2015 44th International Conference on Parallel Processing

Accelerating I/O Performance of Big Data Analytics on HPC Clusters through RDMA-based Key-Value Store*

Nusrat Sharmin Islam, Dipti Shankar, Xiaoyi Lu, Md. Wasi-ur-Rahman and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{islamn, shankard, luxi, rahmanmd, panda}@cse.ohio-state.edu

***Abstract*—Hadoop Distributed File System (HDFS) is the underlying storage engine of many Big Data processing frameworks such as Hadoop MapReduce, HBase, Hive, and Spark. Even though HDFS is well-known for its scalability and reliability, the requirement of large amount of local storage space makes HDFS deployment challenging on HPC clusters. Moreover, HPC clusters usually have large installation of parallel file system like Lustre. In this study, we propose a novel design to integrate HDFS with Lustre through a high performance key-value store. We design a burst buffer system using RDMA-based Memcached and present three schemes to integrate HDFS with Lustre through this buffer layer, considering different aspects of I/O, data-locality, and fault-tolerance. Our proposed schemes can ensure performance improvement for Big Data applications on HPC clusters. At the same time, they lead to reduced local storage requirement. Performance evaluations show that, our design can improve the write performance of TestDFSIO by up to 2.6x over HDFS and 1.5x over Lustre. The gain in read throughput is up to 8x. Sort execution time is reduced by up to 28% over Lustre and 19% over HDFS. Our design can also significantly benefit I/O-intensive workloads compared to both HDFS and Lustre.**

I. Introduction

The outstanding performance requirement in High Performance Computing (HPC) environment pertains to unprecedented demands on the I/O performance of supporting storage systems. As HPC architecture has evolved over the years, there has been a fundamental change in the type of data managed in these systems. Managing the proliferation of digital data in this Big Data era, places a premium on high-throughput, high-availability storage. Recent IDC reports [1, 11] claim that the digital universe is doubling in size every two years and will multiply 10-fold between 2013 and 2020; from 4.4 trillion gigabytes to 44 trillion gigabytes. This phenomenon will present significant challenges not only to manage, store and protect the sheer volume and diversity of the data, but also to operate on the data to perform large scale data analysis.

Over the past few years, Hadoop MapReduce has become one of the most important frameworks for Big Data analytics. Hadoop was initially inspired by Google MapReduce [7] outlining its approach to handle an avalanche of data, and has since become the de facto standard for storing, processing and analyzing hundreds of terabytes, and even petabytes of data [2] in diverse fields including scientific computing and HPC. Hadoop Distributed File System (HDFS) is the primary storage for MapReduce jobs. But HPC clusters follow the traditional Beowulf architecture [29] where the compute nodes are equipped with limited or no local storage space. Current large-scale HPC systems take advantage of enterprise parallel file systems like Lustre that are designed for a range of different applications. Therefore, Hadoop MapReduce jobs are often run on top of the existing installations of the parallel file systems on HPC clusters. For write-intensive applications, running the job on top of Lustre can also avoid the overhead of tri-replication as is present in HDFS [27]. Although parallel file systems are optimized for concurrent access by large scale applications, write overheads can still dominate the run times of data-intensive applications. Besides, there are applications that require significant amount of reads from the underlying file system. The performances of such applications (e.g Grep) are highly influenced by the data locality in the Hadoop cluster [13, 32]. As a result, applications that have equal amounts of reads and writes suffer from poor write performance when run on top of HDFS; whereas, because of inadequate locality, read performs sub-optimally while these applications run entirely on top of Lustre. Such applications need a new design that efficiently integrates these two file systems and can offer the combined benefits from both of these architectures.

A. Motivation

TABLE I: Existing Studies on HDFS

Focus	[4, 5, 12]	[22, 23]	[24, 30]	[13]	This paper
In-Memory I/O	√			√	√
Heterogeneous Storage Support		√		√	√
Parallel File System Integration			√	√	√
Buffering for Parallel File System					√
Multiple Modes Support					√

As shown in Table I, there have been many studies to improve the read performance of HDFS through caching [4, 5], whereas, very few of them shed light on improving the write performance. A recent trace from Cloudera shows that, 34% of the Big Data jobs (weighted by compute time) have outputs as large as their inputs [6]. As stated in [10], HDFS I/O is dominated by the write operation due to the tri-replicated data blocks. Our earlier work [12] has enhanced HDFS Write performance using in-memory techniques. Triple-H [13] improves I/O performance of HDFS through efficiently utilizing the heterogeneous storage architecture on HPC clusters. This work also proposes techniques to utilize parallel file system for

*This research is supported in part by National Science Foundation grants #CNS-1419123 and #IIS-1447804. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.79

HDFS in order to minimize local storage space requirements. Some other studies also incorporate heterogeneous storage media (e.g. SSD) [22, 23] and parallel file systems [24, 30] into HDFS. While all these studies concentrate on improving HDFS I/O performance for Big Data applications, very few focus on eliminating the I/O bottlenecks of parallel file system access for job input-output on HPC environments.

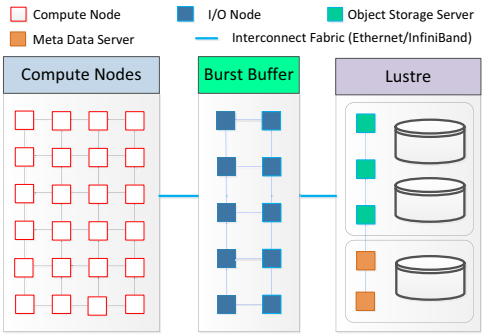


Fig. 1: Burst Buffer System in a typical HPC cluster

In HPC clusters, burst buffer systems are often used to handle the bandwidth limitation of shared file system access [18]; such systems temporarily buffer the bursty I/O and gradually flush datasets to the back-end parallel file system. As demonstrated in Figure 1, burst buffer introduces a tier of solid-state drives into the storage system to absorb application I/O. In HPC systems, burst buffers are typically used to buffer the checkpoint data from HPC applications. Checkpoints are read during application restarts only and thus checkpointing is a write-intensive operation. On the other hand, there are a variety of Hadoop applications; some are write-intensive, some are read-intensive while some have equal amount of reads and writes. Besides, the output of one job is often used as the input of the next one; in such scenarios data locality in the cluster impacts the performance of the latter to a great extent. Moreover, considering the fact that real data has to be stored to the file system through the burst buffer layer, ensuring fault-tolerance of the data is much more important compared to that in the checkpointing domain. All these challenges lead us to the following broad questions:

1) Can we take advantage of high-performance burst buffer system to accelerate I/O performance for Big Data applications on HPC clusters? What are the challenges to integrate HDFS with Lustre through this buffer layer?
2) Can key-value stores such as Memcached be used as the burst buffer? What are the associated challenges here?
3) What are the possible modes that our design can support to guarantee data locality and fault-tolerance for Big Data workloads? Can we propose different schemes for integrating HDFS with Lustre using the buffer layer for deployment across a variety of cluster and system architecture?
4) How much performance improvement can we achieve for Big Data applications with this design on modern clusters?

