write. Each stage has an input queue and thread pool associated with it as shown in Figure 5(b). At any stage, the thread from the thread pool consumes data from input queue, processes it and feeds it to the input queue of next stage. This allows maximum overlapping between different stages of data transfer and I/O. The authors identify read, packet processing, replication and I/O as four major stages of HDFS write.

- **Read** – This state has a buffer pool that is shared by group of end-point in Round-Robin fashion. When an end-point receives data, it selects a free buffer from the buffer pool and copies the data to it. This end-point and buffer-id pair is populated to the next stages of the input queue.

- **Packet processing** - Process Request Controller (PRC) process the input queue containing end-point, buffer-id tuple and assigns a new thread from the pool for new end-point. All subsequent packet for this end-point gets assigned
to the same thread by PRC. This new thread does further processing on the packet in the buffer.

- Replication - Replication Request Controller (RRC) finds a free thread from the thread pool that starts replicating this block. This stage also handles acknowledgement coming from different end-points using responder thread.

- I/O - The I/O request controller selects appropriate I/O threads from the pool. The selection makes sure that data gets written in sequential manner. The selected I/O thread processes its input queue to flush aggregated packets from packet processing stage.

The experimental results reported in [13] show 64% improvement in aggregated write throughput and 37% reduction in job execution time when compared to IPoIB for Enhanced DFSIO benchmark in Intel HiBench.

### 2.3.4 Triple-H Design [15]

In this paper [15], the authors propose design to efficiently utilize heterogeneous storage configuration available on modern HPC clusters. Most HPC clusters are configured with heterogeneous storage comprising of high performance RAM-DISK, near-to-ram fast persistent SSD, and slow but cheap HDD and parallel file system like Lustre. The proposed Hybrid design of HDFS with Heterogeneous storage (Triple-H) uses high speed storage for data buffering and caching thus minimizing I/O bottleneck. Triple-H design has efficient data placement policies that address performance or storage aspect of HPC cluster. In performance sensitive data placement policy, high performance storage devices like RAM-DISK or SSD are used in greedy or load balanced manner to store blocks. Whereas in storage sensitive data
placement policy, blocks are written to parallel file system like Luster that reduces the storage requirements of individual node in the cluster.

Triple-H design also introduces eviction and promotion policies to evict least frequently used blocks to lower level of the storage hierarchy or promote frequently used blocks to higher level of storage hierarchy. This eviction/promotion is decided based on the weights assigned to files that are updated based on access count, access time, and available storage space.

The experimental result reported in [15] show improvement in read and write throughputs of HDFS by 7x and 2x, respectively. Execution time of sort benchmark improves by up to 40% over default HDFS.

In the next chapter, we propose an RDMA-based plugin design that incorporates all the above mentioned HDFS enhancements and can work across Hadoop distributions.
Chapter 3: RDMA-based HDFS Plugin Design

3.1 Introduction

The Hadoop Distributed File System (HDFS) has taken a prominent role in the data management and processing community and it has been becoming the first choice to store very large data sets reliably on modern data center clusters. Many companies, e.g. Yahoo!, Facebook, choose HDFS to manage and store petabyte or exabyte-scale enterprise data. Because of its reliability and scalability, HDFS has been utilized to support many Big Data processing frameworks, such as Hadoop MapReduce [18], HBase [19], Hive [20] and Spark [21]. Such a wide adoption has pushed the community to continuously improve the performance of HDFS [22], [23]. In section 3.2 we propose RDMA-based HDFS plugin and provide implementation detail. Section 3.3 provides performance evaluation of plugin design on Apache, HDP and CDH distribution.

3.2 RDMA-based Plugin Design for Apache and Enterprise HDFS

In this section, we discuss our design of the RDMA-based plugin for HDFS. Hadoop provides server and client side configuration to load user defined classes. By
using such configuration we build RDMA plugin that can work across various Hadoop distribution.