B. Contributions

In this paper, we propose to integrate HDFS with Lustre through RDMA-based key-value store. In this context, we design a burst buffer system for Big Data analytics applications using RDMA-based Memcached [17, 19] and integrate HDFS with Lustre through this high-performance buffer layer. Considering different aspects of I/O, data locality, and fault-tolerance, our design can support different modes of operations. We also present three schemes for integrating HDFS with Lustre to ensure deployment of our design across a variety of cluster architecture.

We perform extensive evaluations with our design and the results show that our design can improve the performance of TestDFSIO write by up to 2.6x over HDFS and 1.5x over Lustre. The gain in read throughput is up to 8x. The execution time of Sort is reduced by up to 28% over Lustre and 19% over HDFS. The performances of I/O-intensive workloads are also significantly improved by our design.

The rest of the paper is organized as follows. Section II presents the background for this paper. We propose our design in Section III. Section IV describes detailed evaluation. Section V provides related studies that currently exist in literature. We conclude in Section VI, with possible future work.

II. Background

In this section, we provide the necessary background information for this work.

A. Hadoop Distributed File System

Hadoop [31] is a popular open source implementation of the MapReduce [7] programming model. The Hadoop Distributed File System (HDFS) [26, 31] is the primary storage for Hadoop cluster. For most Hadoop applications, it provides the storage for both input and output data. However, HDFS is not used to store intermediate data that are generated in the course of job execution. The intermediate data is kept on each node's local file system.

B. Memcached

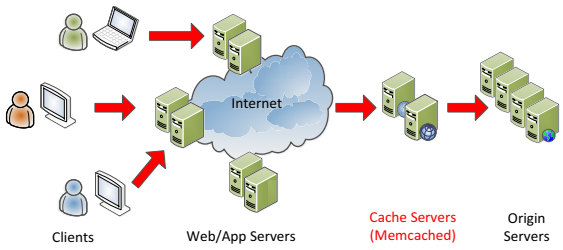
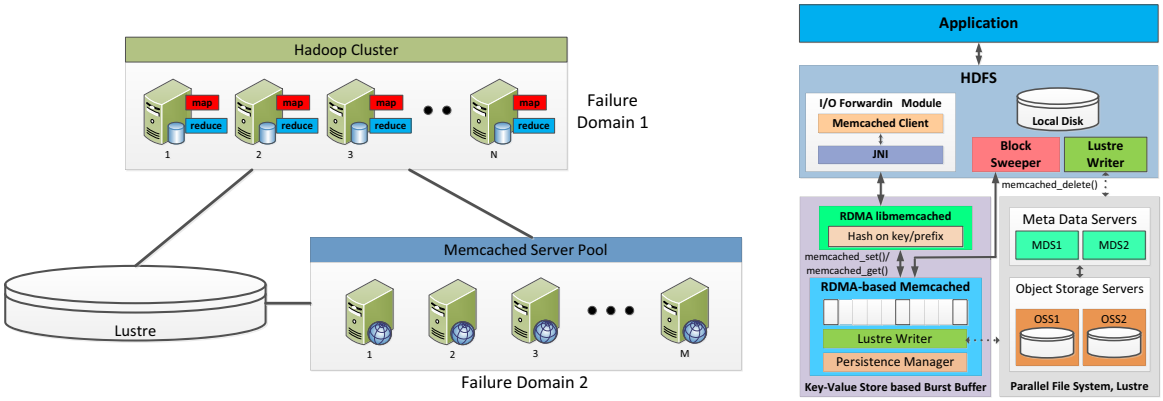


Fig. 2: Memcached in Web 2.0 architecture

Memcached was proposed by Fitzpatrick [8] to cache database request results. It was primarily designed to improve performance of the web site LiveJournal. Due to its generic nature and open-source distribution [3], it was quickly adopted in a wide variety of environments. Using Memcached, spare memory in data-center servers can be aggregated to speedup lookups of frequently accessed information, like database queries, results of API calls, or web-page rendering elements.



(a) Deployment (b) Architecture

Fig. 3: Deployment and Architecture

Figure 2 illustrates how Memcached is deployed in Web 2.0 architecture as a caching layer to improve performance for various client operations. In our previous work [12], we have discussed how key-value stores, such as Memcached, can be deployed with Hadoop systems to enhance the HDFS read/write performance.

III. Design

In this section, we describe our design in detail.

A. Proposed Deployment

Figure 3(a) depicts our proposed deployment to use a key-value store-based burst buffer system to integrate HDFS with parallel file system like Lustre on HPC clusters. In HPC systems, the Hadoop jobs are launched on the compute nodes. The burst buffer pool is deployed as a separate set of servers. In our design, we use Memcached to develop the burst buffer, so the Memcached server daemon runs in each of these servers. The map and reduce tasks of the Hadoop job write their data in the burst buffer layer. The data is also flushed to the parallel file system to guarantee persistence and fault-tolerance. In order to reduce the probability of data loss due to node failure, we propose to deploy the Hadoop DataNodes and Memcached servers in two different failure domains.

The reason that we use a key-value store for buffering HDFS I/O on top of the parallel file system is that key-value stores provide flexible APIs to store the HDFS data packets against corresponding block names (each HDFS block consists of a number of fixed sized packets) [12]. The NameNode generates the block names and they are unique per block. Also, data buffered in this way can be read at memory I/O speed from the key-value store.

B. Internal Architecture

Figure 3(b) demonstrates the internal architecture of our proposed design. We first design a burst buffer system using Memcached and then integrate HDFS with Lustre through this buffer.

1) Key-Value Store-based Burst Buffer System: We use the RDMA-based Memcached to design a burst buffer system. For this, each Memcached server must have a local SSD on it. The burst buffer system has two components: 1) RDMA-based libMemcached (client) and 2) RDMA-based Memcached (server).

HDFS sends data to the Memcached servers through RDMA-based libMemcached. The key is formed by appending the packet sequence number with the corresponding block name and the value is the data packet. Thus, all the packets belonging to the same block have the same key prefix (block name). We maintain the concept of blocks in the Memcached layer by modifying the libMemcached hash function. Instead of hashing on the entire key, we hash only based on the block name prefix of the key. In this way, all the data packets belonging to the same block go to the same Memcached server.

The Memcached server side stores the data sent from the HDFS layer in the in-memory slabs. Data can also be read by the HDFS clients (i.e. map or reduce tasks) from the server. The burst buffer functionalities are introduced in the Memcached server by two main components: `Persistence Manager` and `Lustre Writer`.

Persistence Manager: As demonstrated in Figure 4, the `Persistence Manager` consists of a pool of threads attached to a queue. After completion of each `memcached_set()`, data is put to the queue. Whenever the first packet of a block arrives, a new file is created in the local SSD. This file is identified by the block name prefix in the key of the first packet. The writer threads get the data and the associated file pointer from the queue and write the data to the file. The file pointer is cached in a linked list so that subsequent packets from the same block can be written to this open file. The HDFS client sends the data packets belonging to a block sequentially to the Memcached server through RDMA and when the last packet arrives, the `Persistence Manager` makes sure that the `memcached_set()` returns to the HDFS Client after all the packets including the last one are persisted. At this point, the block file is closed and the file pointer is removed from the linked list. Thus, in addition to having a

persistent file per block in the SSD, the data packets belonging to the blocks are also kept in the in-memory slabs of the Memcached server. The updated data distribution strategy in the libMemcached side helps to preserve the block structure while data is written in the back-end SSDs on the Memcached servers and subsequently, to the parallel file system.

Lustre Writer: The `Lustre Writer` consists of a pool of threads attached to a queue. This module is responsible for forwarding the data to Lustre. In our design, we consider writing to the parallel file system in both asynchronous and synchronous manner. Depending on which mode we choose, the `Lustre Writer` can work accordingly.

In this study, we also consider deployment of our proposed design on different clusters. Therefore, if there is no SSD node on the cluster, the `Persistence Manager` in the Memcached server is not enabled. For such scenarios, we propose different modes to deploy our design. We support these modes via different integration schemes of HDFS with Lustre through the buffer layer. We discuss these schemes in detail in Section III-C. Also, data can be sent to the parallel file system by either the burst buffer or the HDFS layer directly based on the chosen scheme. So, `Lustre Writer` can be enabled either in the Memcached server or in the HDFS layer.

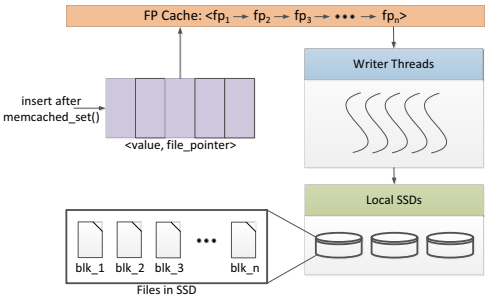


Fig. 4: Persistence Manager

2) Integration of HDFS with Lustre through Key-Value Store-based Burst Buffer: The interaction of HDFS with the buffer layer is controlled by the following two major components as depicted in Figure 3(b):

I/O Forwarding Module: The I/O forwarding module in HDFS Client is responsible for interaction with the Memcached layer. It creates a libMemcached client object and invokes the `memcached_set()` operation via JNI to pass the data to the Memcached server. The JNI layer bridges the Java-based HDFS with the native libMemcached library. While reading the buffered data, HDFS Client uses the `memcached_get()` API.

BlockSweeper: Each DataNode has a `BlockSweeper` that wakes up periodically and removes data from the in-memory slabs of the Memcached servers. Typically, the `BlockSweeper` kicks in at the end of a job or when no job is actively running on the cluster. The interval at which the `BlockSweeper` should wake up is configurable. If the available memory on a Memcached server runs out while a job is running, data eviction takes place following the LRU policy of Memcached.

C. Proposed Schemes to Integrate HDFS with Lustre

Considering the aspects of data locality and fault-tolerance as described in Section I, our design supports three different modes: (1) Asynchronous Lustre Write (ALW), (2) Asynchronous Disk Write (ADW), and (3) Bypass Local Disk (BLD). In this section, we propose different schemes to integrate HDFS with Lustre through the key-value store-based burst buffer system proposed in Section III-B1 to provide support for different modes.

1) Asynchronous Lustre Write (ALW): Figure 5(a) shows the integration scheme to support ALW mode. In this scheme, MapReduce jobs are scheduled on the compute nodes. Each map or reduce task essentially launches an HDFS Client that writes one copy of data to the local storage on the DataNode and the other copy is transferred to the Memcached server through the `I/O Forwarding Module`. The DataNode forwards the data to the parallel file system in an asynchronous manner through the `Lustre Writer`. The DataNode, while writing to Lustre keeps track of the pending writes and when a write is complete, it informs the NameNode of it.

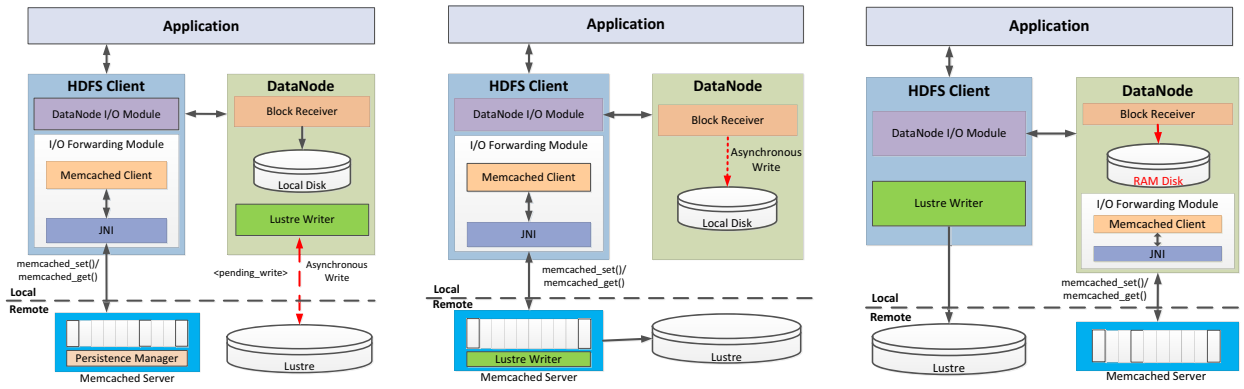
The `I/O Forwarding Module` writes the data packets of HDFS blocks to Memcached. Each Memcached server is backed by SSD or NVRAM. Therefore, on completion of the `memcached_set()` operation, the data is given to the `Persistence Manager` that ensures data persistence by synchronously writing each data to SSD.