Figure 6 RDMA-based HDFS Plugin Overview

Figure 6 shows how the plugin fits into the existing HDFS structure. At the server side, there is RdmaDataXceiverServer that listens to RDMA connection and processes the blocks received. RdmaDataXceiverServer has to be loaded on every DataNode when it starts along with DataXceiverServer. This can be done by using the DataNode configuration parameter, dfs.datanode.plugins. In order for DataNode
to load class using plugins parameter, it has to implement ServicePlugin interface and override start and stop methods provided by it. RdmaDataXceiverServer utilizes this methodology and implements ServicePlugin. Client side configuration parameter, fs.hdfs.impl, is used to load user designed file system instead of the default Hadoop distributed file system. We create a new file system class named RdmaDistributedFileSystem that extends DistributedFileSystem. We load RdmaDistributedFileSystem using client side parameter. Once RdmaDistributedFileSystem is loaded, it makes use of RdmaDFSClient to load RdmaDFSOutPutStream. RdmaDFSOutPutStream is one of the major component of the RDMA based HDFS plugin. For HDFS write operation, OutputStream class is the base abstract class that other classes need to extend if they wish to send data as a stream of bytes to some underlying link. The FSOutputSummer extends this class and adds additional functionality of generating the checksum for the data stream. DFSOutputStream extends FSOutputSummer. DFSOutputStream takes care of creating packets from the byte stream, adding these packets to the DataStreamer's queue and establishing the socket connection with the DataNode. DFSOutputStream has an inner class named DataStreamer that is responsible for creating a new block, sending the block as a sequence of packets to the DataNode and waiting for acknowledgments from the DataNode for the sent packets. For HDFS write operation, there are a set of common procedures that need to be followed to add the data into HDFS. The procedures are:

1. Get the data from byte stream and convert them into packets
2. Contact the NameNode and get the block name and set of DataNodes
3. Send notification to the NameNode once the file is written completely

To design RdmaDFSOutPutStream, we considered two approaches as shown in Figure 7.
Approach 1

Figure 7 shows the first approach on the left-hand side. Here we create an abstract class AbstractDFSOutputStream which extends FSOutputStream. This abstract class would contain implementation of common methods to communicate with the NameNode. Using this abstract class we can implement RdmaDFSOutputStream that can make use of common communication methods defined in the abstract class and can override methods that require RDMA specific implementation. This is a good approach from the perspective of object oriented design but would require code reorganization to move the existing common methods from DFSOutputStream to AbstractDFSOutputStream and make changes in default DFSOutputStream to make use of this abstract class.

Figure 7 HDFS Client side approaches
Approach 2

Figure 7 shows second and selected approach on the right-hand side. Here we directly extend the existing DFSOutputStream to implement RdmaDFSOutputStream. This approach allows code re-usability by changing the access specifier in DFSOutputStream from private to protected and requires minimal code modification in default Hadoop.

We package these RDMA specific plugin files into a distributable jar along with native libraries which are verb level communication. We need to change the access specifier in DFSOutputStream from private to protected that goes as a patch. So the plugin involves jar, native library, patch and a shell script to install this plugin. This script applies the patch to appropriate source files based on the version of the Hadoop distribution. RDMA specific files are bundled as a jar that needs to be added as a dependency in pom.xml file of Maven build system. This jar also needs to be copied to the Hadoop classpath.

![RDMA plugin features and integration](image1)

(b) Default socket design

Figure 8 Implementation and deployment

Figure 8 shows the RDMA plugin features. The RDMA plugin incorporates RDMA based HDFS write, RDMA-based replication, RDMA-based parallel replication, SEDA designs proposed and Triple-H features in [11]–[13], [15]. Figure 8
also depicts the evaluation methodology. RDMA plugin along with Triple-H [15] design is incorporated into Apache Hadoop 2.6, HDP 2.2 and CDH 5.4.2 code base which are shown as Apache2.6-TripleH-RDMAPlugin, HDP-2.2-TripleHRDMAPlugin and CDH-5.4.2-TripleH-RDMAPlugin legends in Section 3.3. Evaluation of Triple-H with RDMA in integrated mode is indicated by Apache-2.6-TripleH-RDMA. For Apache Hadoop 2.5, we apply the RDMA plugin without Triple-H design as it is not available for this version and this is shown as Apache2.5-SORHDFS-RDMAPlugin legend in Section 3.3. Figure 8 shows the evaluation methodology for IPoIB where we have used Apache Hadoop 2.6, 2.5, HDP 2.2 and CDH 5.4.2. These are addressed as Apache-2.6-IPoIB, Apache-2.5-IPoIB, HDP-2.2-IPoIB and CDH-5.4.2-IPoIB legends in Section 3.3.

3.3 Performance Evaluation

In this section, we present a detailed performance evaluation of our RDMA-based HDFS plugin. We apply our plugin over Apache and Enterprise HDFS distributions. We use Apache Hadoop 2.5, Apache Hadoop 2.6, HDP 2.2 and CDH 5.4.2.

We conduct the following experiments:

1. Evaluation with Apache Hadoop Distributions (2.5.0 and 2.6.0)
2. Evaluation with Enterprise Hadoop Distributions and Plugin (HDP, CDH, and R4H)

3.3.1 Experimental Setup

1. **Intel Westmere Cluster (Cluster A):** This cluster consists of 144 nodes with Intel Westmere series processors using Xeon Dual quad-core processor nodes
operating at 2.67 GHz with 12GB RAM, 6GB RAM Disk, and 160GB HDD. Each node has MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces and each node runs Red Hat Enterprise Linux Server release 6.1.

2. **Intel Westmere Cluster with larger memory (Cluster B):** This cluster has nine nodes. Each node has Xeon Dual quad-core processor operating at 2.67 GHz. Each node is equipped with 24 GB RAM and two 1TB HDDs. Four of the nodes have 300GB OCZ VeloDrive PCIe SSD. Each node is also equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces, and are interconnected with a Mellanox QDR switch. Each node runs Red Hat Enterprise Linux Server release 6.1.

In all our experiments, we use four DataNodes and one NameNode. The HDFS block size is 128MB and replication factor is three. All the following experiments are run on Cluster B, unless stated otherwise.

### 3.3.2 Evaluation with Apache Hadoop Distributions

In this section, we evaluate the RDMA-based plugin using different Hadoop benchmarks over Apache Hadoop distributions. We use two versions of Apache Hadoop: Hadoop 2.6 and Hadoop 2.5.

**Apache Hadoop 2.6:** The proposed RDMA plugin incorporates our recent design on efficient data placement with heterogeneous storage devices, Triple-H [15] (based on Apache Hadoop 2.6), RDMA-based communication [11] and enhanced overlapping [13]. We apply the RDMA plugin to Apache Hadoop 2.6 codebase and compare the performances with those of IPoIB. For this set of experiments, we use one RAM
Disk, one SSD, and one HDD per DataNode as HDFS data directories for both default DFS and our designs.

Figure 9 shows the performance of TestDFSIO Write test. As observed from the figure, our plugin (Apache-2.6-TripleH-RDMAPlugin) does not incur any significant overhead compared to Apache-2.6-TripleH-RDMA and is able to offer similar performance benefits (48% reduction in latency and 3x improvement in throughput) like that of Apache-2.6-TripleH-RDMA.
Figure 10 Evaluation of HDFS plugin with Apache Hadoop 2.6 using data 
generation benchmarks

Figure 10 shows the performance comparison of different MapReduce 
benchmarks. Figure 10(a) and Figure 10(b) present performance comparison of 
TeraGen and RandomWriter respectively, for Apache-2.6-IPoIB, Apache-2.6-
TripleH-RDMA, and Apache-2.6-TripleH-RDMAPlugin. The figure shows that, with 
the plugin, we are able to achieve similar performance benefits (27% for TeraGen 
and 31% for RandomWriter) over IPoIB as observed for Apache-2.6-TripleH-RDMA. 
The Triple-H design [15], along with the RDMA-enhanced designs [11], [13], 
incorporated in the plugin, improve the I/O and communication performance, and 
this in turn leads to lower execution times for both of these benchmarks. We also 
evaluate our plugin using the TeraSort and Sort benchmarks. Figure 10(c) and Figure

26
10(d) show the results of these experiments. As observed from the figures, the Triple-H design, as well as the RDMA-based communication incorporated in the plugin, ensure similar performance gains (up to 39% and 40%, respectively) over IPoIB for TeraSort and Sort as Apache-2.6-TripleH-RDMA.

**Apache Hadoop 2.5:** In this section, we evaluate our plugin with Apache Hadoop 2.5. Triple-H design is not available for this Hadoop version. Therefore, we evaluate default HDFS running over IPoIB, and compare it with RDMA-enhanced HDFS [13] used as a plugin. In this set of graphs, default HDFS running over IPoIB is indicated by Apache-2.5-IPoIB and our plugin by Apache-2.5-SORHDFS-RDMAPlugin.