In this way, Memcached servers host an SSD copy along with the in-memory copy of a data block. In this scheme, the `Lustre Writer` is activated in the DataNode side. The `Lustre Writer` can also be enabled on the Memcached servers instead of the DataNodes. But the advantage of keeping it in the DataNode is that the DataNode can inform the NameNode on completions of the asynchronous Lustre writes.

The map and reduce tasks of a MapReduce job, read data from the local storage whenever possible. This ensures the benefit of data locality for Hadoop jobs. If, on the other hand, data is not available locally, they read from the in-memory slabs of the Memcached servers through `memcached_get()` operation. In case of failures (data is not available in local storage or Memcached), data is read from Lustre.

This scheme meets the challenges described in Section I as follows:

1) Improves performance by hiding the latency of parallel file system access through the high performance burst buffer system. The Memcached-based burst buffer layer uses RDMA-based communication and SSD-based data persistence.
2) Improves data locality compared to MR jobs running over Lustre by placing data in the local storage of the DataNodes. By reading data from the local storage and Memcached, this design can achieve much better read performance.
3) Provides fault-tolerance by storing one replica in local disk and one in the burst buffer; these two replicas are in two different failure domains. As presented in [36],



(a) ALW (Asynchronous Lustre Write) (b) ADW (Asynchronous Disk Write) (c) BLD (Bypass Local Disk)

Fig. 5: HDFS Write Flow

two replicas on two different failure domains are enough for fault-tolerance in data centers. In our design, data is also written to the parallel file system asynchronously.

2) Asynchronous Disk Write (ADW): Figure 5(b) shows our proposed scheme to support the ADW mode.

In this design, the HDFS Client writes one copy of data to the local disks on the DataNode asynchronously. The second copy of data is sent to the Memcached server through the `I/O Forwarding Module` by the HDFS Client. In this scheme also, we use the updated hash function to hash the key-value pairs based on the block name prefix rather than the entire key so that all the data packets belonging to the same block are sent to the same Memcached server.

In this scheme, we assume that, the Memcached servers do not have any local SSD; this scheme is applicable on clusters that have no SSD. So the `Persistence Manager` in the Memcached server is not activated by this scheme. Also, this scheme prioritizes data fault-tolerance over write performance. Therefore, the `Lustre Writer` on the Memcached server, writes data synchronously to Lustre. After completion of each `memcached_set()`, data is put to the queue attached to the `Lustre Writer`. Whenever the first packet of a block arrives, a new file is created in Lustre. Just like the `Persistence Manager` as described in Section III-B1, the `Lustre Writer` also caches the file pointers. The pool of threads in the `Lustre Writer` increases overlapping between Memcached operations and Lustre I/O. At the same time, it guarantees that each data block is persisted synchronously.

The map and reduce tasks of a MapReduce job, read data from the local storage or Memcached. In case of failures (data is not available in local storage or Memcached), data is read from Lustre.

This scheme meets the challenges described in Section I as follows:

1) It can reduce I/O bottlenecks by asynchronously writing data to local disks.
2) Improves data locality compared to MapReduce jobs running over Lustre by placing data in the local storage of the DataNodes. By reading data from the local storage and Memcached, this design can achieve better read performance.
3) This scheme synchronously writes data to Lustre. Thus, it can offer same level of fault-tolerance as Lustre.

3) Bypass Local Disk (BLD): Figure 5(c) shows the internal design scheme to support the BLD mode. There are HPC clusters that have large amount of memory on the compute nodes while the local disk space is limited. For this type of cluster, application performance can be largely improved by avoiding disk I/O. Therefore, in this scheme, we propose to place one copy of data to RAM Disks on the compute nodes in a greedy manner. Since dedicating enough memory for storage may not always be feasible for the application, we keep a threshold of RAM Disk usage. When this usage threshold is reached, data is forwarded to the Memcached server by the `I/O Forwarding Module` in the DataNode. We assume that the Memcached servers are not backed by SSD and the key-value pairs are stored on the in-memory slabs only. The other copy of data is synchronously written to Lustre by the `Lustre Writer` in HDFS Client. Thus, in this scheme, the `Lustre Writer` and `Persistence Manager` on the Memcached server are not enabled and it uses the key-value store as a buffer so that the stored data can be read at memory speed. Since persistence of the data in Memcached is not needed, we use the default hash function of libMemcached (hash on the entire key) here to distribute the data across different servers.

For this scheme, we do not send all the data to Memcached; only the data that cannot be fit into the local RAM Disks are sent to the Memcached servers. Therefore, in order to make all the data fault-tolerant we send them to Lustre directly from the HDFS Client side rather than from the Memcached servers.

In this scheme, instead of accessing the parallel file system, MapReduce jobs read data from the local storage or the Memcached servers.

This scheme meets the challenges described in Section I as follows:

1) It can entirely bypass disk I/O on local storage by adaptively switching to place data to Memcached rather than slower local disks.

2) By reading data from the local storage and Memcached, this design can achieve much better data locality and read performance compared to MapReduce run over Lustre.
3) This scheme offers same level of fault-tolerance as Lustre by synchronously writing to Lustre.

D. Applicability of the Proposed Schemes/Modes on HPC Systems

ALW: The ALW mode requires local SSD/NVRAMs on the Memcached servers. This mode is suitable for clusters with heterogeneous storage support.

ADW: This mode is suitable for clusters with limited or poor local disks and no SSDs. Also if the compute nodes do not have the RAM Disk configured or RAM Disk size is very small, this mode is suitable for deployment.