![Graphs showing latency and throughput](image)

**Figure 11 Evaluation of SOR-HDFS RDMA plugin with Apache Hadoop 2.5 using TestDFSIO**

Figure 11 shows the results of our evaluations with the TestDFSIO write benchmark. As observed from this figure, our plugin can ensure the same level of performance as SOR-HDFS [13], which is up to 27% higher than that of default HDFS over IPoIB in terms of throughput for 40 GB data size. For latency on the
same data size, we observe 18% reduction with our plugin, which is also similar to SOR-HDFS [13].

3.3.3 Evaluation with Enterprise Hadoop Distributions

In this section, we evaluate the RDMA-based plugin using different Hadoop benchmarks over Enterprise Hadoop distributions: HDP 2.2 and CDH 5.4.2. We apply the RDMA plugin to HDP 2.2 and CDH 5.4.2 codebases and compare the performances with those of the default ones.

Figure 12 shows the performance of the TestDFSIO Write test. As observed from the figure, our plugin (HDP-2.2-TripleH-RDMAPlugin) is able to offer similar performance gains as shown in [15] for Apache distribution, to HDP 2.2.

For example, with HDP-2.2-TripleH-RDMAPlugin, we observe 63% performance benefit compared to HDP-2.2-IPoIB in terms of latency. In terms of throughput, the benefit is 3.7x. The TripleH-RDMA plugin brings similar benefits for the CDH distribution as well through the enhanced designs [11], [13], [15] incorporated in it.
In Figure 12(c) and Figure 12(d), we present the performance comparisons for data generation benchmarks, TeraGen and RandomWriter respectively, for HDP and CDH distributions. The figure shows that, with the plugin applied to HDP 2.2, we are able to achieve similar performance benefits (37% for TeraGen and 23% for RandomWriter) over IPoIB, as observed in [15] for Apache distribution. The benefit comes from the improvement in I/O and communication performance through Triple-H [15] and RDMAenhanced designs [11], [13]. Similarly, the benefits observed for CDH distribution are 41% for TeraGen and 49% for RandomWriter.
Figure 13 Evaluation of Triple-H RDMA plugin with HDP-2.2 using different benchmarks

We also evaluate our plugin using the TeraSort and Sort benchmarks for HDP 2.2. Figure 13(a) and Figure 13(b) show the results of these experiments. As observed from the figures, the Triple-H design, as well as the RDMA-based communication incorporated in the plugin, ensure performance gains of up to 18% and 33% over IPoIB for TeraSort and Sort, respectively.

Figure 14 Performance comparison between Triple-H RDMA plugin and R4H

30
Figure 14 shows the TestDFSIO latency and throughput performance between R4H applied to HDP 2.2 and HDP-2.2-TripleH-RDMAPlugin. We use four nodes on cluster A for this experiment as R4H requires Mellanox OFED and cluster B has OpenFabrics OFED. As observed from the figures, for HDP-2.2-TripleH-RDMAPlugin we see up to 4.6x improvement in throughput compared to R4H for 10GB data size. As the data size increases, throughput becomes bounded by disk. Even then the enhanced I/O and overlapping designs of Triple-H and SOR-HDFS that are incorporated in the plugin lead to 38% improvement in throughput for 20GB data size. The improvement in latency is 28% for 20GB and 2.6x for 10GB data size.

3.4 Summary

In this work we have proposed an RDMA-based plugin design for Apache and Enterprise Hadoop distribution. We have proposed two approaches to make use of the functionality provided at the HDFS client side. We go with the second approach as it requires least code modification to implement. We have explained the plugin deployment methodology. We saw that proposed RDMA-based design can give the same benefit as the earlier hybrid RDMA-enhanced design without any performance overhead. We also saw that the proposed RDMA-based plugin design has performance advantage when compared with Mellanox R4H (RDMA for HDFS).
Chapter 4: HDFS Profiler

4.1 Introduction

Majority of application running on Hadoop involve HDFS write operation. MapReduce applications write to HDFS in reduce phase. Spark applications running on top of Hadoop uses HDFS to store RDD. HBase uses HDFS to store records in tables. This makes the performance of HDFS crucial to the running time of the application. Tuning HDFS parameter can have large impact on application running time [24]. With many vendors offering different Hadoop distributions there is a need for HDFS profiler that can work across distribution. Here we propose an HDFS profiler that can evaluate performance benefit or bottleneck in HDFS. The proposed HDFS profiler works across Hadoop distribution without any code changes to Hadoop source code. The rest of the chapter is organized this way. In the 2nd section, we propose the design and implementation of the profiler. In the 3rd section, we discuss the various metrics and graphs generated by the profiler.