BLD: Clusters that have large amount of memory i.e. RAM Disk on the compute nodes with limited local disk space, BLD scheme is applicable. Both ADW and BLD schemes can perform well with good Lustre installation and guarantee similar fault-tolerance as Lustre.

IV. Performance Evaluation

In this section, we present a detailed performance evaluation of our design in comparison to that of the default HDFS and Lustre. In this study, we perform the following sets of experiments:

(1) Performance analysis with TestDFSIO
(2) Evaluation with RandomWriter and Sort
(3) Evaluation with PUMA [20]

A. Experimental Setup

We use two different clusters for our evaluations.

(1) Intel Westmere Cluster (Cluster A): This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12GB RAM, 6GB RAM Disk, and 160GB HDD. This cluster also has eight large memory nodes. Each of these nodes is equipped with 24 GB RAM, two 1TB HDDs, single 300GB OCZ VeloDrive PCIe SSD. Each node has 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. We use a threshold of 3 GB RAM Disk usage per node on this cluster.

(2) TACC Stampede [28] (Cluster B): We use the Stampede supercomputing system at TACC [28] for our experiments. Each Stampede node is dual socket containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz. It has 32 GB of memory, a SE10P (B0-KNC) co-processor and a Mellanox IB FDR MT4099 HCA (56 Gbps data rate). The host processors are running CentOS release 6.3 (Final). Each node is equipped with a single 80 GB HDD and 16 GB RAM Disk. We use a threshold of 6 GB RAM Disk usage per node on this cluster.

For our evaluations, we use Hadoop 2.6.0, and RDMA-Memcached 0.9.1 [19]. HDFS block size and Lustre stripe size is 128MB. Default HDFS replication factor is set to three.

B. Evaluation with TestDFSIO

In this section we evaluate our designs with the TestDFSIO benchmark.

First, we determine the optimal number of Memcached servers required to evaluate our designs on Cluster A. For this, we vary the number of DataNodes from 10 to 14 and number of Memcached servers from 8 to 4, respectively. For Memcached server deployment, we use the large memory nodes on this cluster and we use 20GB memory for Memcached on each server. We run the TestDFSIO test with 100GB data size using 80 maps. For these experiments, we use the ALW mode proposed in Section III-C1.

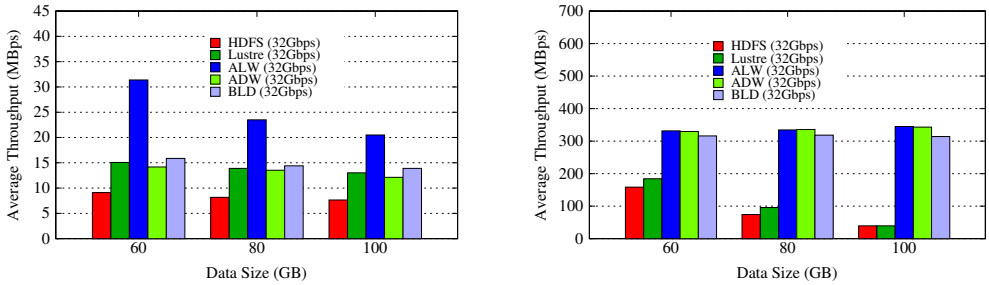
TABLE II: Determining optimal number of Memcached servers

DataNodes	Memcached Servers	Write Throughput	Read Throughput
10	8	12.72 MBps	237.5 MBps
12	6	13.99 MBps	240.6 MBps
14	4	14.03 MBps	67.8 MBps

As observed from Table II, we can achieve the highest read throughput while using 12 DataNodes with 6 Memcached servers. In the ALW mode, we want to buffer the entire dataset in the Memcached servers. Therefore, while using 4 Memcached servers each with 20GB memory the entire dataset does not fit in memory and earlier key-value pairs get dropped to make space for the new ones. The write throughput in this case is the best because it has the largest number of DataNodes and ALW writes one copy of data to local storage and the other to the Memcached servers. But during TestDFSIO read, this experiment reads the entire data from the DataNodes (if data not available locally, goes to other DataNode over IPoIB (32Gbps)) as not all data are available in the Memcached servers. On the other hand, while using 12 DataNodes with 6 Memcached servers, the entire dataset can fit in memory and with this combination of servers, we achieve the highest read throughput with near-optimal write performance. The write throughput in this case is slightly less than that for 14-4 case. This is because due to fewer DataNodes the I/O overhead during write is higher here. But since data is read from both the local storage and Memcached servers over RDMA, the read throughput is much higher in this case.

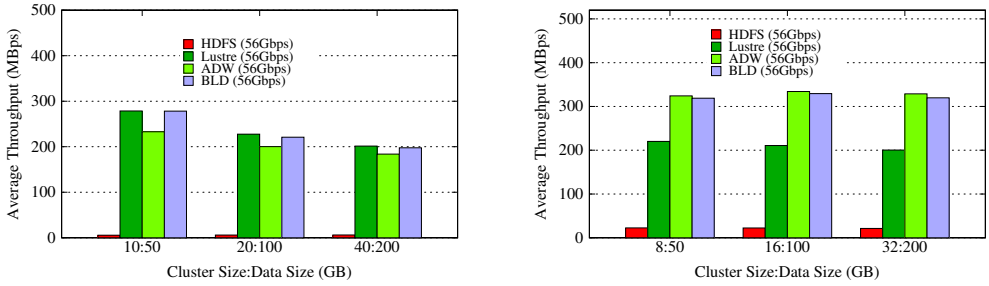
Therefore, in our subsequent experiments on Cluster A, we maintain the 2:1 ratio of DataNodes and Memcached servers and choose the dataset such that the entire data can fit in memory. On Cluster B, we use 4:1 ratio of DataNodes and Memcached servers.

Besides, in our design, we follow a hybrid approach of data read. We read data from local storage when available. In case data is not available locally, we read it from the Memcached servers. The throughput of 40GB data read while using 16 DataNodes and 8 Memcached servers following this hybrid approach is 390.48 MBps. But if data is entirely read from the Memcached servers, this throughput reduces to 304.67 MBps. The reason behind this is, even though in-memory reads reduce the I/O overheads, when data is read entirely from



(a) TestDFSIO Write (b) TestDFSIO Read

Fig. 6: Evaluation of TestDFSIO (Cluster A)



(a) TestDFSIO Write (b) TestDFSIO Read

Fig. 7: Evaluation of TestDFSIO (Cluster B)

Memcached, the read throughput is bounded by the aggregated bandwidth of the Memcached servers. But reading data locally helps reduce the contention for network bandwidth and also scales with the number of DataNodes.