4.2 Proposed Design for HDFS Profiler

In this section we discuss the design and implementation details of HDFS profiler. Figure 15 shows major components of the profiler. The HDFS profiler consists four major components, namely instrumentation engine, collection engine,
The following sections describe various components in detail.

![Diagram](image)

**Figure 15 Overview of HDFS profiler**

### 4.2.1 Instrumentation Engine

Instrumentation is the component that collects data from Hadoop cluster. We use Java byte code instrumentation technique to profile HDFS. Because of instrumentation we don’t have to make any Hadoop source code changes. We identify methods and code blocks at HDFS client and server that needs to be profiled and write instrumentation agents for them. These instrumentation agents are loaded along client and server JVM. Agents make use of Javassist [25] library to modify the
bytecode to get necessary data during runtime. The instrumentation engine has two components, namely task agent and system agent.

**Task Agent**

The task agent gets loaded into HDFS client and server JVM. HDFS DataNode server launches at the beginning of Hadoop cluster initialization and server task agent can be loaded at that time. But HDFS client JVM are launched on demand. It can be launched as part of Map/Reduce tasks which have their own JVM or as standalone HDFS client JVM that gets launched during file system operation. We make use of Hadoop parameter mapred.child.java.opts to load client side task agent into Map/Reduce JVM irrespective of the node they get launched in. Client and server task agents generate data that gets written to Hadoop log task output stream.

**System Agent**

System agent is a light weight shell script that collects CPU, memory and disk statistics on every DataNode. We use SAR [26] (System Activity Report) Linux utility to collect these statistics.

**4.2.2 Collection Engine**

The data generated by Instrumentation engine are all written to log output files. Collection engine collects these files along with Hadoop logs and populates it to the node running parser using secured copy protocol. This is a light weight shell script that takes the snapshot of log directories periodically and compares it with previous snapshots. If there are any changed files, they are aggregated and
transferred. Collection engine can be replaced with full-fledged distributed log aggregator like Apache Flume [27], Apache Chukwa [28] or Scribe [29].

### 4.2.3 Parser

Parser is one of the main component of the profiler. This is written in Python. The data and logs collected by collection engine are processed by the parser. The parser splits the processing part into three jobs. One job processes client part of task agent, second job processes server part of task agent and third one processes system agent. The parser aggregates data over blocks, DataNodes, and storage types and feeds to Visualization engine.

### 4.2.4 Visualization Engine

The output of the parser is processed by visualization engine to create appropriate charts and graphs to represent the profiled HDFS data. We use Bokeh [30] which is a visualization library for python. It generates HTML files that can be viewed in web browser.

### Database

Since the output of the HDFS profiler is HTML file, we use documented-oriented database to store the results. We use MongoDB NOSQL [31] database for this purpose. It is easy to store and render the HTML page directly by using document store database.
4.2.5 MapReduce-based Parallel Parser

The single node parser defined in Section 4.2.3 is a bottleneck for the profiler as the profiled data increases for long running jobs. To overcome this performance bottleneck, a new MapReduce-based parallel parser is implemented. Figure 16 shows the overview of MapReduce-based parallel parser. In this approach, parser uses MapReduce programming paradigm to process profiled data in distributed Hadoop cluster. To run MapReduce based parser, the profiled data needs to be in HDFS. Collection engine takes care of this data population. The collection engine gathers profiled data from all the DataNodes and populates into HDFS instead of feeding the collected data to parser directly as in Section 4.2.2.

Figure 16 Overview of MapReduce-based Parallel Parser
4.3 Profiling Results

In this section we explain various results generated by the HDFS profiling tool. All the experiments are run on Cluster B specified in Section 3.3.1.