Figure 6 shows the results of our evaluations on Cluster A. While running these experiments with our designs, we use 16 nodes on Cluster A as Hadoop DataNodes and 8 large memory nodes as Memcached servers. In order to have a fair distribution of resources, we use 24 DataNodes while running TestDFSIO on HDFS and Lustre. In all these cases, we perform these experiments with 64 maps and vary the data sizes from 60GB to 100GB. As observed from Figure 6, our ALW mode improves the write throughput by 2.6x over HDFS and 1.5x over Lustre for 100GB data size. Compared to HDFS, our approach can avoid the overhead of replication on local disks. Also, we hide the latency of shared file system access by using the Memcached-based burst buffer layer.

For ADW and BLD, we do not see significant gain in write performance compared to Lustre. Because, Lustre writes only a single copy of data, whereas, in both ADW and BLD we write additional data copies on local storage to introduce locality. The write performance of BLD is slightly better than that of ADW. Because, unlike ADW, BLD can avoid the overhead of Memcached write as long as the data fits in the local RAM Disks. In terms of read performance, all three modes perform better than both HDFS and Lustre with the improvement being up to 8x. The read throughput of ADW is slightly higher than that of BLD on Cluster A. This is because, the RAM Disk size on Cluster A is small and thus locality is reduced during read for BLD mode. However, due to RDMA operations over high performance interconnects, the difference in read performance is not significant between these two modes.

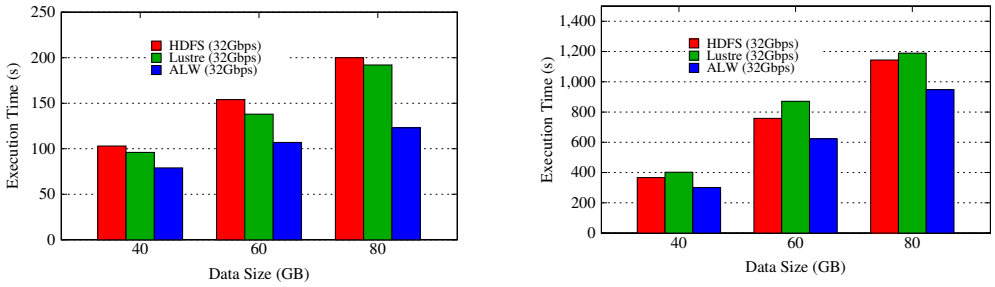
Figure 7 shows the results of our evaluations with TestDFSIO write and read on Cluster B. While running these experiments with our designs, we use 8 DataNodes for Hadoop and 2 servers for Memcached. For experiments with 16 and 32 DataNodes, the numbers of Memcached servers used are 4 and 8, respectively. Each Memcached server uses 25GB of memory. In order to have a fair distribution of resources, we use 10, 20, and 40 DataNodes while running TestDFSIO on HDFS and Lustre. In these experiments, we vary the data sizes from 50GB on 10 nodes to 200GB on 40 nodes. As observed from Figure 7(b), BLD improves the write throughput by 30x over HDFS. On Cluster B, local disk performance is very poor. Thus, bypassing the local disks leads to significant improvement over HDFS. Compared to Lustre, we do not see any obvious improvement in write performance. This is because, in our design (BLD), we write one copy of data to local RAM Disks so that subsequent readers can achieve sufficient data locality. Another copy is written to Lustre for fault-tolerance. In terms of read throughput, our design has up to 64% improvement over Lustre and 15x over HDFS.

These experiments prove that ALW performs best on Cluster A and BLD on Cluster B (write better than ADW and requires less memory for Memcached). Therefore, in our subsequent experiments, we use ALW (BLD) on Cluster A (B).

C. Evaluation with RandomWriter and Sort

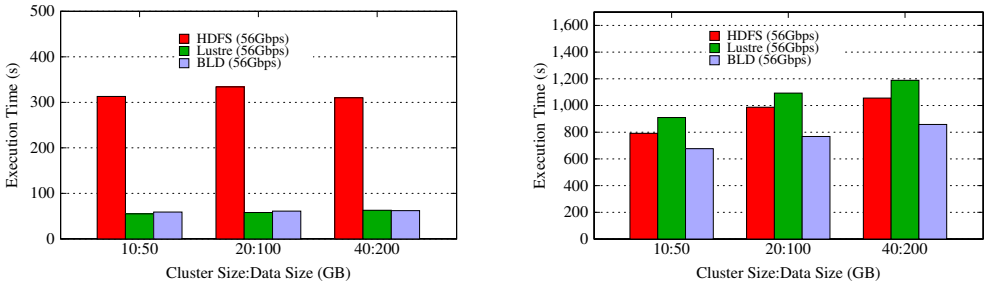
In this section, we evaluate our designs with RandomWriter and Sort benchmarks. RandomWriter is write-intensive, while Sort has equal amount of reads and writes.

Figure 8 shows the performances of RandomWriter and Sort on Cluster A using the ALW mode. We perform these



(a) RandomWriter (b) Sort

Fig. 8: Evaluation of RandomWriter and Sort (Cluster A)



(a) RandomWriter (b) Sort

Fig. 9: Evaluation of RandomWriter and Sort (Cluster B)

experiments on 24 nodes for HDFS and Lustre. For our design, we use 16 DataNodes and 8 Memcached servers each using 20GB memory and backed by SSD. We run 4 concurrent maps per host and vary the data size from 40GB to 80GB. Therefore, for HDFS and Lustre, a total of 96 maps write the same amount of data as 64 maps in our design. As observed from the figure, even with less concurrency, ALW reduces the execution time of RandomWriter by up to 36% over Luster and 39% over HDFS. As observed from Figure 8, our design improves Sort performance by 20% over Lustre and by 18% over HDFS.

TABLE III: Performance comparison of RandomWriter and Sort with 4 and 6 maps per host using ALW

	RandomWriter		Sort	
Data Size (GB)	ALW-4	ALW-6	ALW-4	ALW-6
40	79	68	301	318
60	107	94	623	638
80	123	112	948	967

As observed from Table III, with a total of 96 maps (6 maps per host) like HDFS and Lustre, our design can further reduce the execution time of RandomWriter. Increasing the concurrency does not help improve the performance of Sort. This is because, the major bottleneck of Sort lies in the shuffle phase and with increasing concurrency, the level of locality reduces. Even then, our design has up to 15% gain over HDFS and up to 19% gain over Lustre.

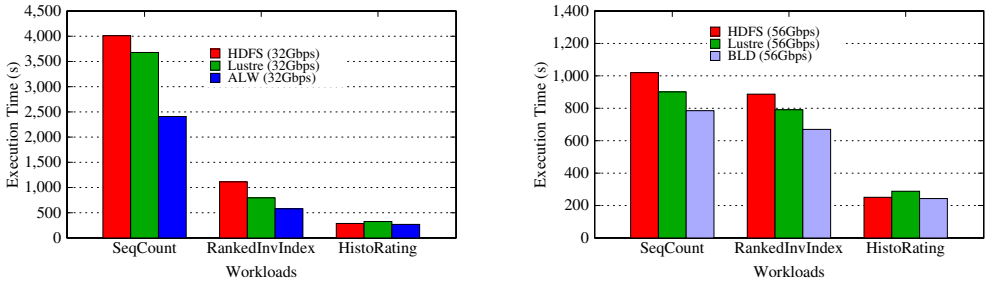
Figure 9 shows the performances of RandomWriter and Sort on Cluster B using the BLD mode. We perform these experiments on 10, 20, and 40 nodes for HDFS and Lustre. For our design, we use 8, 16, and 32 DataNodes with 2, 4, and 8 Memcached servers, respectively, each using 28GB memory. We run 4 maps per host and vary the data size from 50GB to 100GB. Therefore, for HDFS and Lustre on 20 nodes, a total of 80 maps write the same amount of data as 64 maps in our design (16 DataNodes). As observed from the figure, BLD reduces the execution time of RandomWriter by up to 5x over HDFS. But it does not improve the execution time over Lustre. This is because BLD writes one additional copy of data in local RAM Disks compared to Lustre. But BLD improves the performance of Sort by 28% over Lustre and by 19% over HDFS.

TABLE IV: Performance comparison of RandomWriter and Sort with 4 and 6 maps per host using BLD

	RandomWriter		Sort	
Data Size (GB)	BLD-4	BLD-5	BLD-4	BLD-6
50	59	53	677	721
100	61	59	768	801
200	62	59	858	893

In these evaluations, we run 4 maps per host. Therefore, for HDFS and Lustre more concurrent maps run than that in our design. Running BLD with 5 concurrent maps per host can make the total number of maps equal in all the cases. Table IV shows the performance of BLD with 4 maps and 5 maps per host.

As observed from the table, with increased number of concurrent maps, the execution time of RandomWriter reduces. But increasing the concurrency does not help improve the performance of Sort. This is because, the major bottleneck of Sort lies in the shuffle phase and with increasing concurrency, the level of locality reduces. Even then, our design has up to 15% gain over HDFS and up to 25% gain over Lustre.



(a) Workloads (Cluster A)

(b) Workloads (Cluster B)

Fig. 10: Evaluation with PUMA [20]

D. Evaluation with PUMA

In this section, we evaluate the performance of different PUMA workloads like SequenceCount, RankedInvertedIndex and Histogram_rating. On Cluster A, we use the ALW mode and on Cluster B, we use BLD.

On Cluster A, our design has up to 34.5% improvement over Lustre and 40% over HDFS for SequenceCount. For RankedInvertedIndex, the gain is 48% over HDFS and 27.3% over Lustre. Our gain for Histogram_rating is up to 17% on this cluster.

On the other hand, on Cluster B, our design has up to 13% improvement over Lustre and 23% over HDFS for SequenceCount. For RankedInvertedIndex, the gain is 34% over HDFS and 16% over Lustre. Our gain for Histogram_rating is up to 16% over Lustre.

The SequenceCount workload has almost thrice as much output as input. Thus, it is a write-intensive application. RankedInvertedIndex also has equal amount of reads and writes. So our design has significant gains compared to both HDFS and Lustre for these workloads. Because, in addition to reducing the I/O overheads, our design improves locality that enhances the read performance. The amount of write is less than that of read for Histogram_rating. But our design gains mainly due to improved read performance in this case. However, Lustre on Cluster B uses InfiniBand (56Gbps) verbs interface, whereas on Cluster A, the interconnect to access Lustre is IPoIB (32Gbps). Lustre installation on Cluster B is also much larger with higher number of I/O nodes than that on Cluster A. Cluster B also has large memory on the compute nodes. For these reasons, the performance of the workloads over Lustre are better on Cluster B. Therefore, on Cluster A, our design has higher gain over Lustre compared to that in Cluster B.

V. Related Work

Extensive research has been carried out in the recent past to improve the performance of Hadoop distributed file system. The default HDFS architecture has already been analyzed for its drawbacks and limitations. For instance, Shvacho [27] has studied correlation between HDFS design goals and the feasibility of achieving them with current system architecture. By analyzing the performance of HDFS, Shafer et al. [25] identified three critical issues including architectural bottlenecks that cause delays in scheduling new tasks, portability limitations posed by the Java implementation and assumptions about storage management. Our initial works proposed RDMA-enhanced HDFS designs [15, 16] to improve HDFS performance with pipelined and parallel replication schemes, respectively. But these designs kept the default HDFS architecture intact. In our recent work [14], we proposed a SEDA (Staged Event-Driven Architecture) [33] based approach to redesign HDFS architecture (SOR-HDFS). Through SOR-HDFS, we have significantly eliminated the architectural bottlenecks in HDFS for maximizing the possible overlapping during different stages, like communication, processing, and I/O. But it still encounters I/O performance bottlenecks during both HDFS read and write operations by accessing disks.