4.3.1 Evaluation of Different Parsers

In this section we evaluate the performance of single node parser and MapReduce-based parallel parser described in Sections 4.2.3 and 4.2.5, respectively. Table 1 shows the performance comparison of different parsers for 5GB and 40GB TestDFSIO using Apache Hadoop 2.6.0 over IPoIB. We use four DataNodes with one SSD each for this experiment.

<table>
<thead>
<tr>
<th>Test</th>
<th>Log Size</th>
<th>Single Node Parser</th>
<th>MapReduce Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>5GB TestDFSIO</td>
<td>92MB</td>
<td>5.8 Sec</td>
<td>18 Sec</td>
</tr>
<tr>
<td>40GB TestDFSIO</td>
<td>842MB</td>
<td>72.8 Sec</td>
<td>22 Sec</td>
</tr>
</tbody>
</table>

Table 1 Performance comparison of different Parsers

From the table we can see that single node parser performs better than MapReduce-based parallel parser when the input size is low. This is because MapReduce framework itself has some initialization overhead on every job startup. But as the input size for parser increases, we can see that MapReduce-based parallel parser has clear advantage over single node parser as it processes data in distributed fashion.
4.3.2 Evaluation of RDMA and IPoIB HDFS Profiler

In this section we show the results of HDFS profiler for RDMA-based plugin applied to Apache Hadoop 2.6.0 and Apache Hadoop 2.6.0 IPoIB. All the results are obtained using the following configuration, unless stated otherwise. We use four DataNodes, each having one RAM DISK, one SSD, and one DISK. The block size is 128MB and block replication factor is three. We write 5GB file into HDFS using HDFS Put operation.

Block Transmission Time

Block transmission time is the time taken by the block to reach DataNode from client. We use one DataNode with one SSD and write 5GB file into HDFS using HDFS Put operation for this experiment. From Figure 17 we can see that RDMA network latency is low compared to IPoIB latency.

![Block Transmission Time](image)

Figure 17 Block Transmission Time for RDMA and IPoIB
The average network latency for block transmitted using RDMA is 90.25 millisecond compared to 367.77 millisecond for IPoIB. This low latency along with advanced RDMA designs [11], [13], [15] for HDFS improves the overall job execution time.

Block write time

![HDFS Block Write Time](image)

Figure 18 HDFS Block Write time
Figure 18 shows block write time for RDMA and IPoIB for a job. Every point in figure represents a block and the time it took to write that block into HDFS in seconds. The graph also displays the file this block belongs to. The block write time metric is collected at HDFS client and is a measure of time the first packet of the first block is sent to the acknowledgement for the last packet of the last block is received. From Figure 18 we see that block write time for RDMA is more stable compared to IPoIB. Initial blocks for RDMA in Figure 18(a) have higher write time compared to the rest of the blocks. Upon further investigation we found that it is the replication pipeline connection setup time at the DataNode. Once the connection is setup, it gets cached and has no impact on block write time for later blocks. We don’t see major variation in write time for rest of the blocks as most of the blocks are cached in the memory.

**Block write time breakdown**

Figure 19 shows the block write time breakdown in terms of packet processing phase and I/O phase at each DataNode for RDMA and IPoIB. The packet processing phase at DataNode is the time required to parse the header associated with every packet. I/O phase involves the time required to write the data in the packet to output stream plus the time required to flush the output stream.

From the graphs in Figure 19(a), we can see that for RDMA, packet processing phase has constant time for all the blocks, whereas I/O phase is more stable than IPoIB shown in Figure 19(b). The Triple-H [15] and SEDA based [13] designs, incorporated in RDMA, use RAM DISK and SSD as buffer cache for blocks and improve overlapping between various data transfer stages and I/O stage respectively. This reduces the I/O phase time in RDMA compared to IPoIB. IPoIB
in Figure 19(b) uses round robin method for choosing storage media that results in variation in I/O phase.

Figure 19 Block Write Time Breaktime at DataNode
Block Distribution across DataNodes

Figure 20 shows the block distribution across the DataNodes for RDMA and IPoIB. The 5GB HDFS put operation has 40 blocks and with a replication factor of three, the job has a total of 120 blocks.

The chart shows the number of blocks received by each DataNode. For example, DataNode storage04 received 33 blocks. It is evident from the figure that irrespective of the underlying network protocol, blocks are distributed in a similar fashion across DataNodes.