For HDFS, the read throughput has been improved by caching data in memory or using explicit caching systems [4, 5, 9]. The Apache Hadoop community [5] has also proposed such kind of centralized cache management scheme in HDFS, which is an explicit caching mechanism that allows users to specify paths to be cached by HDFS. Singh et. al [9] presented a dynamic caching mechanism for Hadoop by integrating HDFS with Memcached. However, they also focused on HDFS read operation performance only. In [12], we presented an in-memory design for HDFS using Memcached that improves both the read and write performances. Recently, the new in-memory computing systems, like Spark [35], are emerging. Spark caches the intermediate data into memory and provides the abstraction of Resilient Distributed Datasets (RDDs) [34] to support lineage based data recovery (re-computation). However, RDDs are stored in the JVM heap, and cannot be shared across jobs. And also, the re-computation based data recovery is more expensive than replication [34]. Some recent studies [22, 23] also pay attention to incorporate heterogeneous storage media (e.g. SSD) in HDFS. Authors in [22] deal with data distribution in the presence of nodes that do not have uniform storage characteristics; whereas [23] caches data in SSD. Researchers in [30] present HDFS-specific optimization for PVFS and [24] propose to store cold data of Hadoop cluster to Network-attached file systems. In our recent work [21], we propose an RDMA-based design of YARN MapReduce over Lustre that stores the intermediate data of MapReduce jobs in Lustre. This work also shows that Lustre read performance degrades with increasing concurrency. In [13], we propose an integrated design of HDFS with Lustre that guarantees better data locality for MapReduce jobs compared to that of

Lustre. However, none of these works propose any solution to minimize the bottlenecks of parallel file system access for data-intensive applications. Therefore, in this paper, we considered different aspects of locality, fault-tolerance and I/O for Big Data applications and proposed to integrate HDFS with Lustre through a high-performance key-value store.

VI. Conclusion

In this paper, we proposed to integrate HDFS with Lustre through RDMA-based key-value store. We design a burst buffer system for Big Data analytics applications using RDMA-based Memcached [17] and integrate HDFS with Lustre through this high-performance buffer layer. We also present three schemes for integrating HDFS with Lustre considering different aspects of I/O, data locality and fault-tolerance. Performance evaluations show that our design can improve the performance of TestDFSIO write by up to 2.6x over HDFS and 1.5x over Lustre. The gain in read throughput is up to 8x. The execution time of Sort is reduced by up to 28% over Lustre and 19% over HDFS. The performances of PUMA workloads like SequenceCount, RankedInvertedIndex, and Histogram_rating are also significantly improved by our design.

In future, we plan to design our burst buffer system with NVRAM and evaluate Big Data workloads with this design.

References

[1] "Digital Universe Invaded By Sensors," http://www.emc.com/about/news/press/2014/20140409-01.htm.

[2] "Hadoop and Big Data," http://www.cloudera.com/content/cloudera/en/about/hadoop-and-big-data.html.

[3] "Memcached: High-Performance, Distributed Memory Object Caching System," http://memcached.org/.

[4] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[5] Apache Software Foundation, "Centralized Cache Management in HDFS," http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[6] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems," in *VLDB Endowment*, 2012.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Operating Systems Design and Implementation (OSDI)*, 2004.

[8] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, 2004.

[9] G. Singh and P. Chandra and R. Tahir, "A Dynamic Caching Mechanism for Hadoop using Memcached," https://wiki.engr.illinois.edu/download/attachments/197297260/ClouData-3rd.pdf.

[10] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Analysis of HDFS Under HBase: A Facebook Messages Case Study," in *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[11] IDC, "The Internet of Things," http://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm.

[12] N. S. Islam, X. Lu, M. W. Rahman, R. Rajachandrasekar, and D. K. Panda, "In-Memory I/O and Replication for HDFS with Memcached: Early Experiences," in *2014 IEEE International Conference on Big Data (IEEE BigData)*, 2014.

[13] N. S. Islam, X. Lu, M. W. Rahman, D. Shankar, and D. K. Panda, "Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[14] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "SOR-HDFS: A SEDA-based Approach to Maximize Overlapping in RDMA-Enhanced HDFS," in *23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[15] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High Performance RDMA-based Design of HDFS over InfiniBand," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[16] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "Can Parallel Replication Benefit Hadoop Distributed File System for High Performance Interconnects?" in *IEEE 21st Annual Symposium on High-Performance Interconnects (HOTI)*, 2013.

[17] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *International Conference on Parallel Processing (ICPP)*, 2011.

[18] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *MSST/SNAPI*, 2012.

[19] OSU NBC Lab, "High-Performance Big Data (HiBD)," http://hibd.cse.ohio-state.edu.

[20] Purdue MapReduce Benchmarks Suite (PUMA), https://sites.google.com/site/farazahmad/pumabenchmarks.

[21] M. W. Rahman, X. Lu, N. S. Islam, R. Rajachadrasekar, and D. K. Panda, "High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA," in *29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[22] K. Ravindranathan, A. Anwar, and A. Butt, "hatS: A Heterogeneity-Aware Tiered Storage for Hadoop," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2014.

[23] K. Ravindranathan, S. Iqbal, and A. Butt, "VENU: Orchestrating SSDs in Hadoop Storage," in *2014 IEEE International Conference on Big Data (IEEE BigData)*, 2014.

[24] K. Ravindranathan, A. Khasymski, A. Butt, S. Tiwari, and M. Bhandarkar, "AptStore: Dynamic Storage Management for Hadoop," in *International Conference on Cluster Computing (CLUSTER)*, 2013.

[25] J. Shafer, S. Rixner, and A. L. Cox, "The Hadoop Distributed Filesystem: Balancing Portability and Performance," in *Internation Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[27] K. Shvachko, "HDFS Scalability: The Limits to Growth," 2010. [Online]. Available: http://www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf

[28] Stampede at TACC, http://www.tacc.utexas.edu/resources/hpc/stampede.

[29] T. Sterling, E. Lusk, and W. Gropp, *Beowulf Cluster Computing with Linux*. Cambridge, MA, USA: MIT Press, 2003.

[30] W. Tantisiriroj, S. Patil, G. Gibson, S. Son, S. Lang, and R. Ross, "On the Duality of Data-intensive File System Design:Reconciling HDFS and PVFS," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[31] The Apache Software Foundation, "The Apache Hadoop Project," http://hadoop.apache.org/.

[32] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and Optimization of Memory-Resident MapReduce on HPC Systems," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[33] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[36] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A Reliable and Highly-Available Non-Volatile Memory System," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.