Block Distribution across Storage Types

Figure 21 shows block distribution across different storage media for RDMA and IPoIB. There are 120 blocks in 5GB file. For RDMA, 80 blocks got replicated into RAM DISK and 40 blocks into SSD. This is because of hybrid replication feature.
in [15] that is included in RDMA designs. In hybrid replication two replicas are placed in RAM Disk buffer cache in a greedy manner and the other one in SSD. This provides fault tolerance and low latency. For IPoIB, the blocks were almost equally distributed across storage types as IPoIB selects storage media using round robin methodology.

![Block distribution across Storage types](image)

**Figure 21 Block Distribution across Storage Types**

**CPU Usage**

Figure 22 shows CPU usage of all the DataNodes for RDMA and IPoIB from the time the job was launched to sometime after the job completed. The RDMA has higher CPU usage as compared to IPoIB for the entire duration of the job. This is because RDMA uses polling mechanism to receive packets at DataNode server.
After the job completion, CPU usage for both RDMA and IPoIB drops down. We use SAR [26] utility to capture CPU information. SAR provides -u flag that exposes the CPU utilization in percentage when executing in user space.

**Memory Usage**

Figure 23 shows the memory consumption of DataNodes for RDMA and IPoIB.
Figure 23 Memory Usage on DataNodes

RDMA has higher memory consumption as every DataNode registers some amount of memory at the beginning for RDMA data transfer. SAR -r flag gives memory usage information. The graph shows memory consumption in megabytes (MB) as job progresses.
Disk Usage

Figure 24 shows disk utilization of DataNodes for RDMA and IPoIB.

![Disk Usage for RDMA](image)

(a) Disk Usage on DataNodes for RDMA

![Disk Usage for IPoIB](image)

(b) Disk Usage on DataNodes for IPoIB

Figure 24 Disk Usage on DataNodes

We can see that IPoIB has a higher peak value compared to RDMA as replicated blocks are stored on RAM DISK, SSD and DISK as opposed to RAM
DISK and SSD in RDMA. We use SAR -b flag to exposes the disk I/O related information.

4.4 Summary

In this work we have proposed an HDFS profiler that uses Java Instrumentation to attach to JVM and profile it. We use MapReduce-based parallel parser to speed up the parsing of profiled data. Proposed HDFS profiler does not require any code changes to Hadoop source code. We see that RDMA and HDFS enhanced designs [11], [13], [15] improve the performance of a job. We show various results that are profiled and visualized. These metrics help in profiling different HDFS distributions and give a high level picture of any new performance benefit or bottleneck introduced by the Hadoop vendors in HDFS.
Chapter 5: Conclusion and Future Work

In this thesis, we have proposed an RDMA-based plugin for Hadoop Distributed File System (HDFS), to leverage the benefits of RDMA across different Hadoop distributions, including Apache and Enterprise. This flexible technique provides smart and efficient way to integrate fast data transmission functionalities from existing hybrid RDMA-enhanced HDFS designs, while adapting the general plugin based approach in HDFS, in order to bring the network-level benefits to the end application. We implement our RDMA-based plugin that includes the RDMA enhancements existing in the literature [11], [13] and apply on top of different Hadoop distributions. We then propose an HDFS profiler that shows that RDMA can benefit over IPoIB for block transmission time.

Our experimental results demonstrate that our proposed RDMA-based HDFS plugin incurs no extra overhead in terms of performance for different benchmarks. We observe up to 3.7x improvement in TestDFSIO write throughput, and up to 48% improvement in latency, as compared to different Hadoop distributions running over IPoIB. We also demonstrate that our plugin can achieve up to 4.6x improvement in TestDFSIO write throughput and 62% improvement in TestDFSIO write latency, as compared to Mellanox R4H plugin.

We have currently made RDMA-based HDFS plugin with Triple-H design [15] publicly available as a part of the HiBD project [32], for Apache Hadoop and HDP. In the future, we plan to make the RDMA-based plugin available for other
popular Hadoop distributions such as CDH. We plan to undertake detailed studies to assess the benefits of using the proposed plugin for real-world Hadoop applications. The HDFS profiler tool is available for Apache and HDP distributions. We plan to get the tool working with CDH and make detailed HDFS profiling for various applications across Hadoop distributions.
Bibliography


